Module 3

Contents of Module 3

- Function: Built in libraries, Parameter passing in functions, call by value, Passing arrays to functions: call by reference
- Recursion: Example programs, Finding Factorial, Fibonacci series, Ackerman function
- Sorting: Quick sort or Merge sort.

Objectives of Module 3

- Learn function and their types.
- Compare inbuilt and user defined functions.
- Use recursion technique to solve problems.
- Learn basic idea of sorting.

Function

- In C programming, a function is a self-contained block of code that performs a specific task or set of tasks. Functions allow you to break down a complex program into smaller, manageable pieces or modules. Each function can focus on a specific task or subtask, making the code easier to understand, test, and maintain. Every C program has at least one functions, which is main( ).
- Once you define a function, you can reuse it in multiple places within your program without duplicating code. This reduces redundancy and helps ensure consistency in your code.

Syntax

```
return_type function_name(arguments)
{
        statements;
}
```

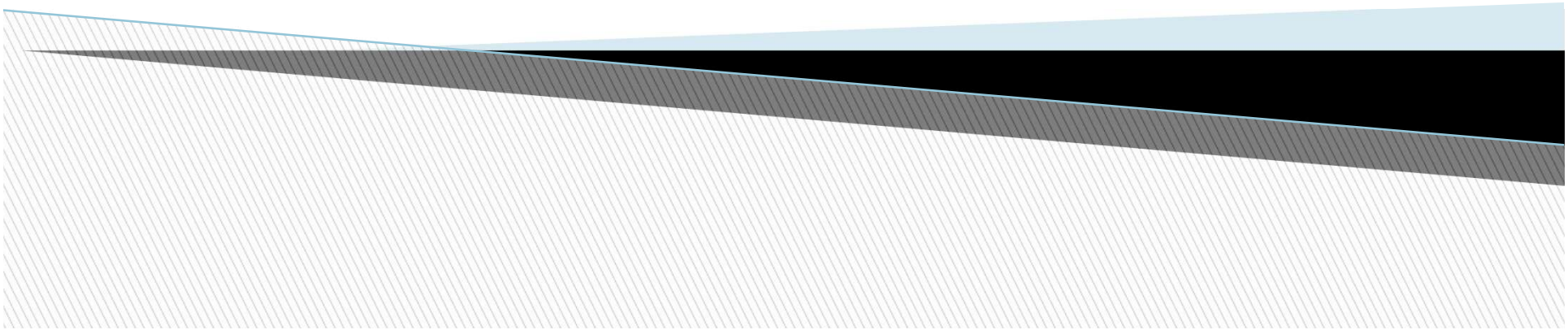
(Here, return type can be void, int, float, char etc.)

Arguments: In C programming, "arguments" refer to the values or expressions that we pass to a function when we call it. These arguments are used by the function to perform its task or calculations. Arguments provide a way to pass data into a function, allowing us to work with different data each time we call the same function.

CONTD..

Function Declaration: A function must be declared before it is used in the program. The declaration typically includes the function's return type, name, and a list of parameters (if any).

Example: int add(int a, int b); // Function declaration

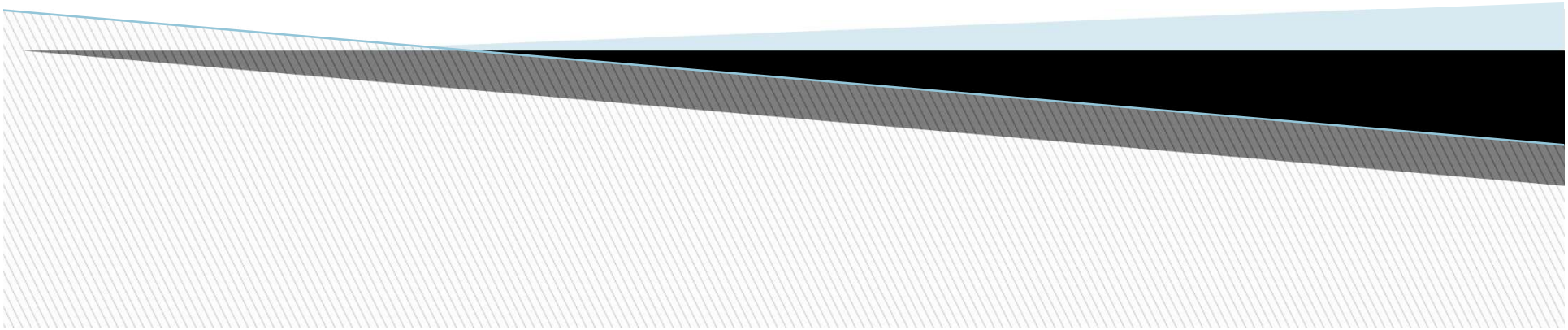


CONTD..

Function Definition: The actual implementation of a function is called its definition. It includes the function's return type, name, parameters, and the code block (function body) enclosed in curly braces.

