FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

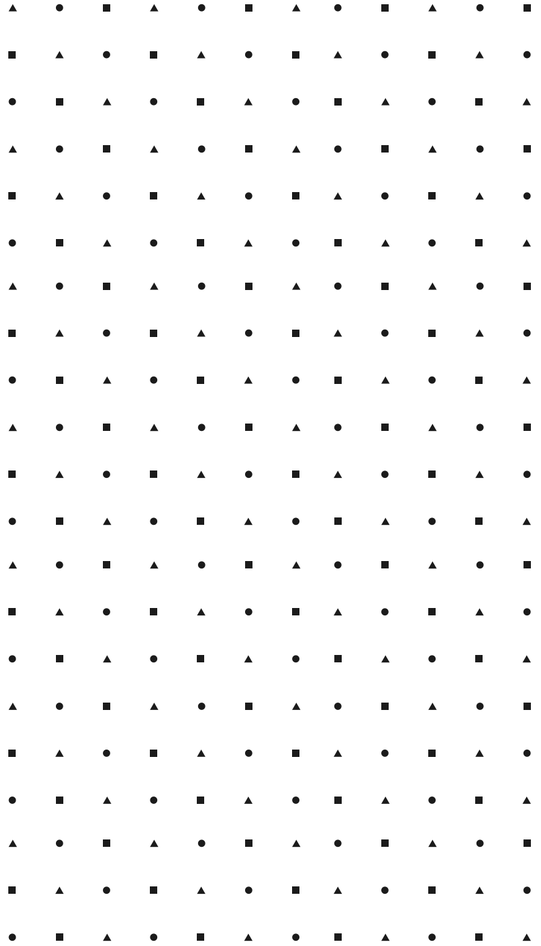
INSERTION AND MERGE SORTS ALGORITHMS COMPLIXITY