Example:

```
int add(int a, int b) {
return a + b; // Function definition
}
```

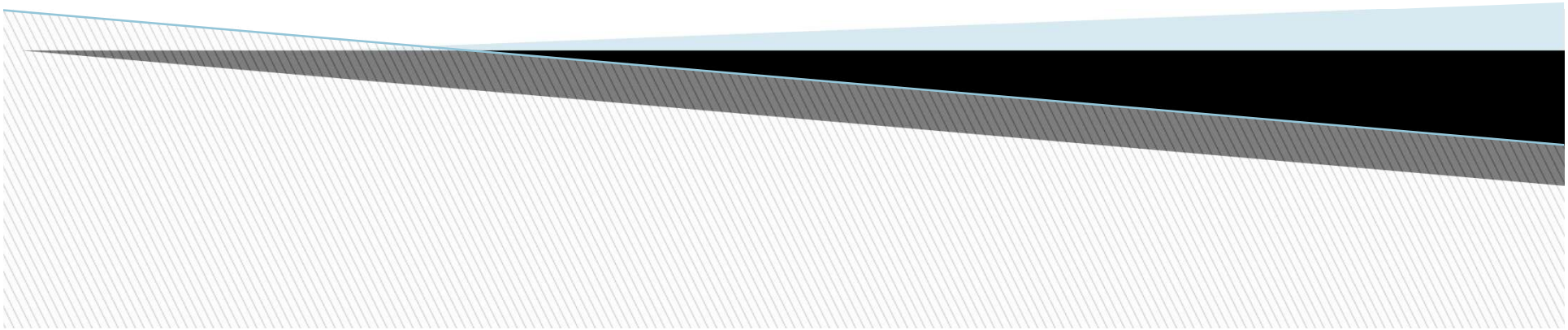


CONTD..

Function Call: To execute a function, you call it by its name.

Example:

```
int result = add(5, 3); // Function call
```

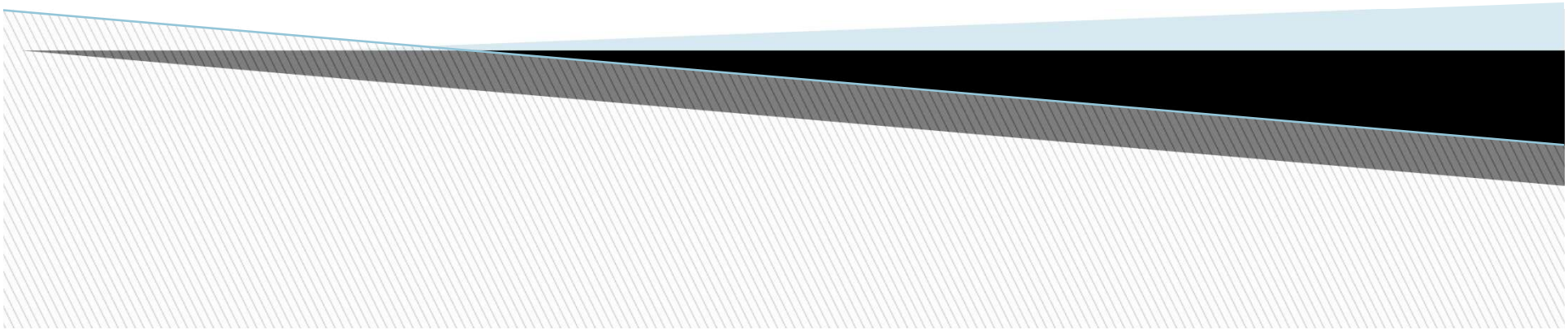


Addition of two numbers using Function

```
#include <stdio.h>
int add(int a, int b);
int main() {
int a, b, result;
int add(int a, int b) {          // Function to add two numbers
return a + b;
}
printf("Enter the two numbers: ");
scanf("%d%d", &a,&b);
result = add(a, b);  // Call the add function and store the result
printf("The sum of %d and %d is %d\n", a, b, result);
return 0;
}
```

Output..

```
Enter the two numbers: 70 40
The sum of 70 and 40 is 110
```



Types of Function

Mainly there are two types of function-

- **Pre-defined/ Standard Library Functions:** These are built-in functions provided by the C Standard Library. They perform various tasks and are available for use without the need for explicit declaration or definition. Like- **printf(), scanf()** etc.

- **User-Defined Functions:** These are functions created by the user to perform specific tasks. They are defined by the user and can be called from other parts of the program. User-defined functions help modularize code and improve code readability.

Function Prototype

A function prototype is a special type of function declaration that also specifies the function's name, return type, and parameter types. In addition to this, a function prototype typically appears at the beginning of a program and it provides information to the compiler about functions that are defined elsewhere in the program. It is just the declaration of the function without any body.

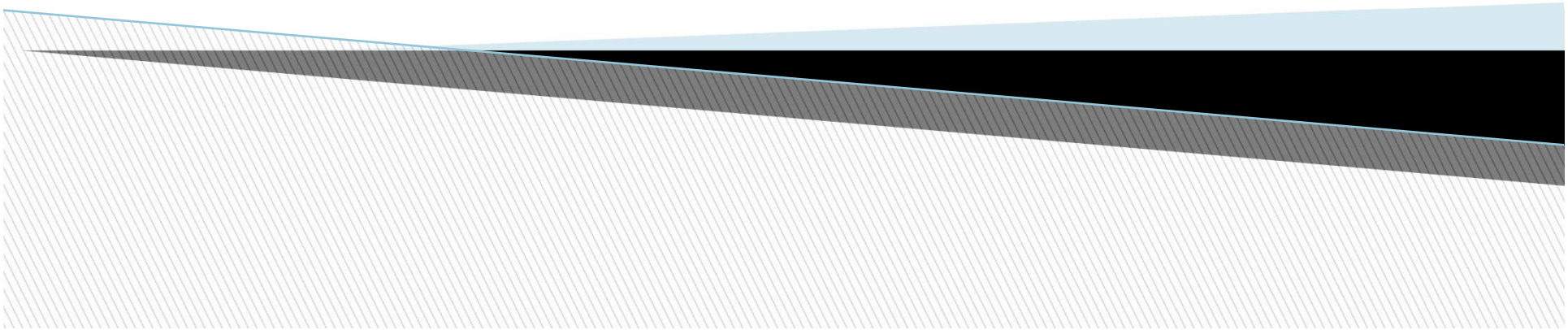
CONTD..

Purpose:

- **Forward Declaration**: Function prototypes are often used for forward declaration. They allow you to declare functions at the beginning of a program before the actual function definition. This is useful when you need to use a function before it's defined in the code.

CONTD..

- **Type Checking**: Function prototypes help in type checking. They ensure that you use the correct number and types of arguments when calling a function.



Example:

```
// Function prototype
int add(int a, int b);
// Function definition
int add(int a, int b) {
    return a + b;
}
// Function call
int result = add(5, 3);
```

Recursion

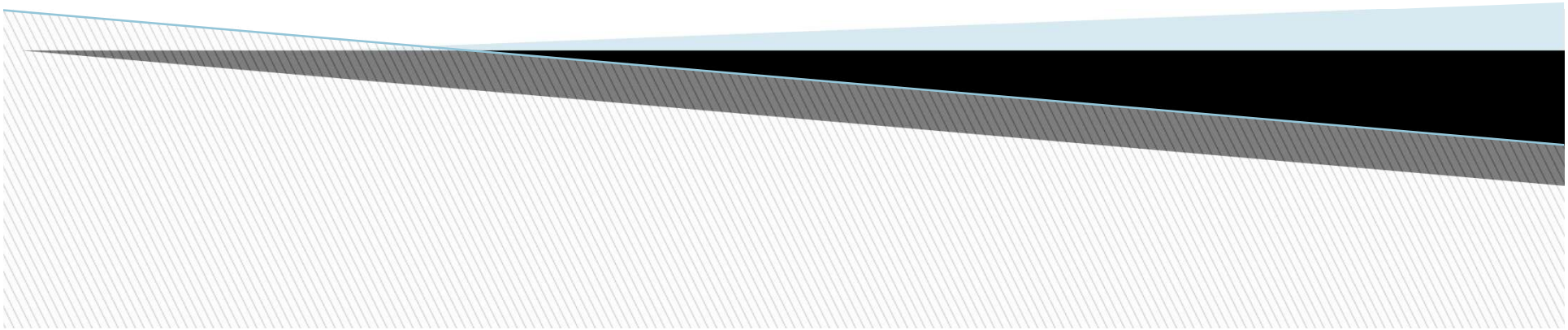
- Recursion is a programming technique where a function **calls itself** to solve a problem. In C, a recursive function is a function that invokes itself either directly or indirectly to solve a problem.
- A termination condition is required to make it finite.
- Any function which calls itself is called recursive function, and such function calls are called recursive calls. Recursion cannot be applied to all the problem, but it is more useful for the tasks that can be defined in terms of similar subtasks. For Example, recursion may be applied to sorting problems.

Factorial of a number using recursion

```
#include <stdio.h>
int factorial(int n) {
if (n == 0 || n == 1) {
        return 1;
    } else {
return n * factorial(n - 1);
    }
}
int main() {
    int num, result;
    printf("Enter a positive integer: ");
    scanf("%d", &num);
 if (num < 0) {
        printf("Factorial is not defined for negative numbers.\n");
    } else {
        int result = factorial(num);
        printf("Factorial of %d is %d\n", num, result);
    }
  return 0;
}
```

CONTD..

Enter a positive integer: 6
Factorial of 6 is 720
Enter a positive integer: -1
Factorial is not defined for negative numbers.

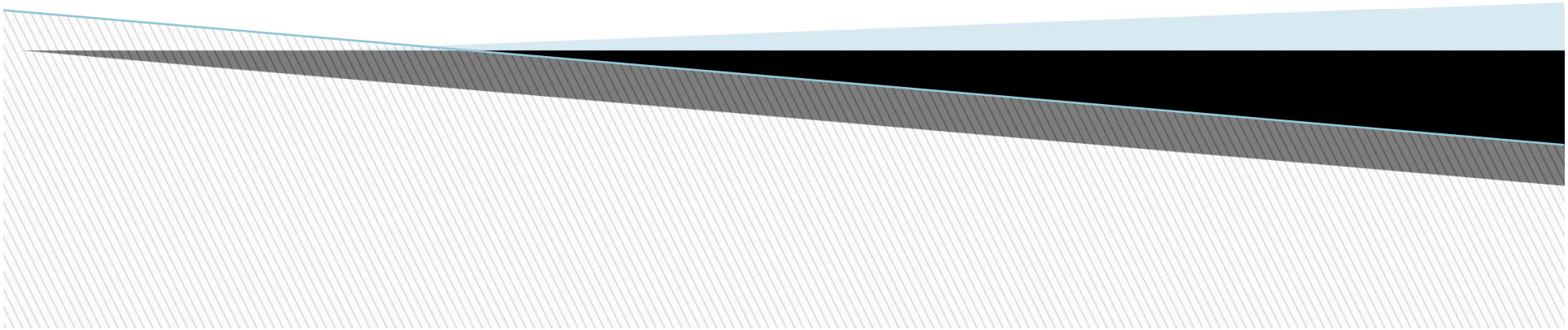


Fibonacci Series using recursion in C


```
#include <stdio.h>
int fibonacci(int n) {
if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
return fibonacci(n - 1) + fibonacci(n - 2);
    }
}int main() {
    int num;
    printf("Enter the number of Fibonacci terms to generate: ");
    scanf("%d", &num);
if (num < 0) {
        printf("Please enter a non-negative integer.\n");
    } else {
        printf("Fibonacci Series: ");
        for (int i = 0; i < num; i++) {
            printf("%d ", fibonacci(i));
        }
}
 return 0;
}
```