PRESENTED BY
- ABDELHAKIM AZZOUZ.
- CHAOUKI NOUAR

SUPERVISED BY:
- TAHAR ZIOUAL

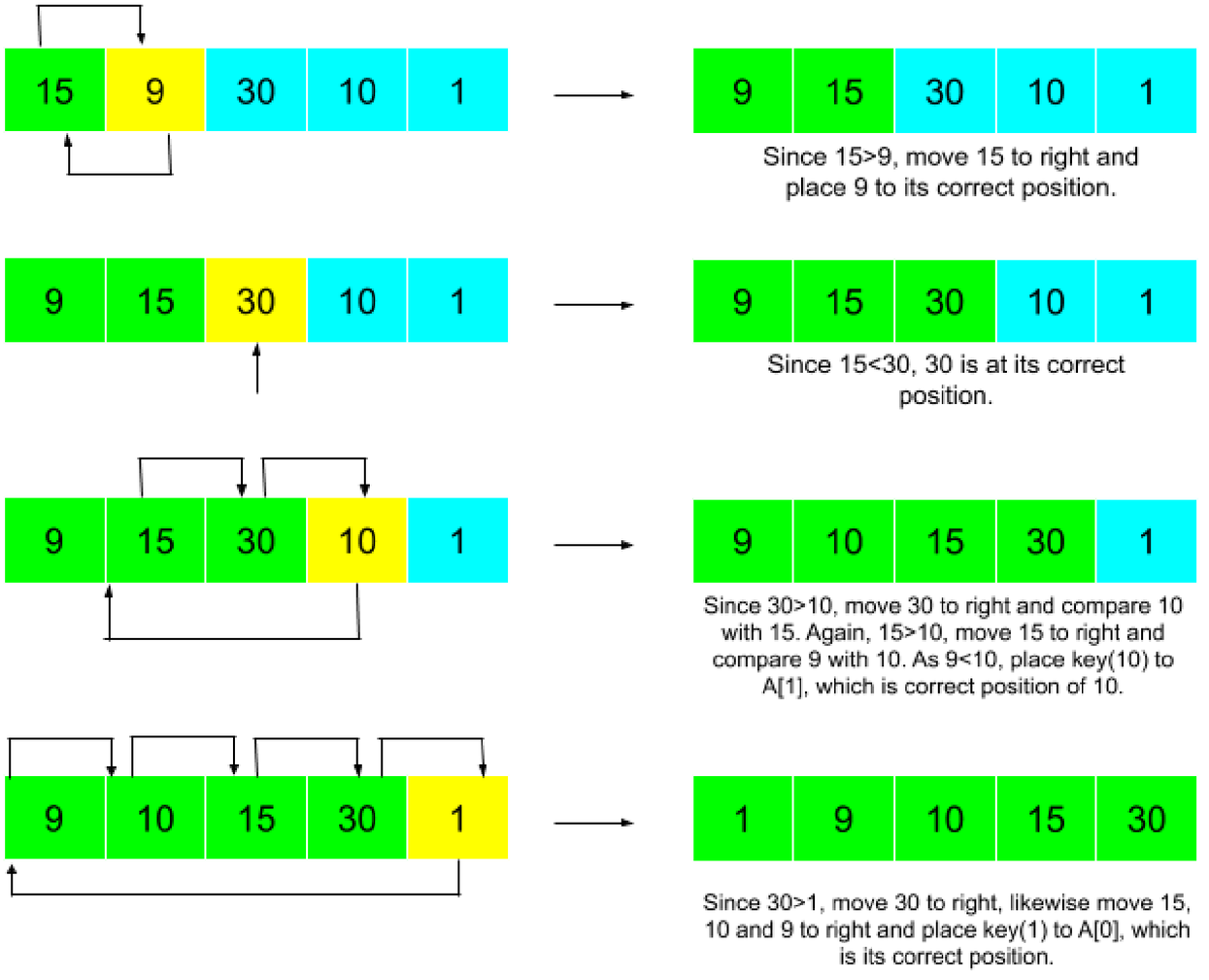
Insertion Sort Algorithm

Insertion sort is the sorting mechanism where the sorted array is built having one item at a time. The array elements are compared with each other sequentially and then arranged simultaneously in some particular order. This sort works on the principle of inserting an element at a particular position, hence the name Insertion Sort.



How Insertion Sort Works?

1. The first step involves the comparison of the element in question with its adjacent element.
2. And if at every comparison reveals that the element in question can be inserted at a particular position, then space is created for it by shifting the other elements one position to the right and inserting the element at the suitable position.
3. The above procedure is repeated until all the elements in the array is at their apt position.


15 9 30 10 1
9 15 30 10 1
Since 15>9, move 15 to right and place 9 to its correct position.
9 15 30 10 1
9 15 30 10 1
Since 15<30, 30 is at its correct position.
9 15 30 10 1
9 10 15 30 1
Since 30>10, move 30 to right and compare 10 with 15. Again, 15>10, move 15 to right and compare 9 with 10. As 9<10, place key(10) to A[1], which is correct position of 10.
9 10 15 30 1
1 9 10 15 30
Since 30>1, move 30 to right, likewise move 15, 10 and 9 to right and place key(1) to A[0], which is its correct position.

```python
x = [2,0,8,3,9,4,5,4,5,5,5,7,4,8,5,2,1]

def f(start):
    next = start+1
    if next == len(x):
        return print("end")
    else:
        if x[start] > x[next] :
            mid = x[start]
            x[start] = x[next]
            x[next]= mid
            if start != 0:
                f(start-1)
            else:
                f(next)
        else:
            f(next)

f(0)
print (x)
```