Output

```
Enter the number of Fibonacci terms to generate: 10
Fibonacci Series: 0 1 1 2 3 5 8 13 21 34
```

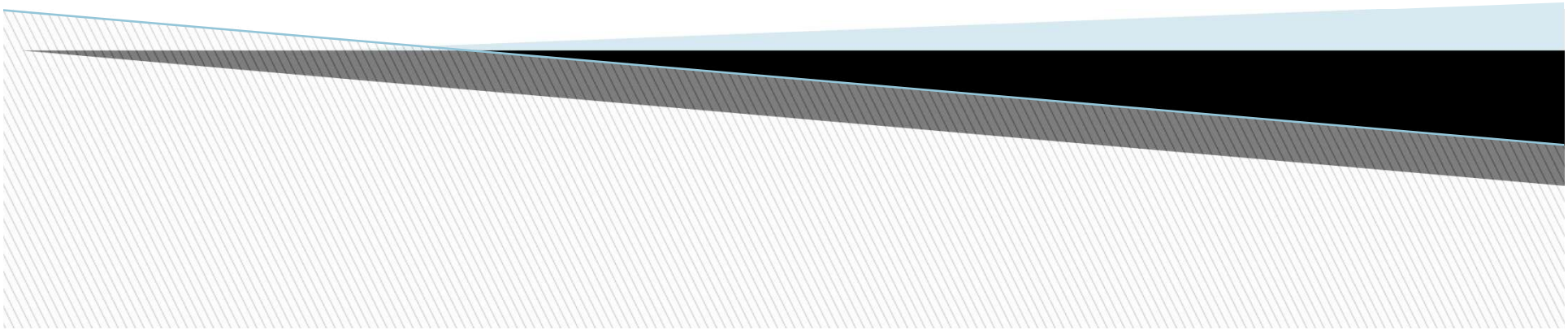


Calculate the sum of the first n natural numbers using recursion


```
#include <stdio.h>
int sumOfNaturalNumbers(int n) {
    if (n == 0) {
        return 0;
    } else {
        return n + sumOfNaturalNumbers(n - 1);
}
}
int main() {
    int num; Result
    printf("Enter a positive integer: ");
    scanf("%d", &num);
  if (num < 0) {
        printf("Please enter a positive integer.\n");
    } else {
        Result = sumOfNaturalNumbers(num);
        printf("Sum of the first %d natural numbers is %d\n", num, Result);
    }
return 0;
}
```

CONTD..

```
Enter a positive integer: 10
Sum of the first 10 natural numbers is 55
Enter a positive integer: -10
Please enter a positive integer.
```



Sum of Array Elements

```
#include <stdio.h>
int main() {
 int n, sum;
printf("Enter the number of elements in the array: ");
scanf("%d", &n);
if (n <= 0) {
        printf("Invalid input. Please enter a positive integer.\n");
        return 1; }
int arr[n];
printf("Enter %d elements of the array:\n", n);
    for (int i = 0; i < n; i++) {
        printf("Element %d: ", i + 1);
        scanf("%d", &arr[i]);
        sum += arr[i];
    }
printf("Sum of array elements: %d\n", sum);
return 0;
}
```

OUTPUT

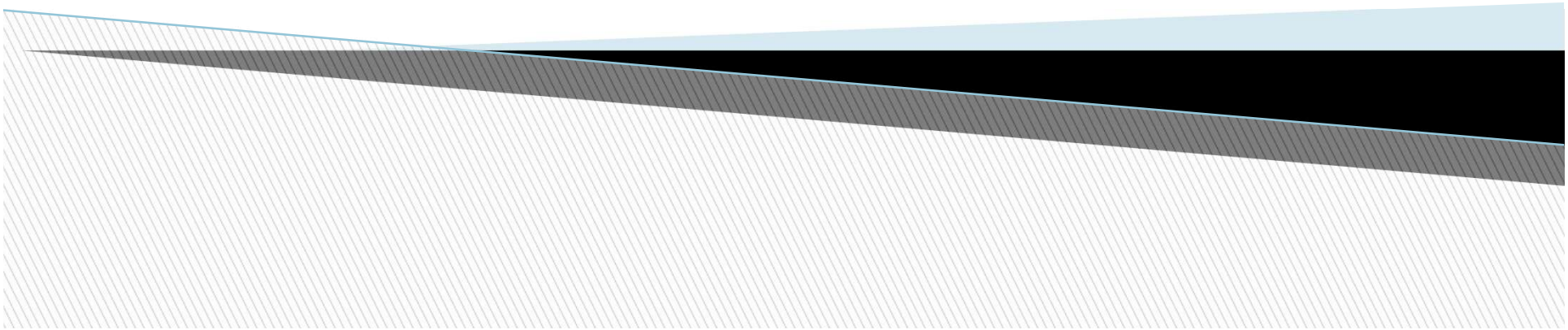

```
Enter the number of elements in the array: 3
Enter 3 elements of the array:
Element 1: 2
Element 2: 4
Element 3: 0
Sum of array elements: 6
```