Complexity

Worst case :

$$T(n) = \sum_{i=0}^{n-1} T(i) + c$$

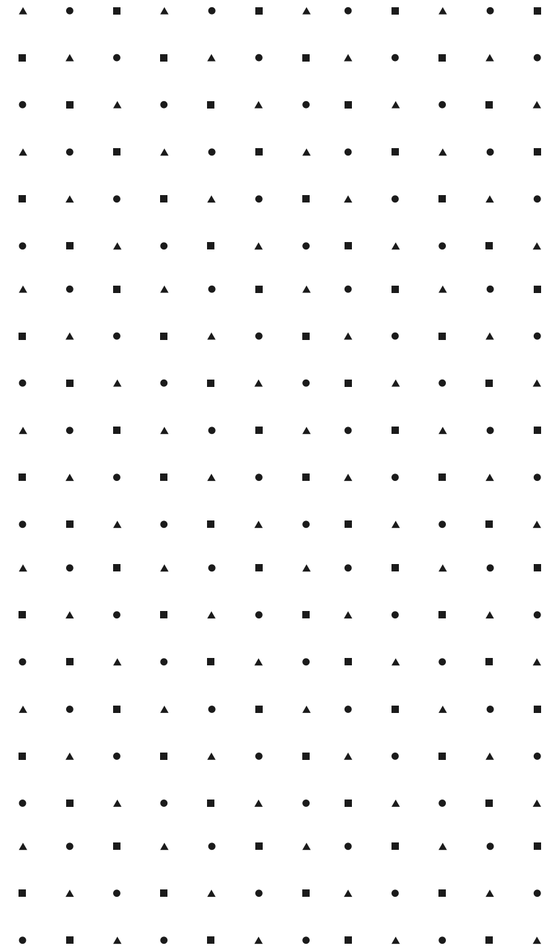
$$T(n) = c \sum_{i=0}^{n} i = c + 2 + \cdots + nc = c\left(\frac{n(n+1)}{2}\right) = \frac{c}{2}((n^2) + n) = O(n^2)$$

Best case :

$T(n) = c + T(n-1) = O(n)$

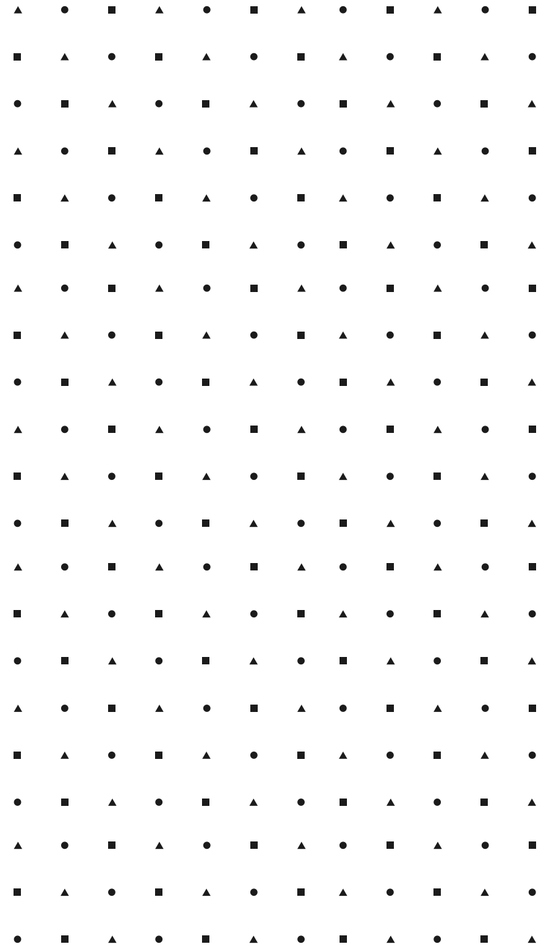
Advantages

1. Simple and easy-to-understand implementation.
2. Efficient for small data
3. Maintains relative order of the input data in case of two equal values (stable)
4. It requires only a constant amount O(1) of additional memory space (in-place Algorithm)



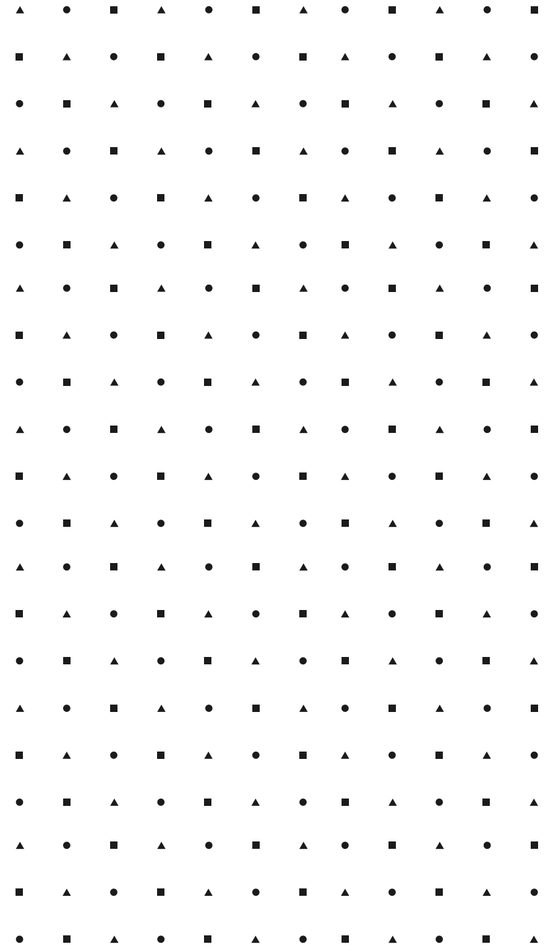
Disadvantages

1. The insertion sort is particularly useful only when sorting a list of few items.
2. Goes through the whole process even if the list is sorted



Merge Sort Algorithm

Merge sort is an efficient, general-purpose, comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the order of equal elements is the same in the input and output. Merge sort is a divide and conquer algorithm that was invented by John von Neumann in 1945.

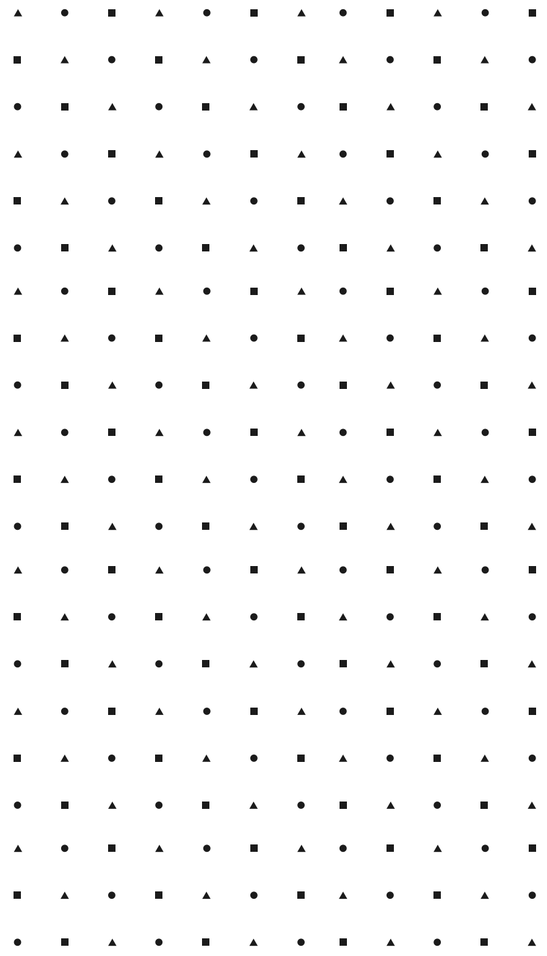


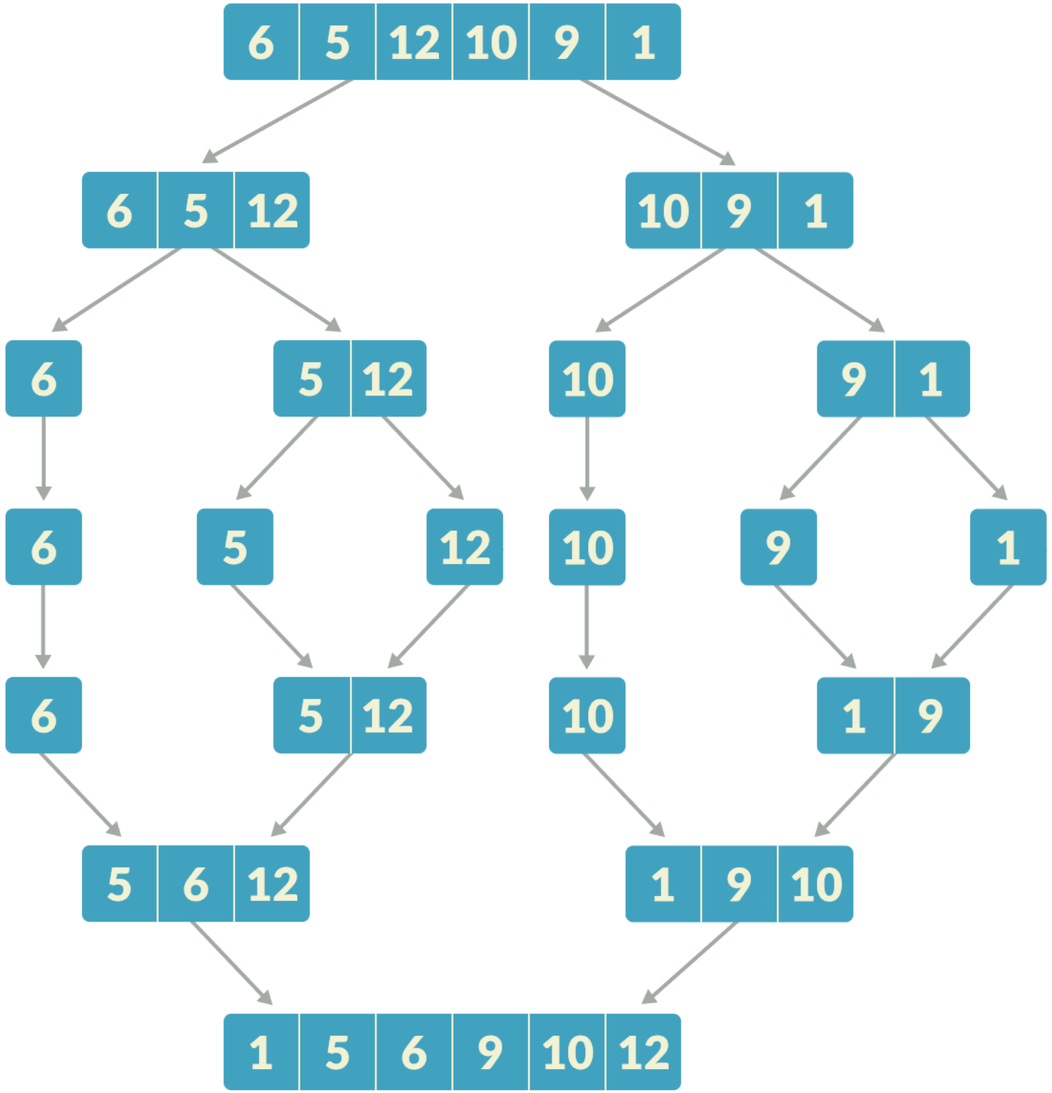
How Merge Sort Works?

MergeSort(arr[], left, right)

If right > left:

1. **Find the middle point to divide the array into two halves**
2. **Call mergeSort for the first half**
3. **Call mergeSort for the second half**
4. **Merge the two halves sorted in step 2 and 3**

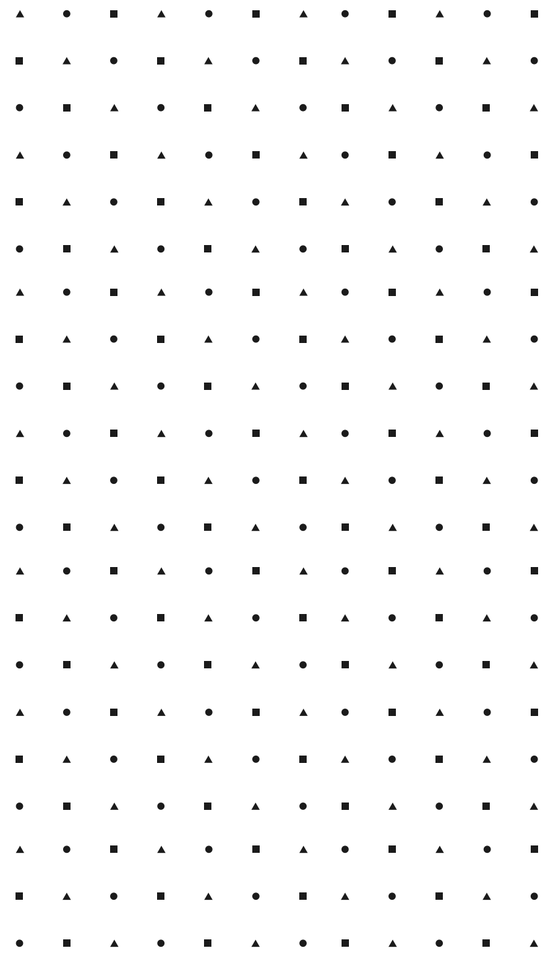



6 5 12 10 9 1
6 5 12
10 9 1
6
5 12
10
9 1
6
5
12
10
9
1
6
5 12
10
1 9
5 6 12
1 9 10
1 5 6 9 10 12

```python
def merge_sort(x):
    if len(x) <= 1:
        return x
    mid = len(x) // 2
    left, right = merge_sort(x[:mid]), merge_sort(x[mid:])
    return merge(left, right, x.copy())


def merge(left, right, merged):

    left_cursor, right_cursor = 0, 0
    while left_cursor < len(left) and right_cursor < len(right):

        # Sort each one and place into the result
        if left[left_cursor] <= right[right_cursor]:
            merged[left_cursor+right_cursor]=left[left_cursor]
            left_cursor += 1
        else:
            merged[left_cursor + right_cursor] = right[right_cursor]
            right_cursor += 1

    for left_cursor in range(left_cursor, len(left)):
        merged[left_cursor + right_cursor] = left[left_cursor]

    for right_cursor in range(right_cursor, len(right)):
        merged[left_cursor + right_cursor] = right[right_cursor]

    return merged

x = [2,0,8,3,9,4,5,4,5,5,5,7,4,8,5,2,1]

print(merge_sort(x))
```

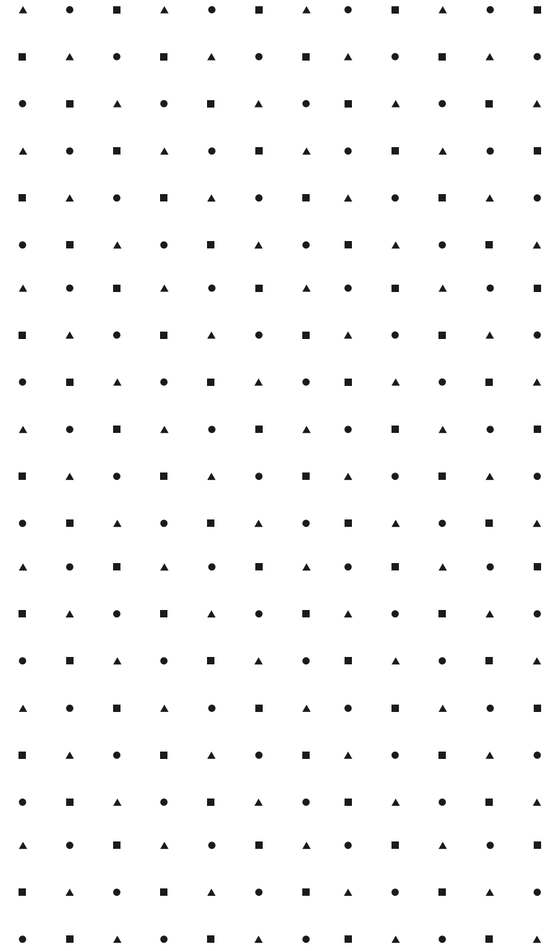
Complexity

T(n) = 2T(n/2) + n
We guess the solution as
T(n) = O(nLogn)
we need to prove that
T(n) <= cnLogn.
We can assume that it is true
for values smaller than n.
T(n) = 2T(n/2) + n
<= 2cn/2Log(n/2) + n = cnLogn - cnLog2 + n
= cnLogn - cn + n <= cnLogn
when cn+n <=0
So: T(n) = O(nLogn)



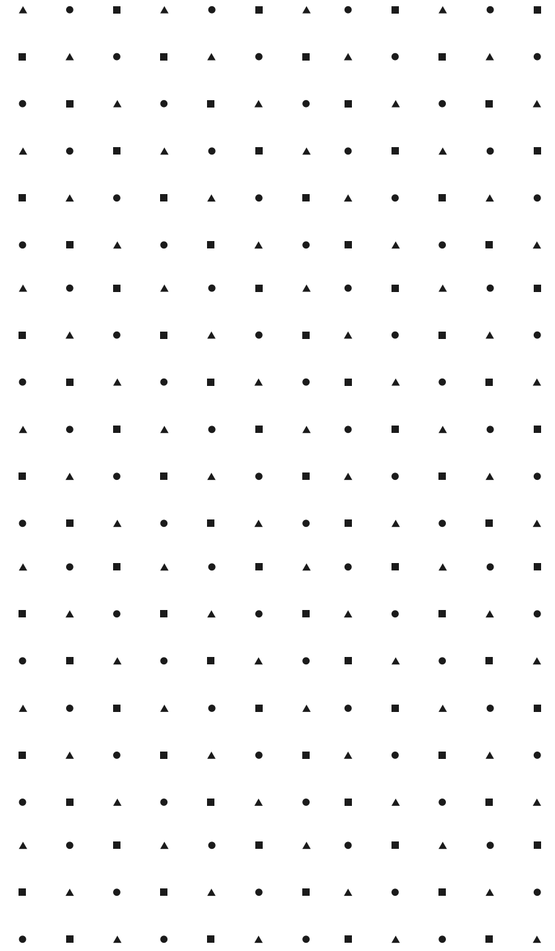
Advantages

1. It can be applied to files of any size.
2. If heap sort is used for the in-memory part of the merge, its operation can be overlapped with I/O



Disadvantages

1. Requires extra space »N
2. Merge Sort requires more space than other sorts.
3. Goes through the whole process even if the list is sorted
4. Slower comparative to the other sort algorithms for smaller tasks.



Merge Sort vs. Insertion Sort

Parameters	Merge Sort	Insertion Sort
Worst Case Complexity	O(N*log N)	$O(N^2)$
Average Case Complexity	O(N*log N)	$O(N^2)$
Best Case Complexity	O(N*log N)	O(N)
Auxiliary Space Complexity	O(N)	O(1)
Works well on	On huge dataset.	On small dataset.
Efficiency	Comparitively Efficient.	Comparitively Inefficient.
Inplace Sorting	No	Yes
Algorithm Paradigm	Divide and Conquer	Incremental Approach