Write a C program to find out the average height of persons

```
#include <stdio.h>
int main() {
    int n;
    double sum;
    double average;
   printf("Enter the number of persons: ");
    scanf("%d", &n);
   if (n <= 0) {
    printf("Invalid input. Number of persons must be greater than 0.\n");
        return 1;
    }
double heights[n];
    for (int i = 0; i < n; i++) {
        printf("Enter the height of person %d (in centimeters): ", i + 1);
        scanf("%lf", &heights[i]);
        sum += heights[i];
 }
average = sum / n;
 printf("Average height of %d persons is: %.2lf centimeters\n", n, average);
return 0;
}
```

Output

```
Enter the number of persons: 5
Enter the height of person 1 (in centimeters): 100
Enter the height of person 2 (in centimeters): 200
Enter the height of person 3 (in centimeters): 300
Enter the height of person 4 (in centimeters): 400
Enter the height of person 5 (in centimeters): 500
Average height of 5 persons is: 300.00 centimeters
```

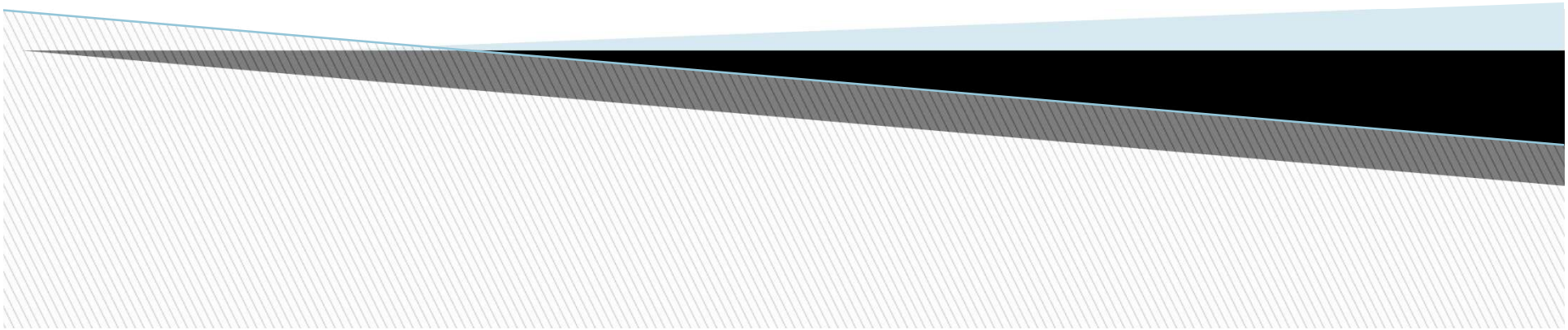


A C program to get the largest element of an array using the function

```
#include <stdio.h>
int findLargest(int arr[], int size) {
  int largest = arr[0];
for (int i = 1; i < size; i++) {
       if (arr[i] > largest) {
          largest = arr[i];
       }
   }
return largest;
}
int main() {
   int n;
   printf("Enter the size of the array: ");
   scanf("%d", &n);
   int arr[n];
   printf("Enter %d elements:\n", n);
   for (int i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }
int largest = findLargest(arr, n);
   printf("The largest element in the array is: %d\n", largest);
 return 0;
}
```

Output

Enter the size of the array: 4
Enter 4 elements:
1
2
3
4
The largest element in the array is: 4



Sorting

Sorting is a fundamental operation in computer science and is used to arrange elements in a specific order.

Example:

int A[] = { 5, 4, 10, 2, 30, 45, 34, 14, 18, 9)

The Array sorted in ascending order will be given as:

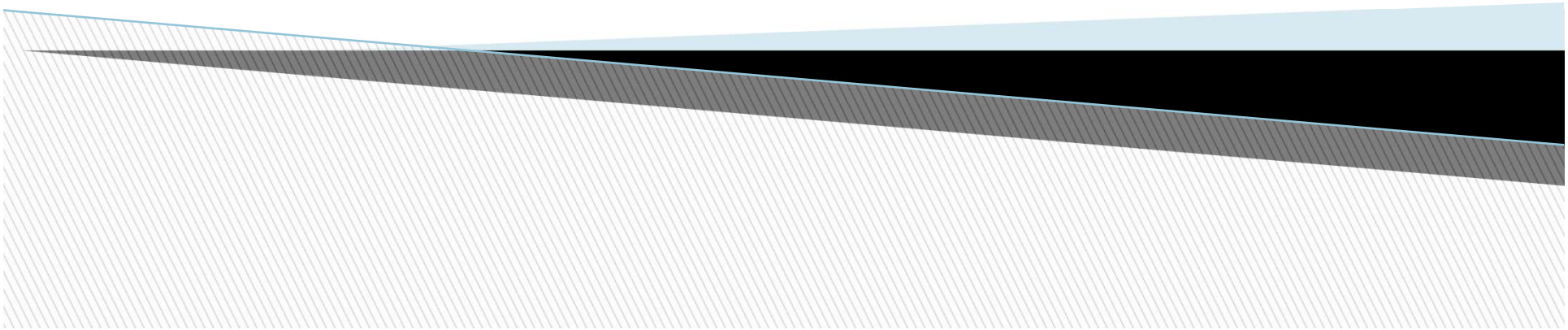
int A[] = { 2, 4, 5, 9, 10, 14, 18, 30, 34, 45 }

Types:

- Bubble Sort
- Merge Sort
- Quick Sort

Bubble Sort

Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order. Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world. It is not suitable for large data sets.

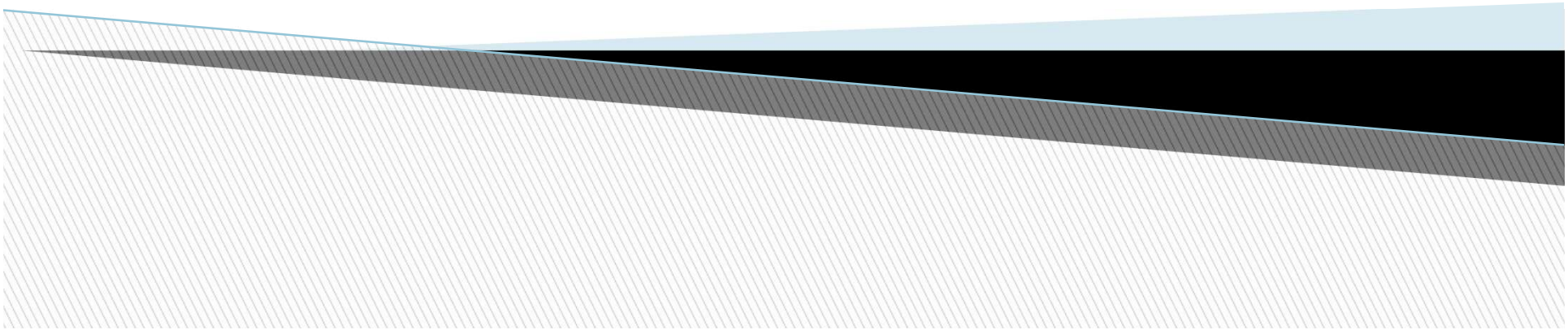


Algorithm:

- Start with an unsorted list of elements.
- Begin a pass through the list by comparing the first two elements (the 0th and 1st elements).
- If the first element is greater than the second element, swap them.
- Move to the next pair of elements (the 1st and 2nd elements) and repeat the comparison and swap if necessary.
- Continue this process, comparing and swapping adjacent elements as you move through the list. After the first pass, the largest element will have "bubbled up" to the end of the list.
- Repeat steps 2-5 for the entire list, excluding the last element (which is already in its correct position after the first pass).

CONTD..

- After the first pass, the largest element is in its correct position at the end of the list. On the second pass, focus on the remaining unsorted portion of the list (excluding the last element), and repeat steps 2-5 to move the second-largest element to its correct position.
- Continue this process for all remaining elements, one at a time, until the entire list is sorted.
- The algorithm terminates when no more swaps are needed during a pass, which indicates that the list is fully sorted.



Working of Bubble Sort Algorithm

Let's take an unsorted array.

13	32	26	35	10

First Pass

Sorting will start from the initial two elements. Let compare them to check which one is greater.

13	32	26	35	10

CONTD..

Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.

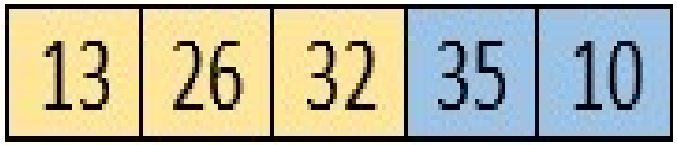
Here, 26 is smaller than 36. So, swapping is required. After swapping new array will look like -

CONTD..

Now, compare 32 and 35.

Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.
Now, the comparison will be in between 35 and 10.

CONTD



Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be -

CONTD...

The same process will be followed for second iteration.

13	26	32	10	35

13	26	32	10	35

13	26	32	10	35

Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -

13	26	10	32	35

13	26	10	32	35

CONTD..

Now, move to the third iteration.

Third Pass:

The same process will be followed for third iteration.

13	26	10	32	35

13	26	10	32	35

Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -

13	10	26	32	35

13	10	26	32	35

13	10	26	32	35

CONTD..

Now, move to the fourth iteration.

10	13	26	32	35

Hence, there is no swapping required, so the array is completely sorted.

Advantages

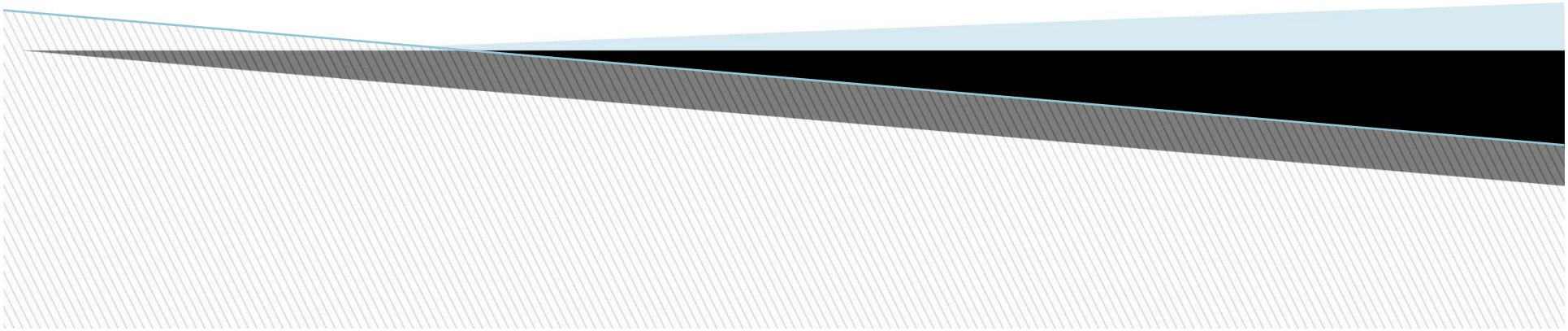
- **Ease of Implementation:** Bubble sort is one of the simplest sorting algorithms to understand and implement.
- **No Extra Memory Requirement:** It doesn't require additional memory or storage to sort the elements. This can be beneficial in situations where memory usage is a big concern.
- **Small Dataset Efficiency:** Bubble sort can be efficient for sorting small datasets or lists with only a few elements. In such cases, its simplicity and low overhead may make it competitive with more complex sorting algorithms.

Disadvantages:

- Bubble sort is a comparison-based sorting algorithm, which means that it requires a comparison operator to determine the relative order of elements in the input data set. It can limit the efficiency of the algorithm in certain cases.
- **Inefficient for Large Datasets:** Bubble sort becomes extremely slow for sorting large datasets, making it impractical for real-world applications where sorting efficiency is a critical factor.
- **Obsolete in Most Real-World Applications:** Bubble sort has largely been replaced by more efficient sorting algorithms like quicksort, merge sort in real-world applications. These algorithms offer better performance across a wide range of scenarios than bubble sort.

Merge Sort

Merge sort is using the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal(or nearby equal) halves, calls itself for the two halves and then merges the two sorted halves. We have to define the **merge()** function to perform the merging.



Algorithm

Step 1: Divide

1.1. Begin with an unsorted array we want to sort.

1.2. Divide the array into two equal (or nearly equal) halves.

1.3. Recursively apply the merge sort algorithm to each of the two halves. This division and recursion continue until we have subarrays with one elements, as these are considered sorted.

Step 2: Conquer

2.1. The conquer step happens implicitly as part of the recursion. As the algorithm divides the array into smaller subarrays, it eventually reaches subarrays with one element, which are considered sorted by definition.

CONTD..

Step 3: Merge

3.1. Merge the sorted subarrays back together into a single, sorted array. This is done by comparing elements from the two subarrays and placing them in the correct order in a temporary storage (usually a new array).

3.2. Start with two pointers, one for each subarray, initially pointing to the first element of each subarray.

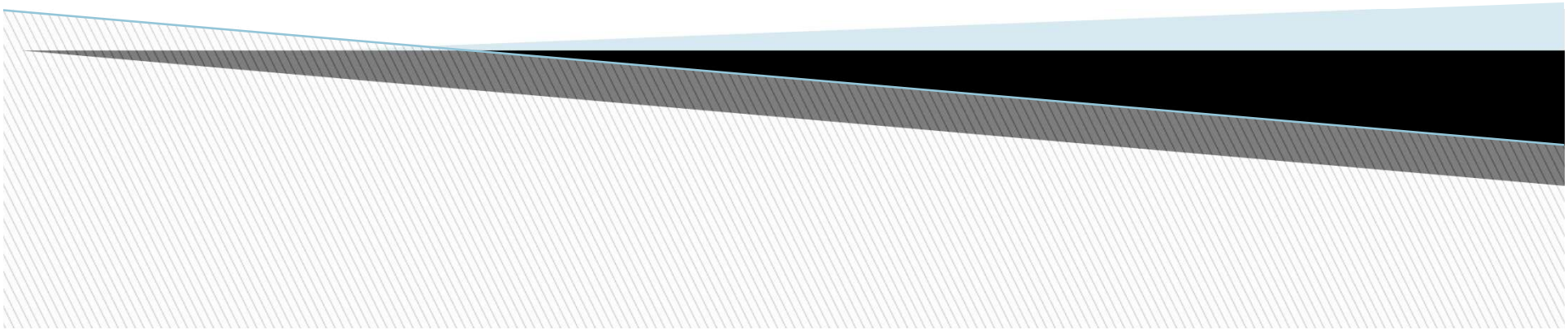
3.3. Compare the elements at the current positions of the two pointers. Take the smaller element and place it in the temporary storage. Move the pointer one position forward in the subarray from which we took the element.

CONTD..

3.4. Repeat the comparison and placement process (3.3) until you have processed all elements from both subarrays.

3.5. If one of the subarrays has remaining elements, simply copy them into the temporary storage, as they are already sorted.

3.6. Copy the merged elements from the temporary storage back into the original array, overwriting the unsorted elements with the sorted ones.



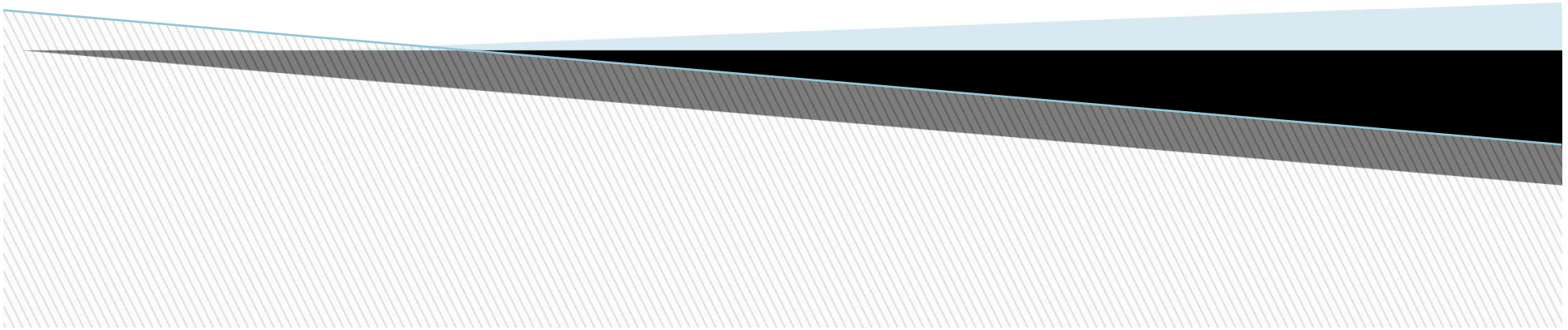
CONTD..

Step 5: Repeat (if necessary)

5.1. If there are more divisions to be made; repeat the process from Step 1 on each of the subarrays generated in Step 1.

NOTE:

The merge sort algorithm continues recursively dividing, conquering, merging, and copying back until the entire array is sorted.



Advantages

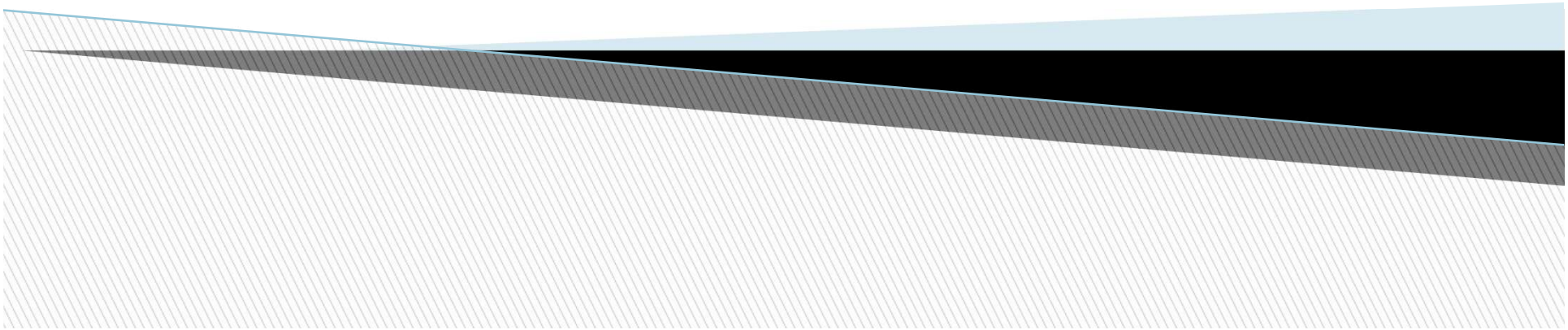
- **Efficiency for Large Datasets:** Merge sort is particularly efficient for sorting large datasets. Its divide-and-conquer approach allows it to efficiently sort large arrays or lists by breaking them into smaller, manageable subproblems. This is in contrast to some other sorting algorithms, like quicksort, which can perform poorly on large datasets in the worst case.
- **Parallelization:** Merge sort is well-suited for parallelization, as you can easily divide the sorting task into subproblems that can be processed concurrently. This improving sorting speed even further.
- **Adaptability:** Merge sort can be easily adapted to handle various data structures, including arrays.

Disadvantages:

- **Space Complexity:** Merge sort requires additional memory for the temporary storage of subarrays during the merge phase. This space requirement can be a drawback, especially when working with limited memory resources.
- **Slower on Small Lists:** Merge sort can be less efficient than some other sorting algorithms, like insertion sort or quicksort, when sorting very small lists.

CONTD..

- **Slower on Some Hardware:** Merge sort's performance can be impacted by the architecture of the computer it runs on. On machines with limited memory, the additional memory allocation and data copying steps during merging can make the process slower.
- **Complex Implementation:** Implementing merge sort can be more complex compared to some other sorting algorithms. It involves recursive calls and careful management of subarrays and temporary storage.



Working of Merge Sort Algorithm

Let the elements of array are -

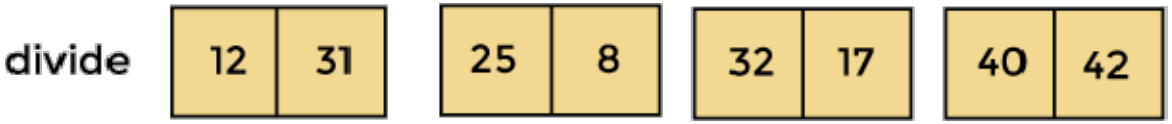
12	31	25	8	32	17	40	42

According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

CONTD..

As there are eight elements in the given array, so it is divided into two arrays of size 4.

Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.

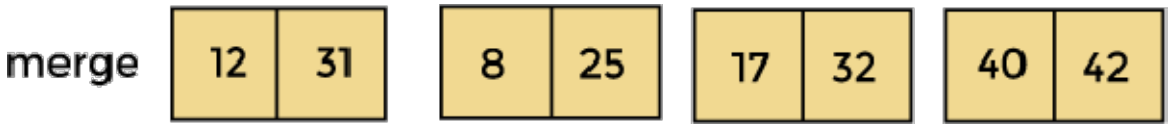


CONTD..

Now, again divide these arrays to get the atomic value that cannot be further divided.

Now, combine them in the same manner they were broken.
In combining, first compare the element of each array and then combine them into another array in sorted order.
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.

CONTD..



In the next iteration of combining, now compare the arrays with two data values and merge them into an array of four values in sorted order.

CONTD..

Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like: