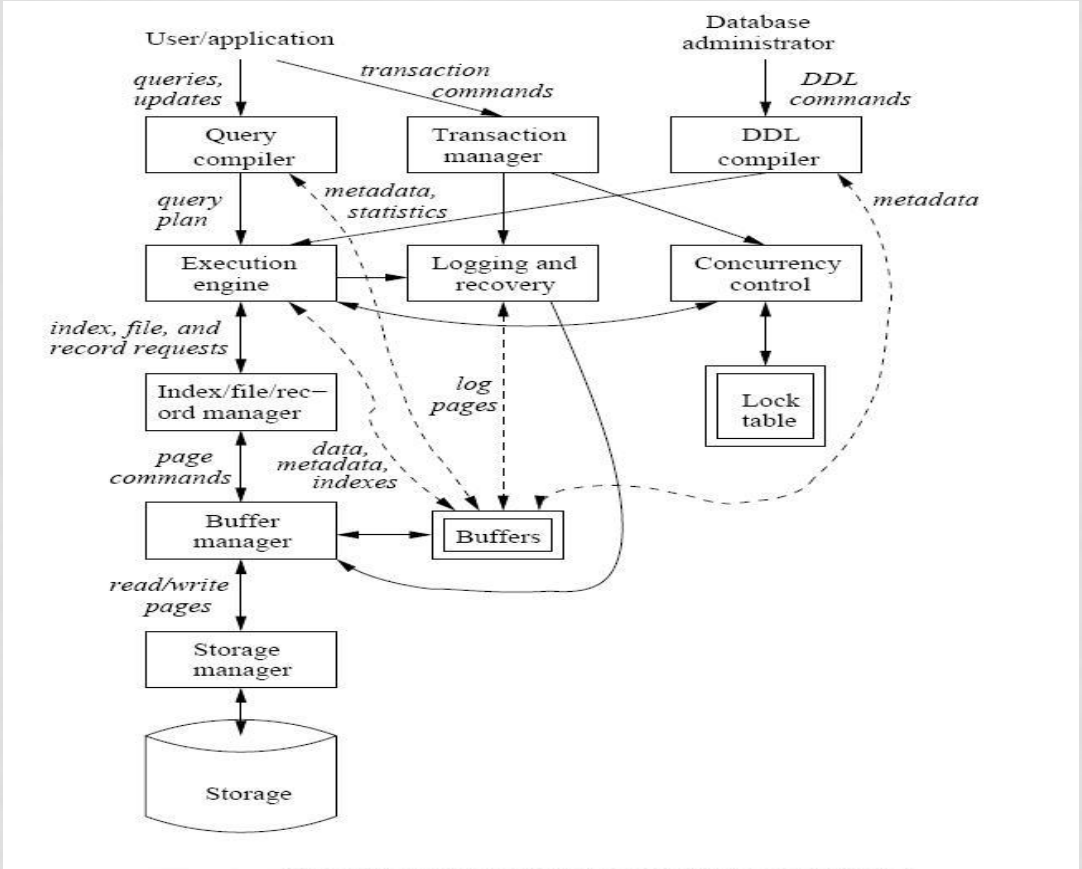
# PARALLEL & DISTRIBUTED DATABASES

## INTRODUCTION

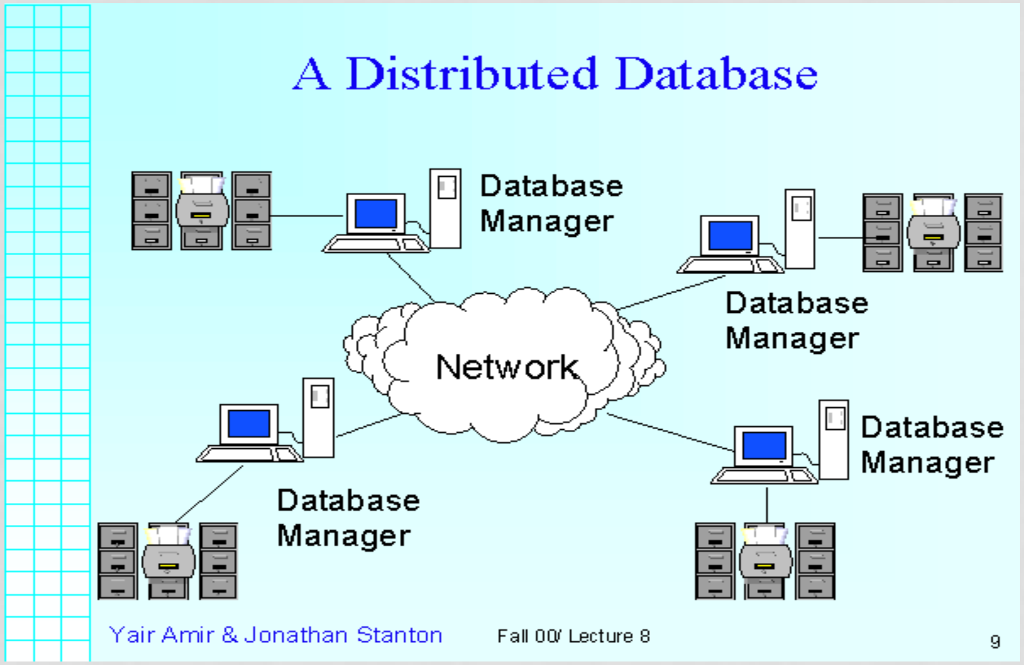
- **In centralized database:**
  - Data is located in one place (one server)
  - All DBMS functionalities are done by that server
    - Enforcing ACID properties of transactions
    - Concurrency control, recovery mechanisms
    - Answering queries



Database management system components

# INTRODUCTION

- **In Distributed databases:**
  - Data is stored in multiple places (each is running a DBMS)
  - New notion of distributed transactions
  - DBMS functionalities are now distributed over many machines
    - Revisit how these functionalities work in distributed environment



# WHY DISTRIBUTED DATABASES

- Data is too large (***Volume*** in Big Data)

- Applications are by nature distributed
  - Bank with many branches
  - Chain of retail stores with many locations
  - Library with many branches

- Get benefit of distributed and parallel processing
  - Faster response time for queries

## PARALLEL VS. DISTRIBUTED

- Distributed processing usually imply parallel processing (not vise versa)
  - Can have parallel processing on a single machine

- **Assumptions about architecture**
  - **Parallel Databases**
    - Machines are physically close to each other, e.g., same server room
    - Machines connects with dedicated high-speed LANs and switches
    - Communication cost is assumed to be small
    - Can shared-memory, shared-disk, or shared-nothing architecture
  - **Distributed Databases**
    - Machines can far from each other, e.g., in different continent
    - Can be connected using public-purpose network, e.g., Internet
    - Communication cost and problems cannot be ignored
    - Usually shared-nothing architecture

# OVERVIEW ON TRANSACTIONS

```
BEGIN TRANSACTION;

INSERT INTO Takes
    SELECT Students.SSN, Courses.CID
    FROM Students, Courses
    WHERE Students.name = 'Mary Johnson' and
                   Courses.name = 'CSE444'

-- More updates here....

IF everything-went-OK
      THEN COMMIT;
ELSE ROLLBACK
```

If system crashes, the transaction is still either committed or aborted

## OVERVIEW ON TRANSACTIONS

- A *transaction* = sequence of statements that either all succeed, or all fail
- Basic unit of processing
- **Transactions have the ACID properties:**

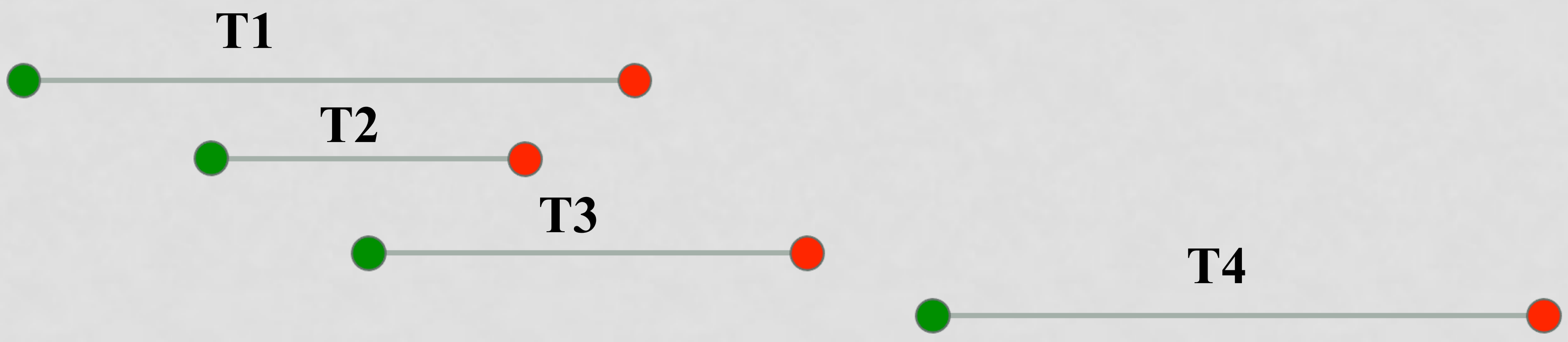
  A = atomicity

  C = consistency

  I = independence (Isolation)

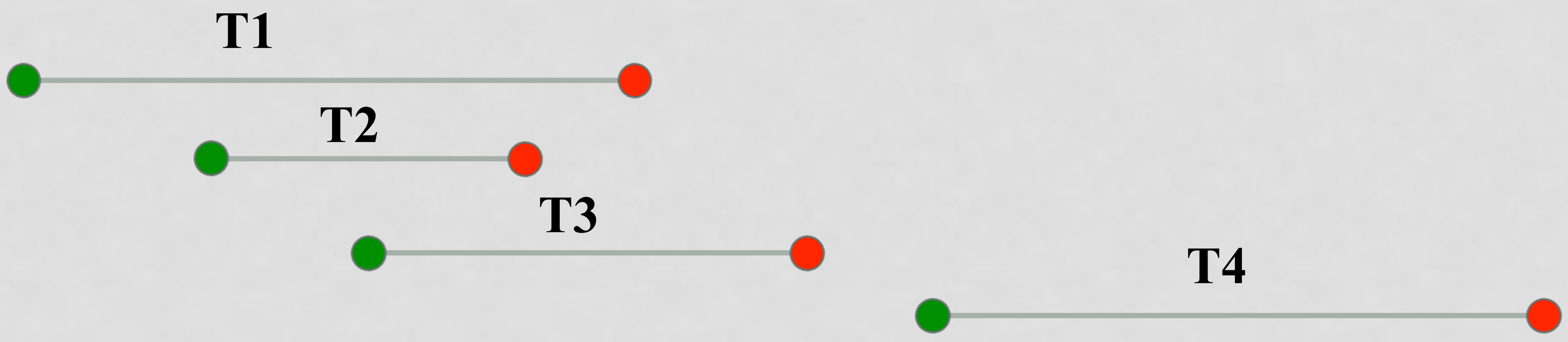
  D = durability

## TRANSACTION ACID PROPERTIES



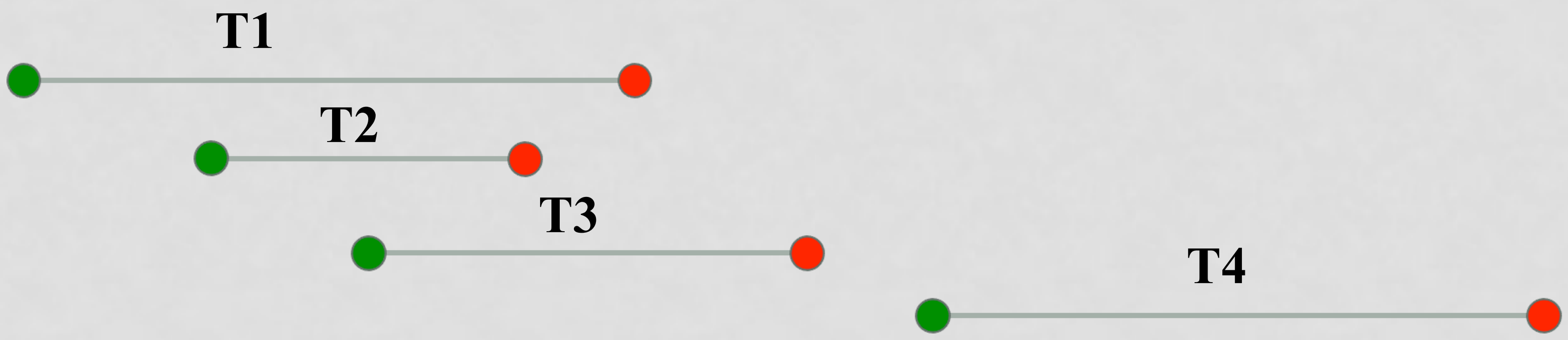
- Each transaction has a Start 🟢 and End 🔴 and does many things in between
- **“A” ➔ Atomic**: Either the entire transaction is done (all its actions) or none.
- **“C” ➔ Consistency:** A transaction must move the DB from one consistent state to another consistent state

# TRANSACTION ACID PROPERTIES (CONT'D)



- **What about interaction**
  - Can T2 read what T1 is writing?
  - Can T3 read what T1 is reading?
  - Can T4 read what T1 wrote?
- **"I" ➔ Isolation:** Although running concurrently, they should appear ***as if*** they run is a certain serial order

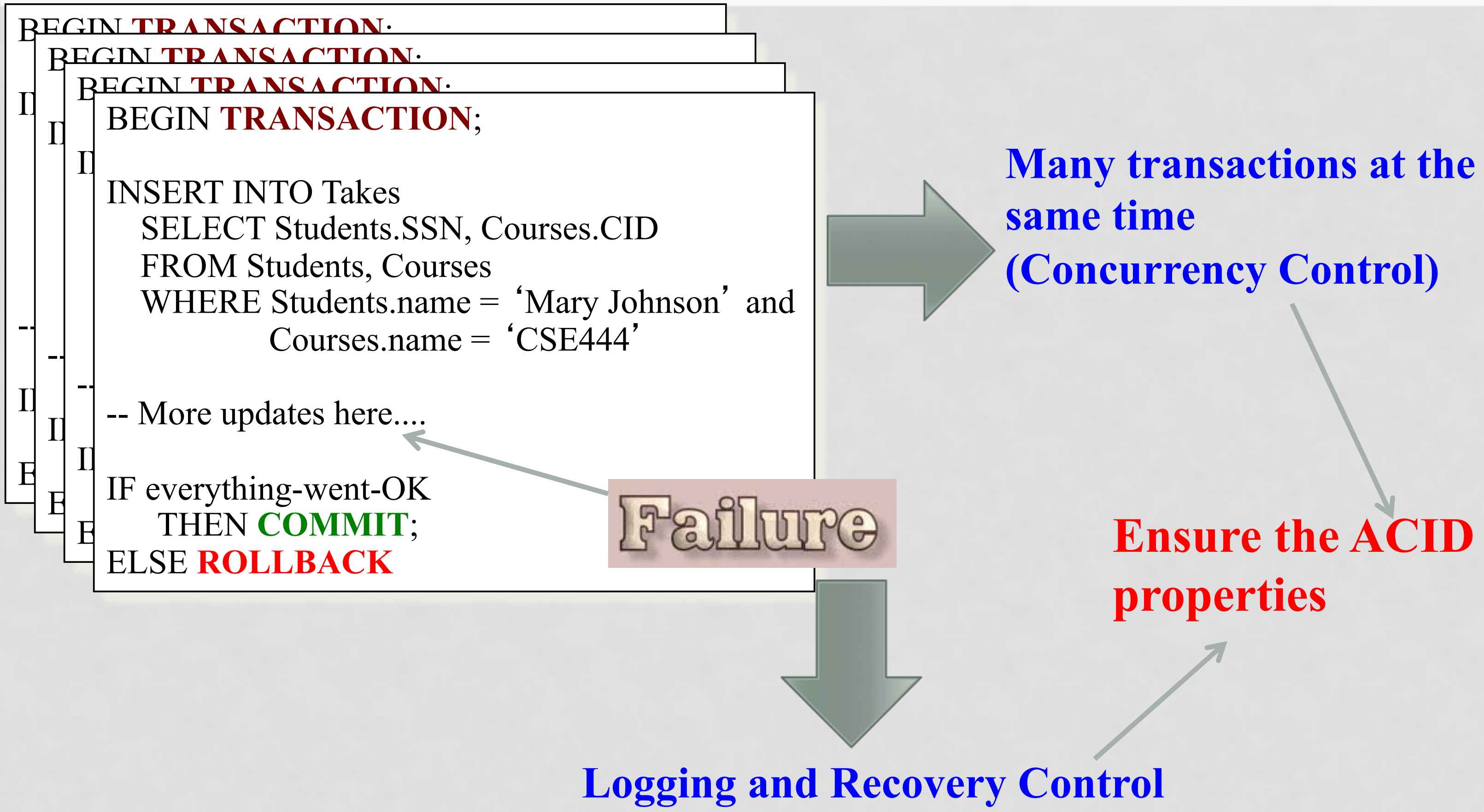
# TRANSACTION ACID PROPERTIES (CONT'D)



- If T1 failed & T2 completed ➔ This means what?
- T1 🡐 ***Rolledback*** & T2 🡐 ***Committed***

- **"D" ➔ Durability:** The effect of a committed transaction must be persistent (not lost)

# OVERVIEW ON TRANSACTIONS



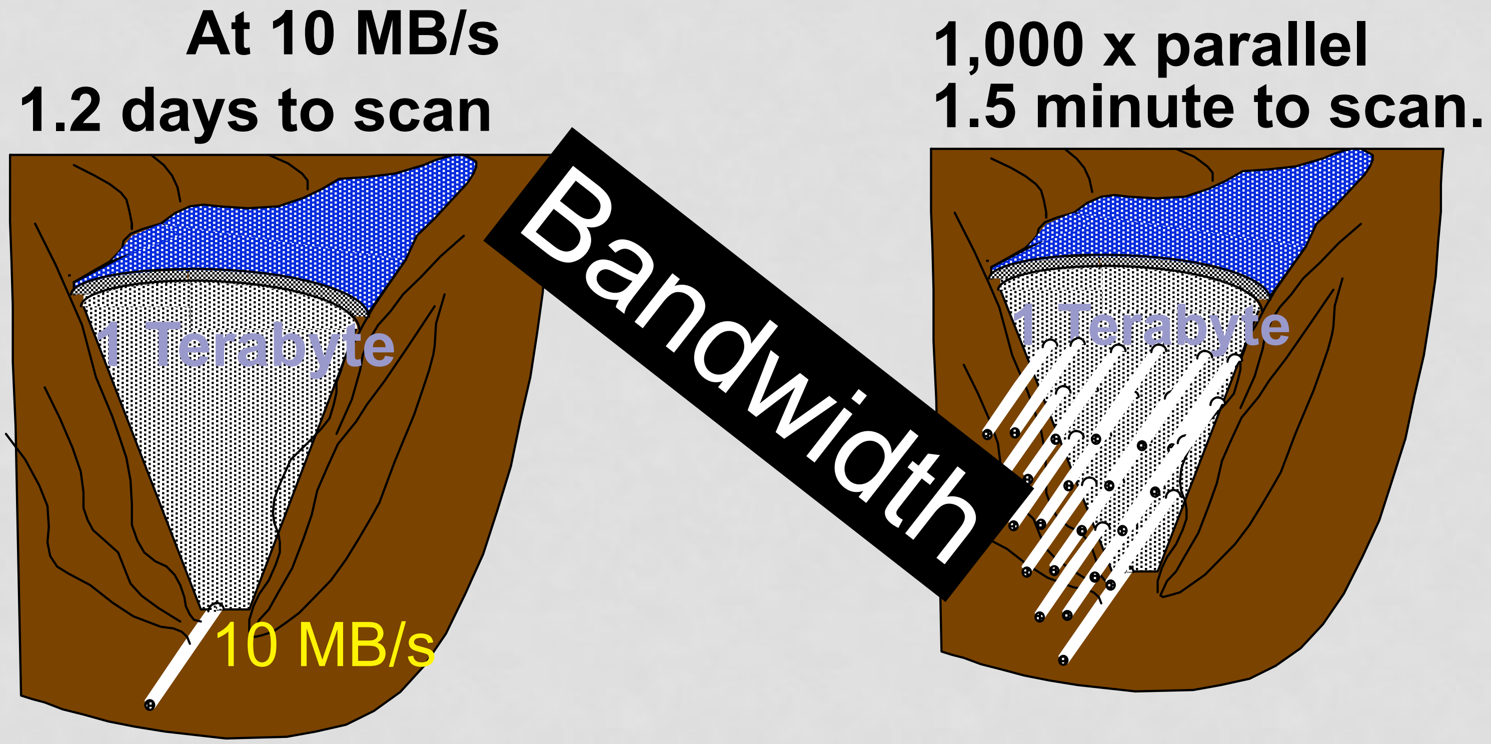
```
BEGIN TRANSACTION;

INSERT INTO Takes
    SELECT Students.SSN, Courses.CID
    FROM Students, Courses
    WHERE Students.name = 'Mary Johnson' and
                Courses.name = 'CSE444'

-- More updates here....

IF everything-went-OK
     THEN COMMIT;
ELSE ROLLBACK
```

Failure
Many transactions at the same time (Concurrency Control)
Ensure the ACID properties
Logging and Recovery Control

Fundamental differences on how to do it in centralized vs. distributed DBs.

## PARALLEL DATABASE & PARALLEL PROCESSING

## WHY PARALLEL PROCESSING

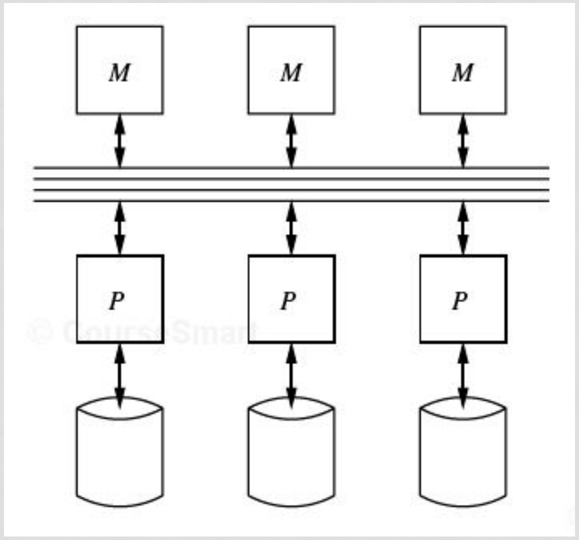


- Divide a big problem into many smaller ones to be solved in parallel
- Increase bandwidth (in our case decrease queries' response time)

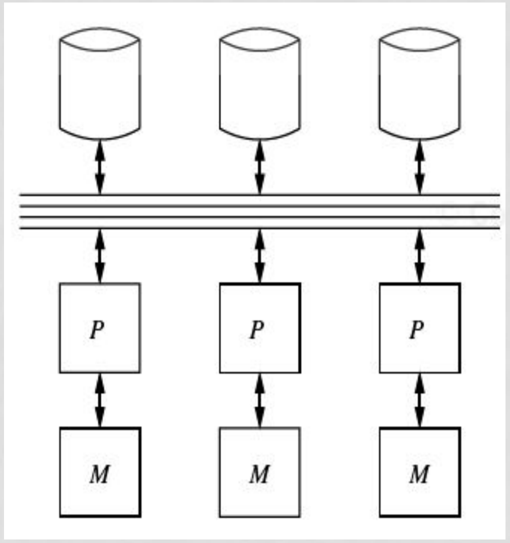
# DIFFERENT ARCHITECTURE

- Three possible architectures for passing information

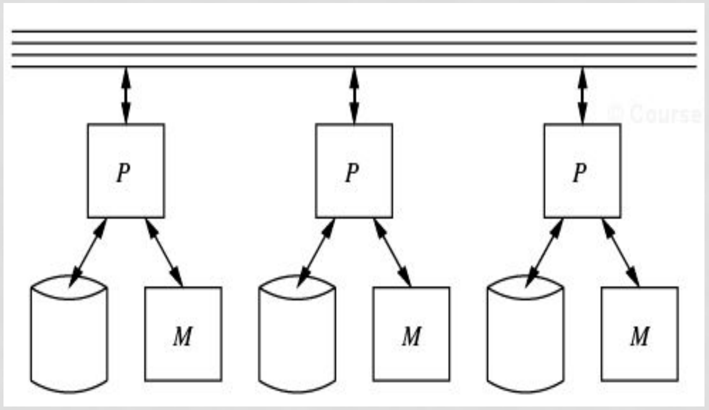
**Shared-memory**



**Shared-disk**

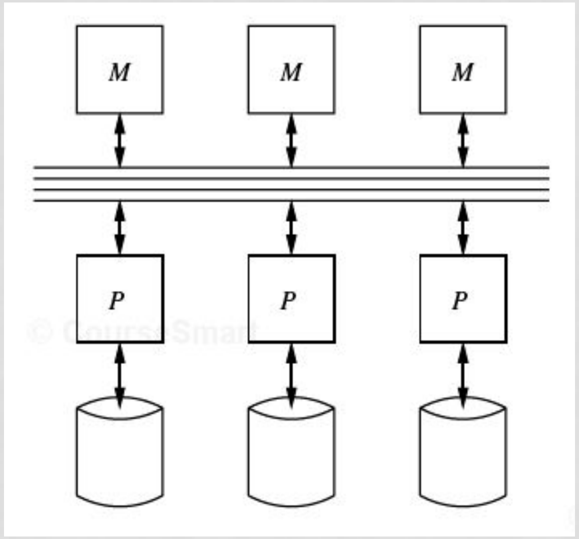


**Shared-nothing**



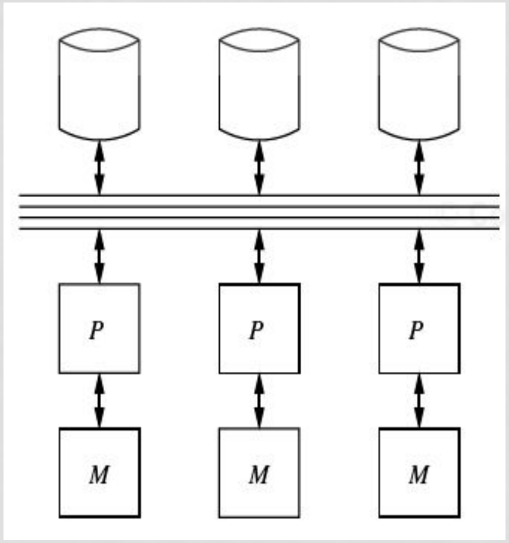
## 1- SHARED-MEMORY ARCHITECTURE

- Every processor has its own disk
- Single memory address-space for all processors
  - Reading or writing to far memory can be slightly more expensive
- Every processor can have its own local memory and cache as well



## 2- SHARED-DISK ARCHITECTURE

- Every processor has its own memory (not accessible by others)
- All machines can access all disks in the system
- Number of disks does not necessarily match the number of processors

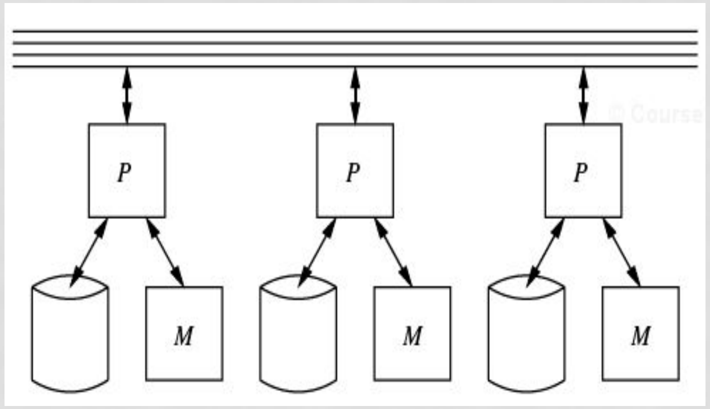


# 3- SHARED-NOTHING ARCHITECTURE

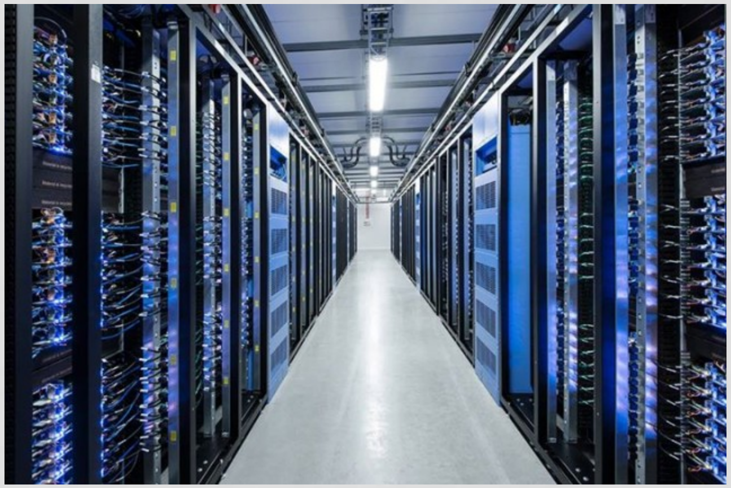
- Every machine has its own memory and disk
  - Many cheap machines (commodity hardware)

- Communication is done through high-speed network and switches

- **Usually machines can have a hierarchy**
  - Machines on same rack
  - Then racks are connected through high-speed switches

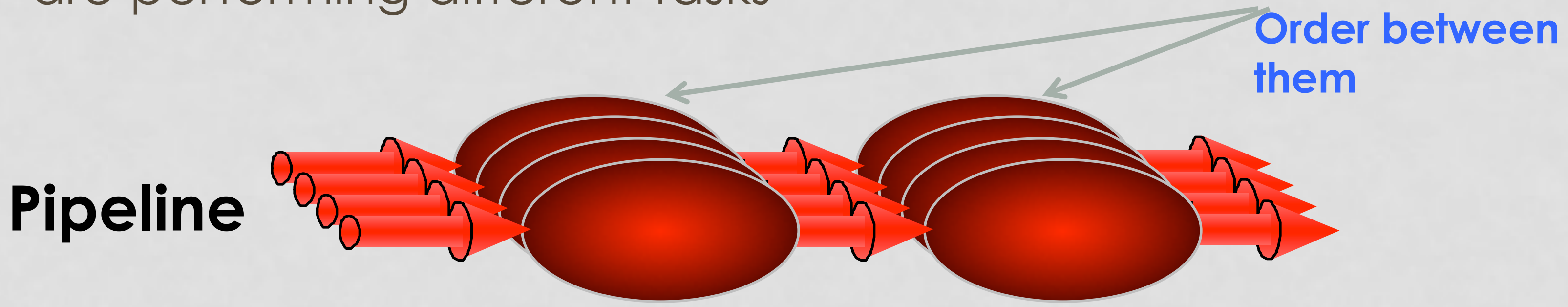


- **Scales better**
- **Easier to build**
- **Cheaper cost**

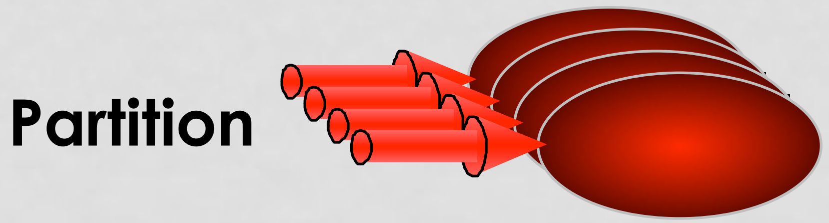


## TYPES OF PARALLELISM

- **Pipeline Parallelism (Inter-operator parallelism)**
  - Ordered (or partially ordered) tasks and different machines are performing different tasks

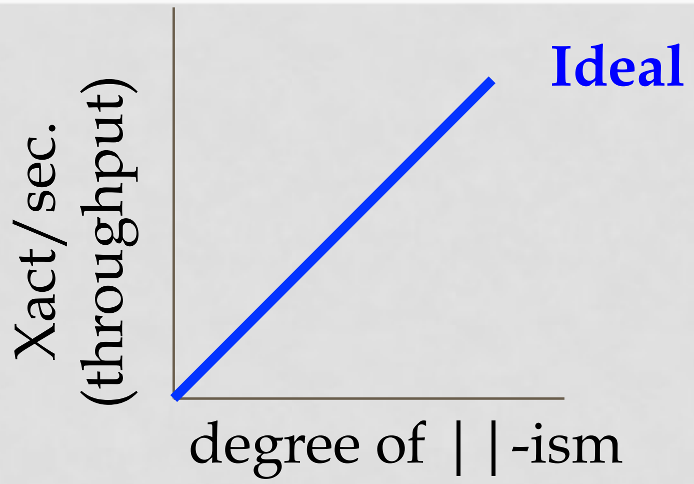


- **Partitioned Parallelism (Intra-operator parallelism)**
  - A task divided over all machines to run in parallel

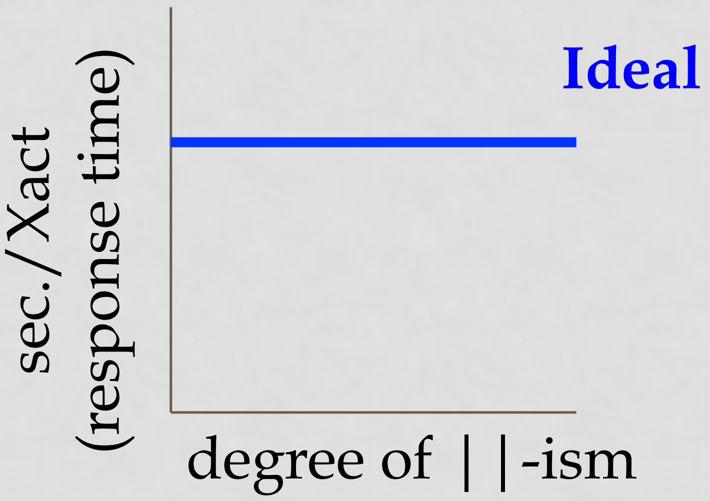


# IDEAL SCALABILITY SCENARIO

- Speed-Up
  - More resources means proportionally less time for given amount of data.

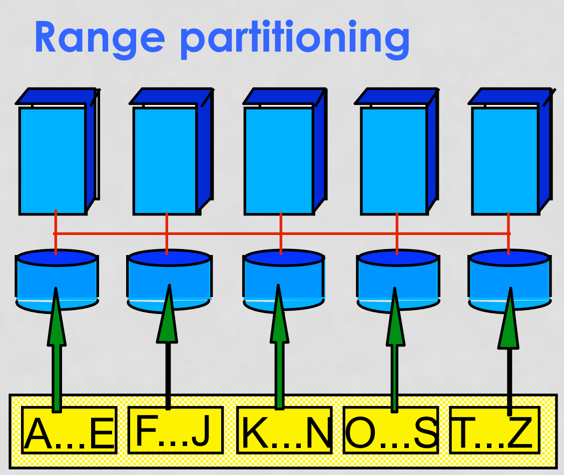


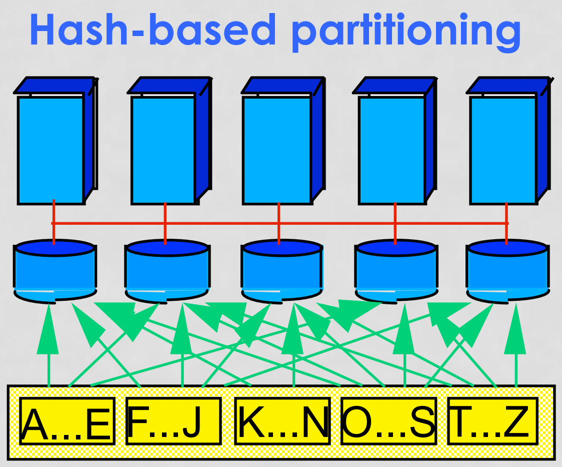
- Scale-Up
  - If resources increased in proportion to increase in data size, time is constant.

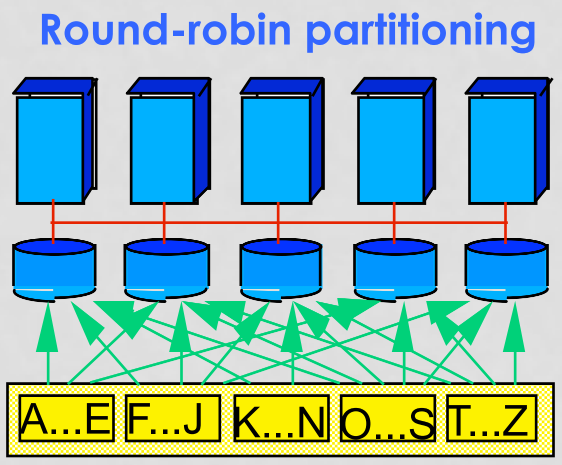


# PARTITIONING OF DATA

**To partition a relation R over m machines**







- Shared-nothing architecture is sensitive to partitioning
- Good partitioning depends on what operations are common

## PARALLEL ALGORITHMS FOR DBMS OPERATIONS

## PARALLEL SCAN $\sigma_c(R)$

- Relation R is partitioned over ***m*** machines
  - Each partition of R is around |R|/m tuples

- Each machine scans its own partition and applies the selection condition c

- **If data are partitioned using round robin or a hash function (over the entire tuple)**
  - The resulted relation is expected to be well distributed over all nodes
  - All partitioned will be scanned

- **If data are range partitioned or hash-based partitioned (on the selection column)**
  - The resulted relation can be clustered on few nodes
  - Few partitions need to be touched

- Parallel Projection is also straightforward
- All partitions will be touched
- Not sensitive to how data is partitioned

## PARALLEL DUPLICATE ELIMINATION

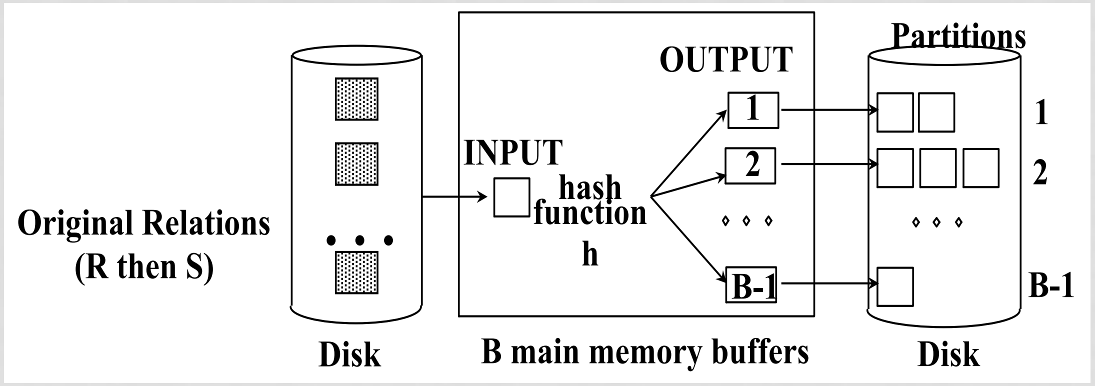
- **If relation is range or hash-based partitioned**
  - Identical tuples are in the same partition
  - So, eliminate duplicates in each partition independently

- **If relation is round-robin partitioned**
  - Re-partition the relation using a hash function
  - So every machine creates m partitions and send the ith partition to machine i
  - machine i can now perform the duplicate elimination

- Same idea applies to Set Operations (Union, Intersect, Except)
- But apply the same partitioning to both relations R & S

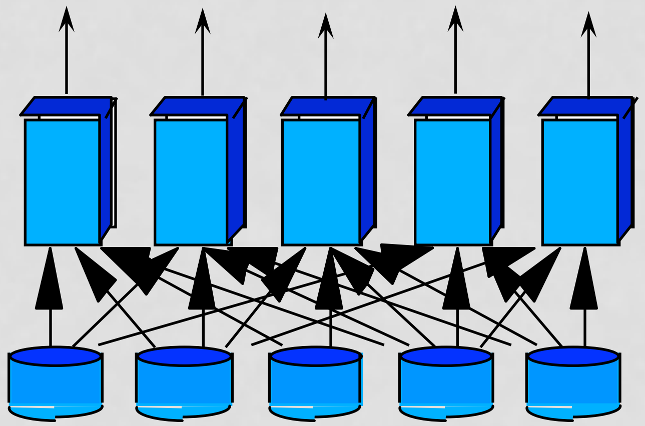
# PARALLEL JOIN R(X,Y) ⋈ S(Y,Z)

- **Re-partition R and S on the join attribute Y (natural join) or (equi join)**
  - Hash-based or range-based partitioning
- **Each machine i receives all ith partitions from all machines (from R and S)**
  - Each machine can locally join the partitions it has
- Depending on the partitions sizes of R and S, local joins can be hash-based or merge-join



## PARALLEL SORTING

- **Range-based**
  - Re-partition R based on ranges into m partitions
  - Machine i receives all $i^{th}$ partitions from all machines and sort that partition
  - The entire R is now sorted
  - **Skewed data is an issue**
    - Apply sampling phase first
    - Ranges can be of different width



- **Merge-based**
  - Each node sorts its own data
  - All nodes start sending their sorted data (one block at a time) to a single machine
  - This machine applies merge-sort technique as data come

# PARALLEL GROUP BY

- Step 1: server i partitions chunk $R_i$ using a hash function h(t.A) mod P: $R_{i0}$, $R_{i1}$, …, $R_{i,P-1}$
- Step 2: server i sends partition $R_{ij}$ to serve j
- Step 3: server j computes $\gamma_{A,\ sum(B)}$ on $R_{0j}$, $R_{1j}$, …, $R_{P-1,j}$

## WHAT ABOUT PARALLEL *MERGE-SORT*?

## ONE-PROCESSOR MERGE-SORT

- **Sorting takes an unordered collection and makes it an ordered one.**

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 77 | 42 | 35 | 12 | 101 | 5 |

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 5 | 12 | 35 | 42 | 77 | 101 |

# DIVIDE AND CONQUER

- Divide and Conquer cuts the problem in half each time, but uses the result of both halves:
  - cut the problem in half until the problem is trivial
  - solve for both halves
  - combine the solutions

# MERGE-SORT

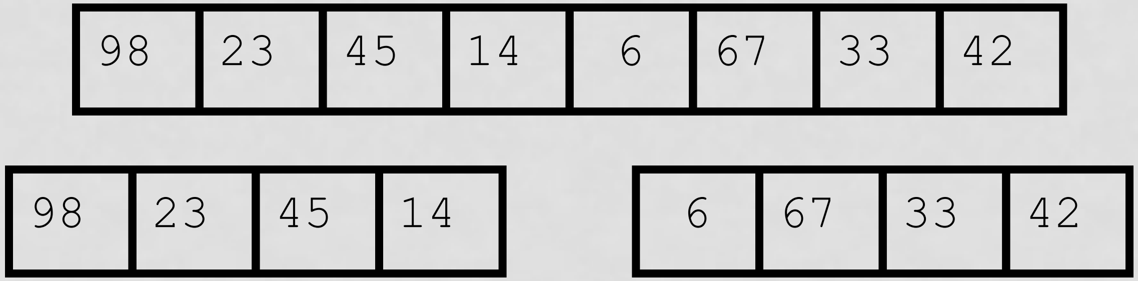
- A divide-and-conquer algorithm
- Divide the unsorted array into 2 halves until the sub-arrays only contain one element
- Merge the sub-problem solutions together:
  - Compare the sub-array's first elements
  - Remove the smallest element and put it into the result array
  - Continue the process until all elements have been put into the result array

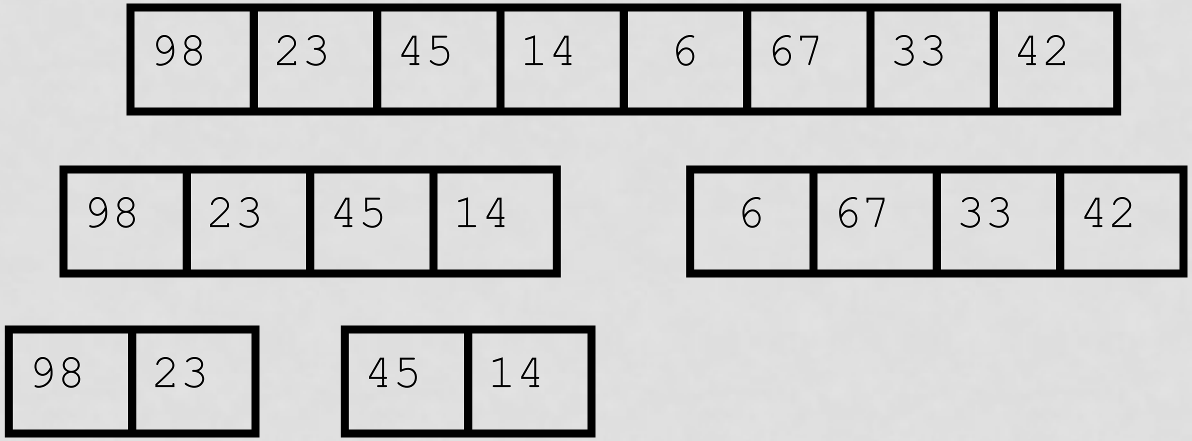
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 37 | 23 | 6 | 89 | 15 | 12 | 2 | 19 |

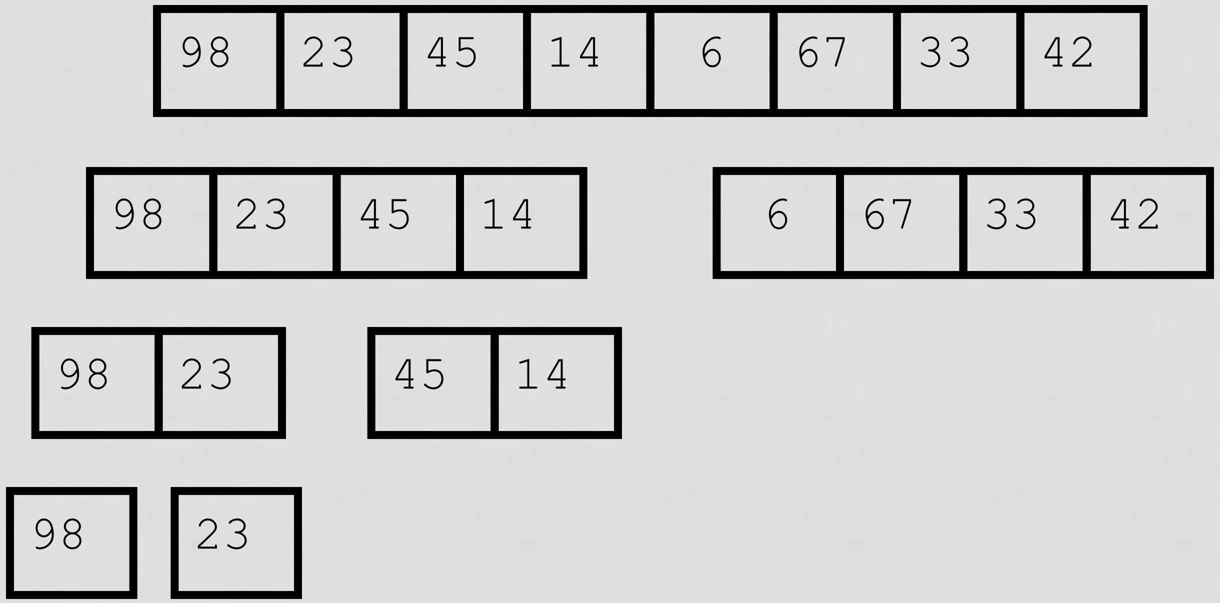
## ALGORITHM

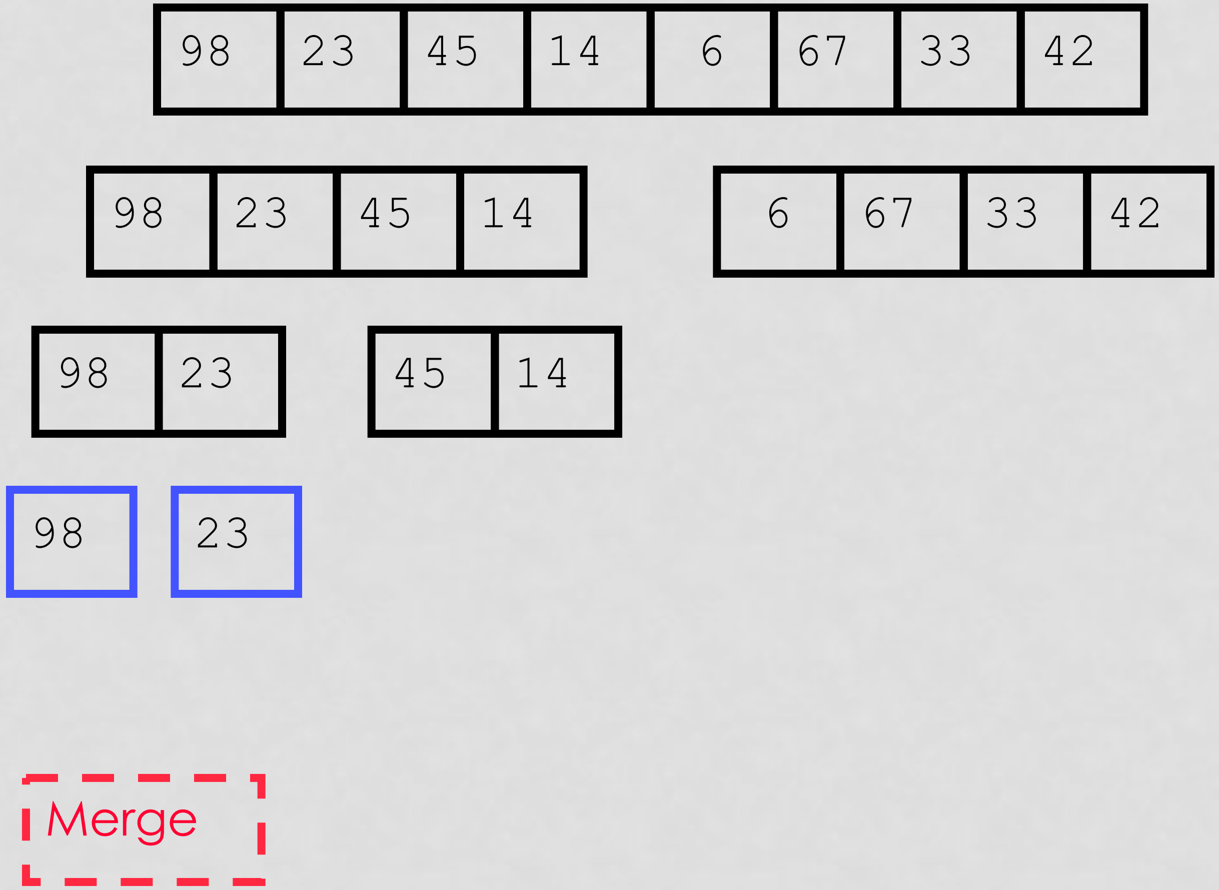
```
Mergesort(Passed an array)
  if array size > 1
    Divide array in half
    Call Mergesort on first half.
    Call Mergesort on second half.
    Merge two halves.

Merge(Passed two arrays)
  Compare leading element in each array
  Select lower and place in new array.
    (If one input array is empty then place
     remainder of other array in output array)
```

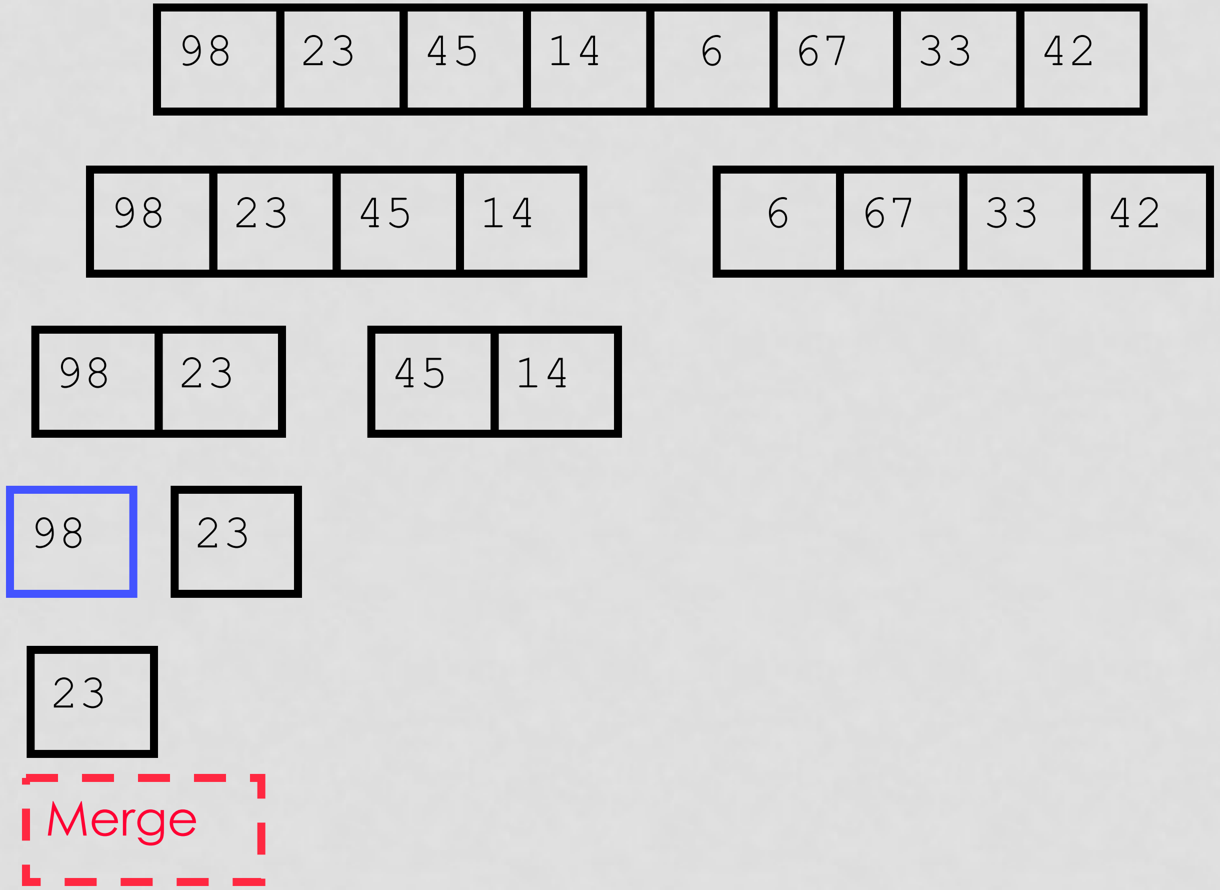
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 98 | 23 | 45 | 14 | 6 | 67 | 33 | 42 |

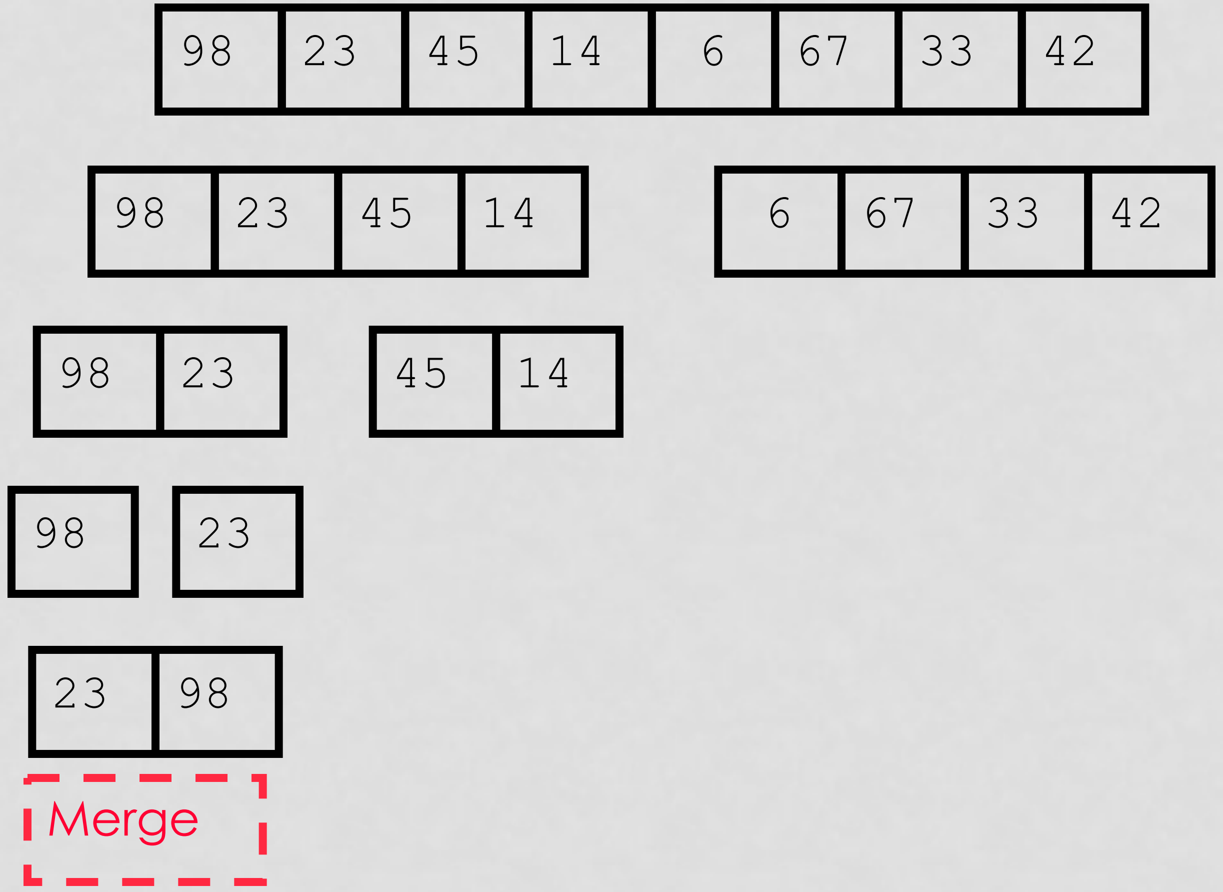

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42

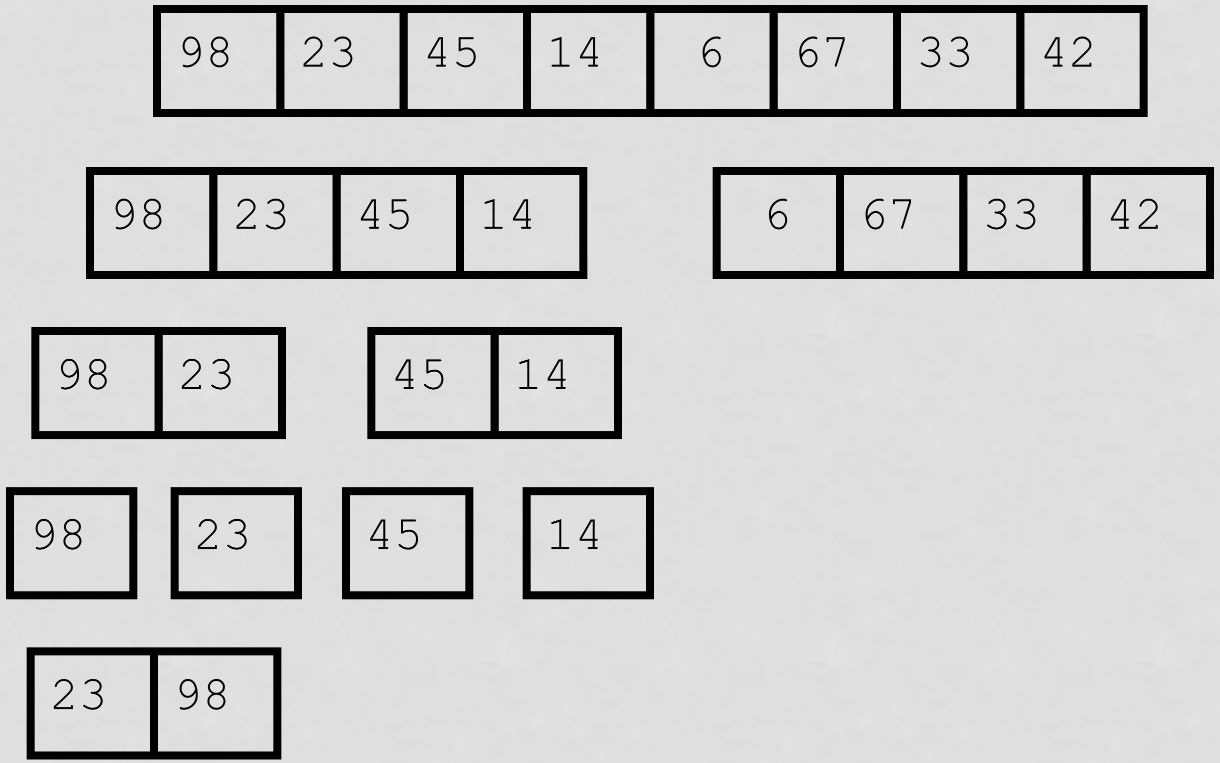

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14

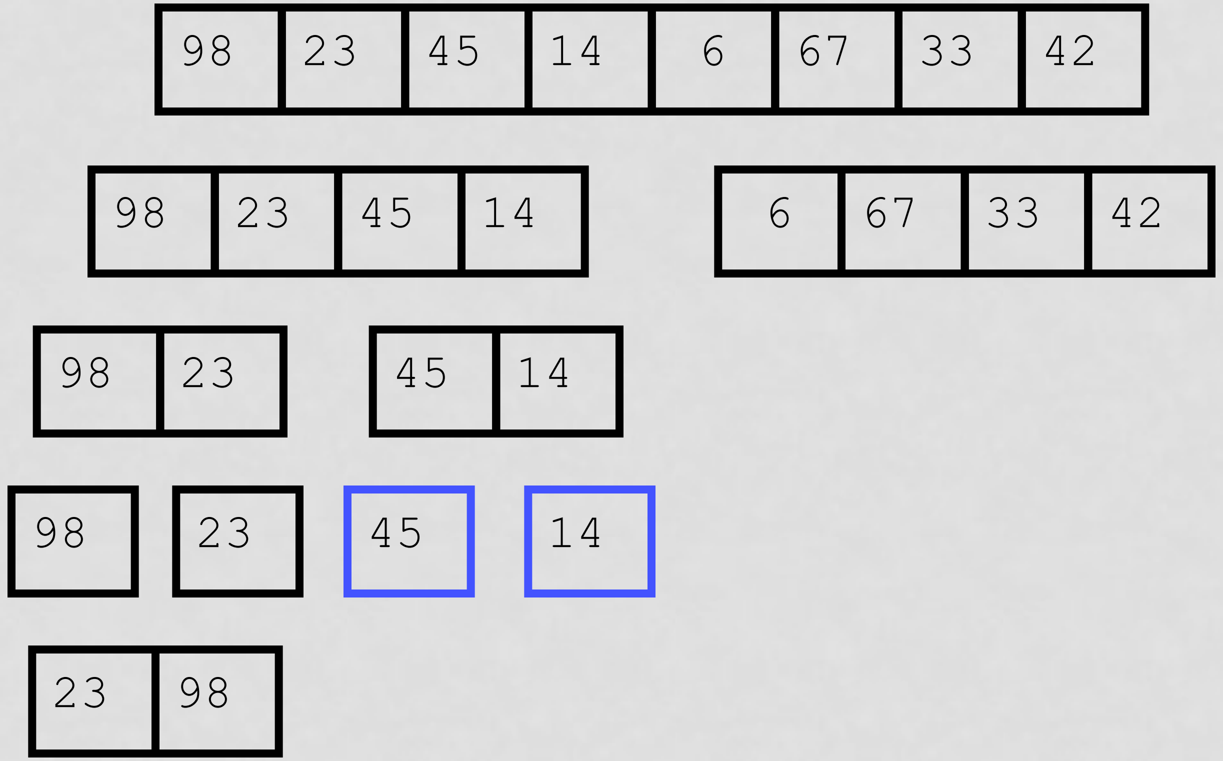

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23


98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
Merge

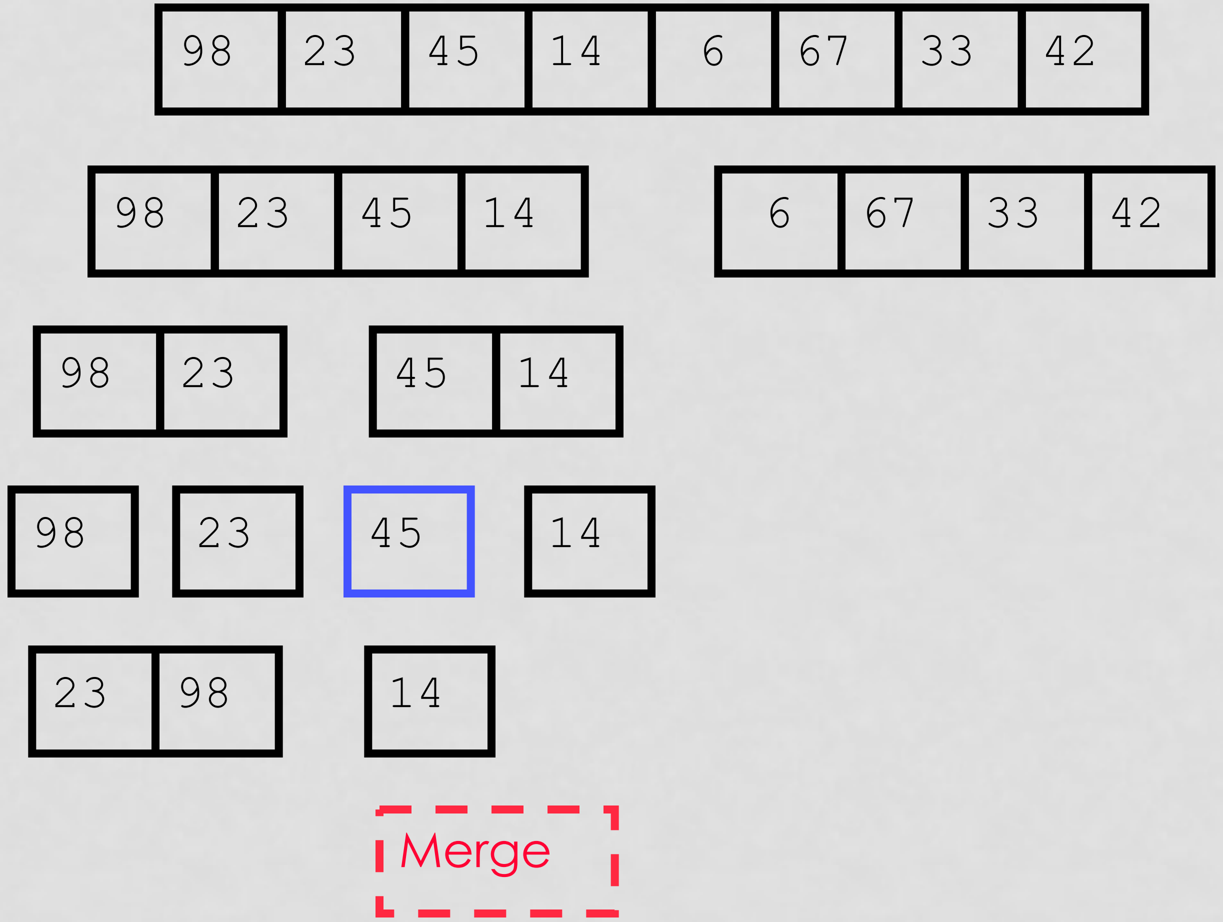

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23
Merge

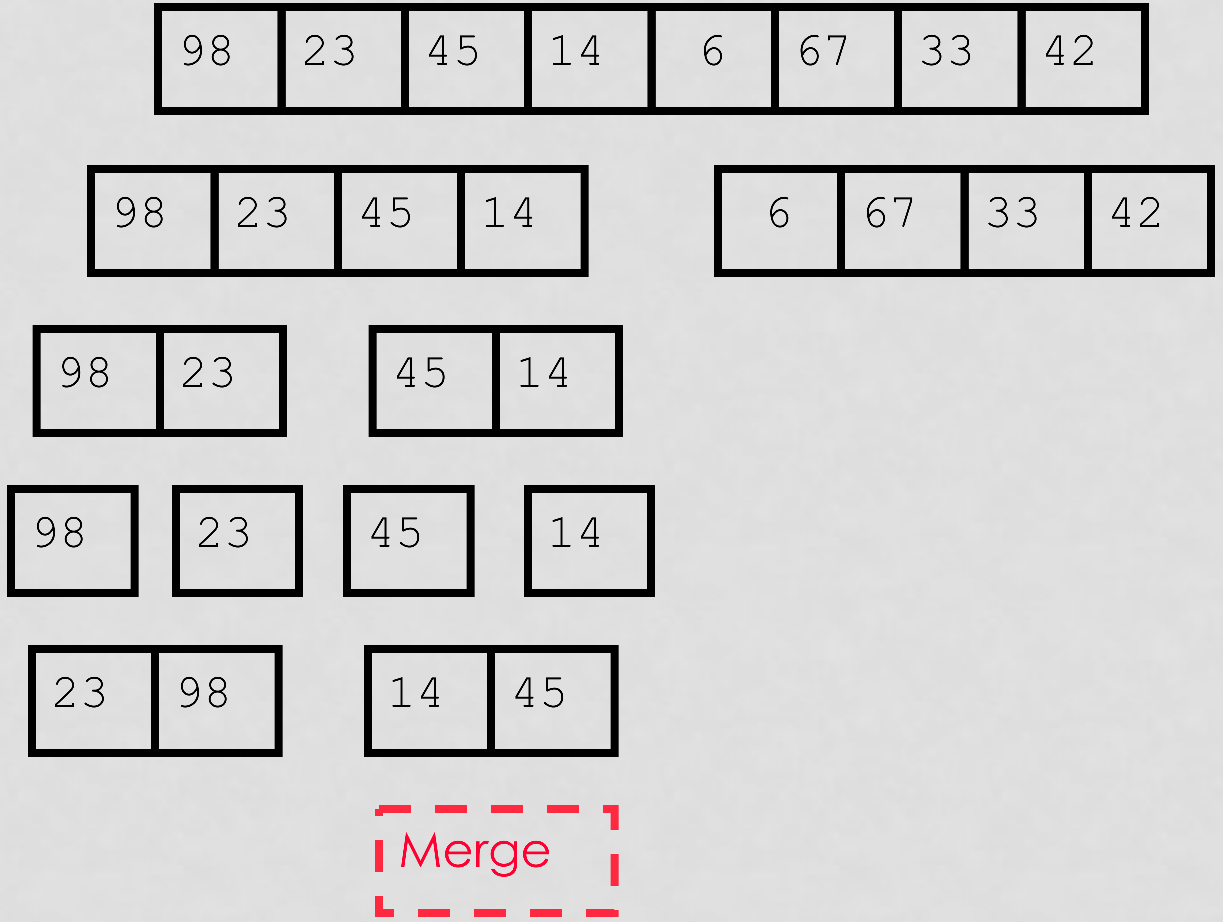

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23 98
Merge

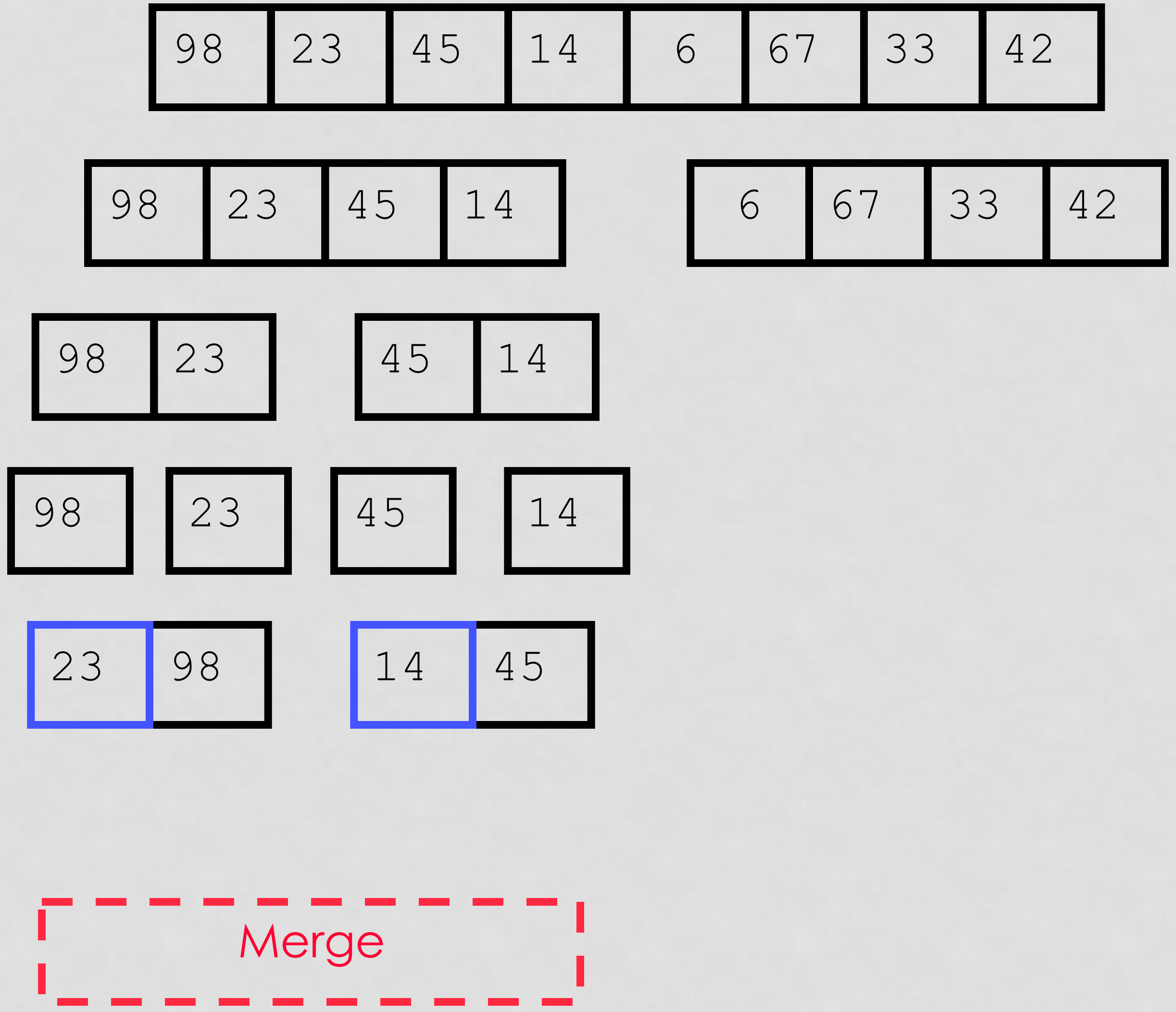

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98

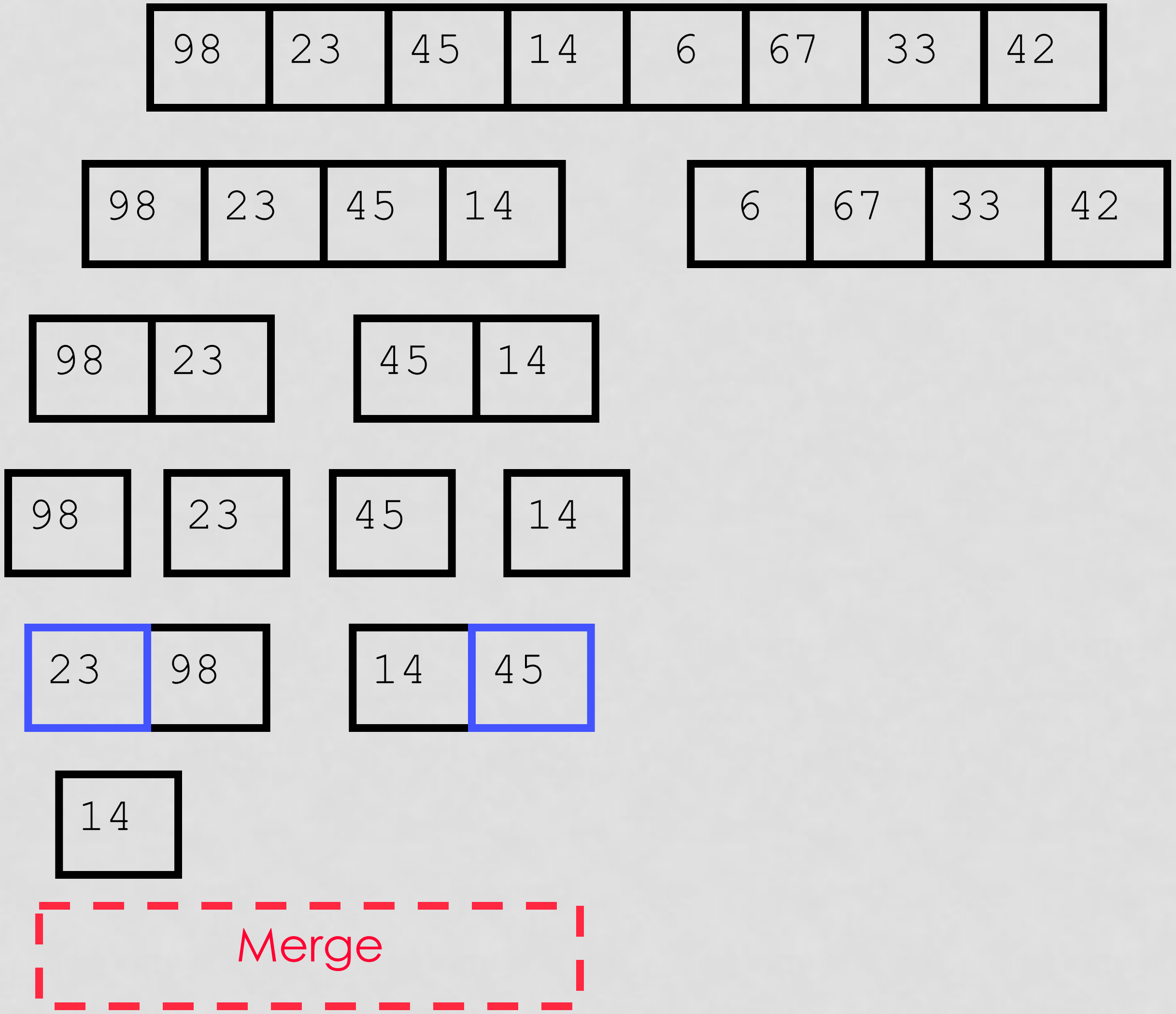

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98

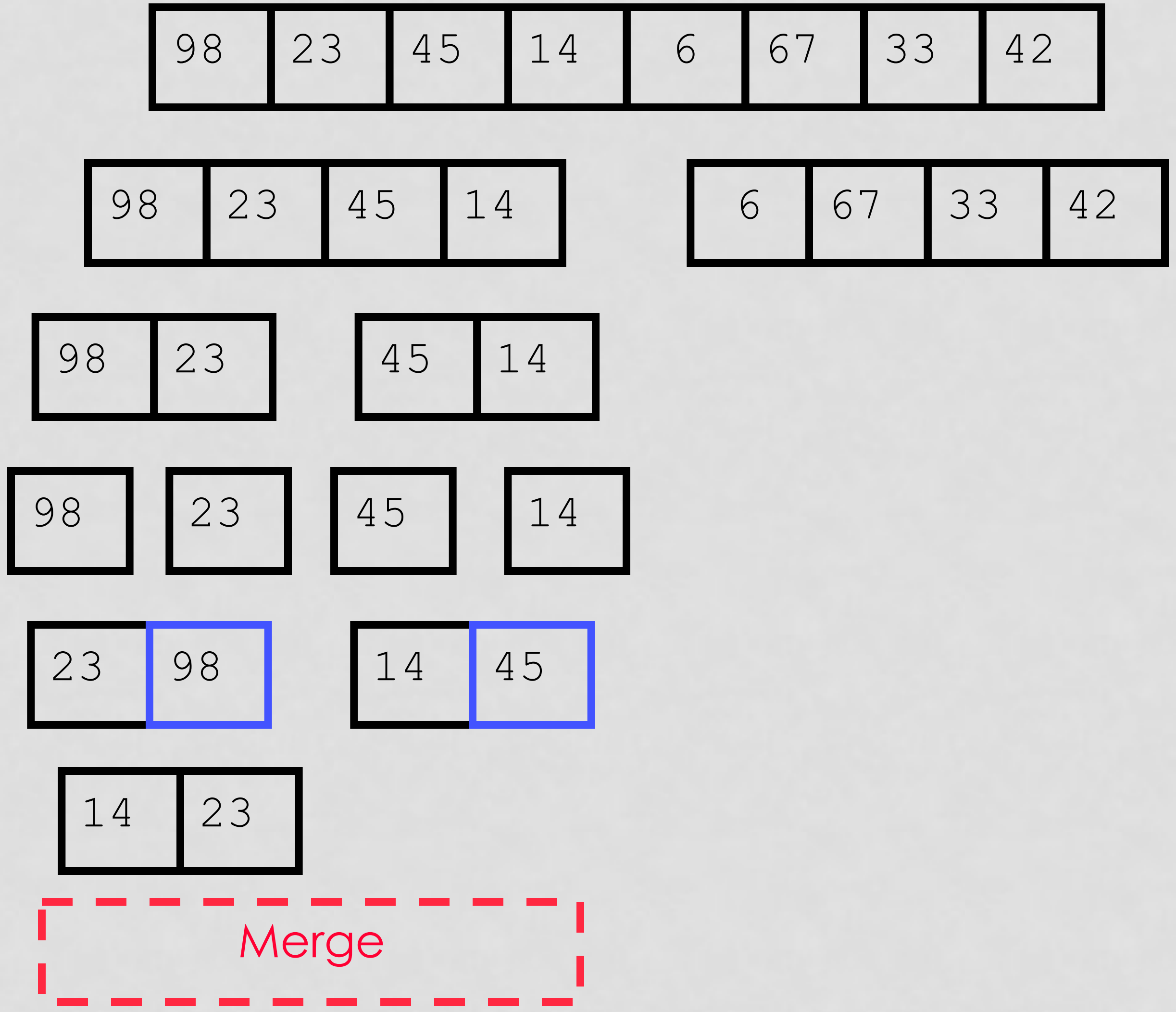
Merge

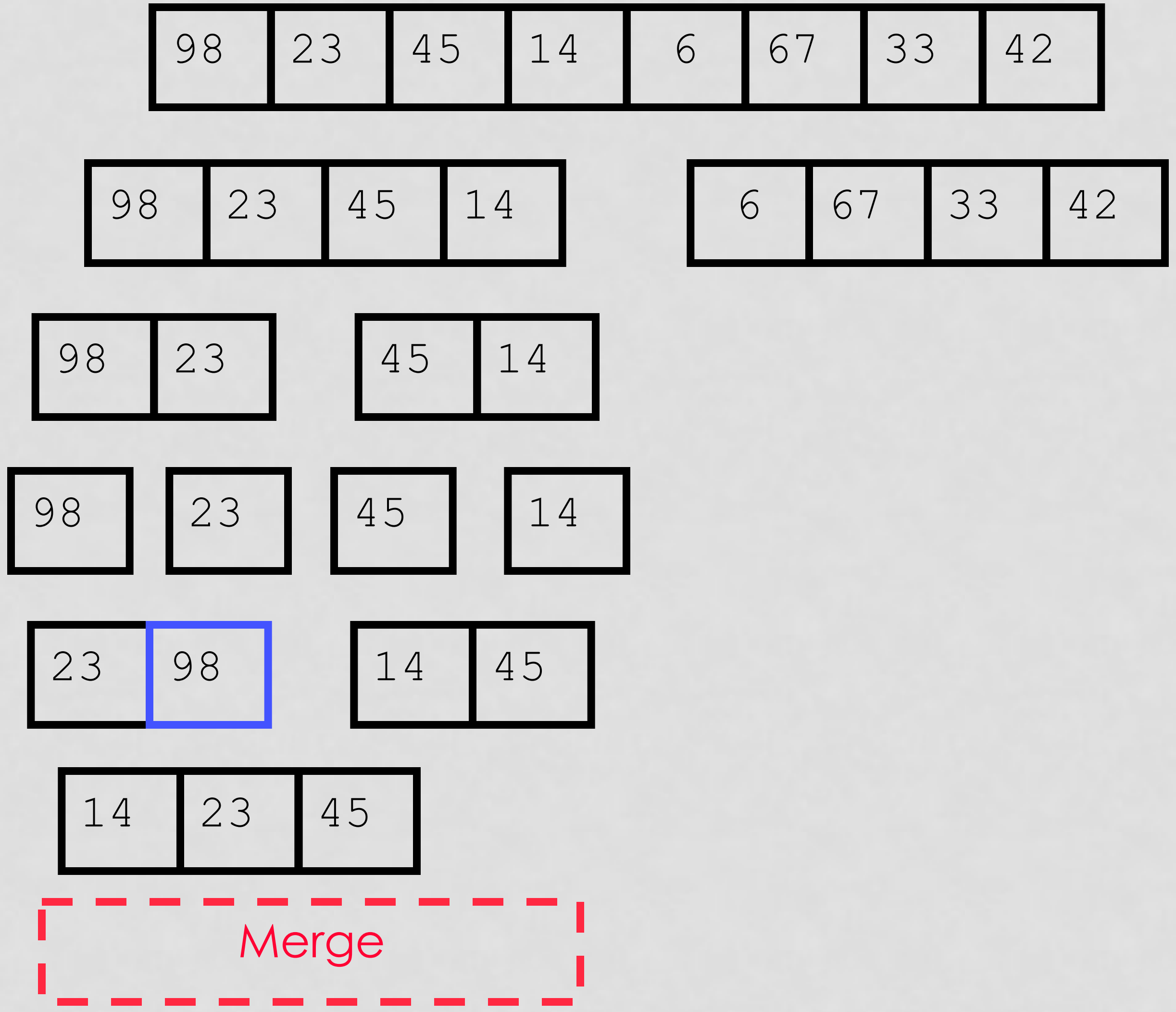

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14
Merge

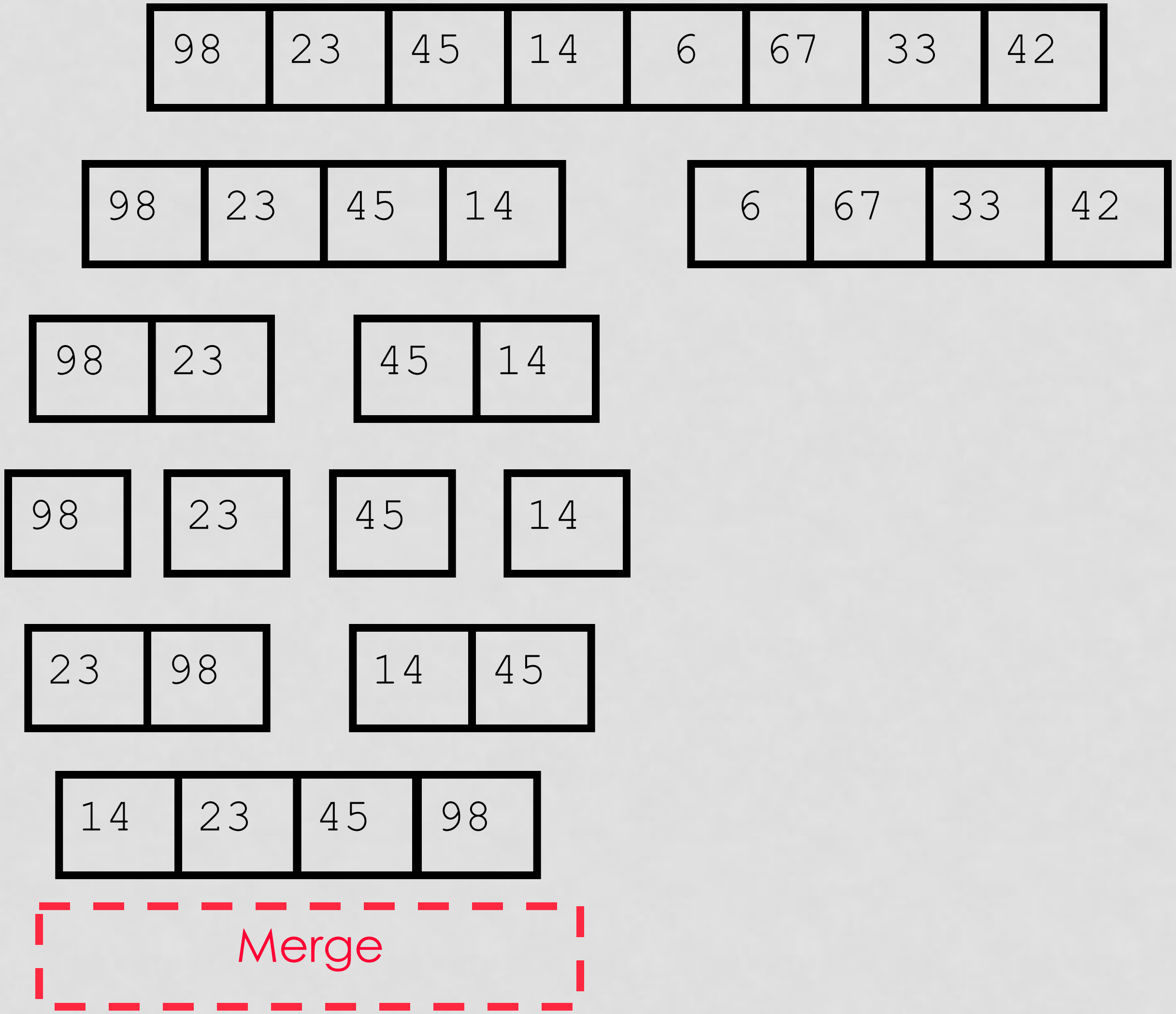

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
Merge

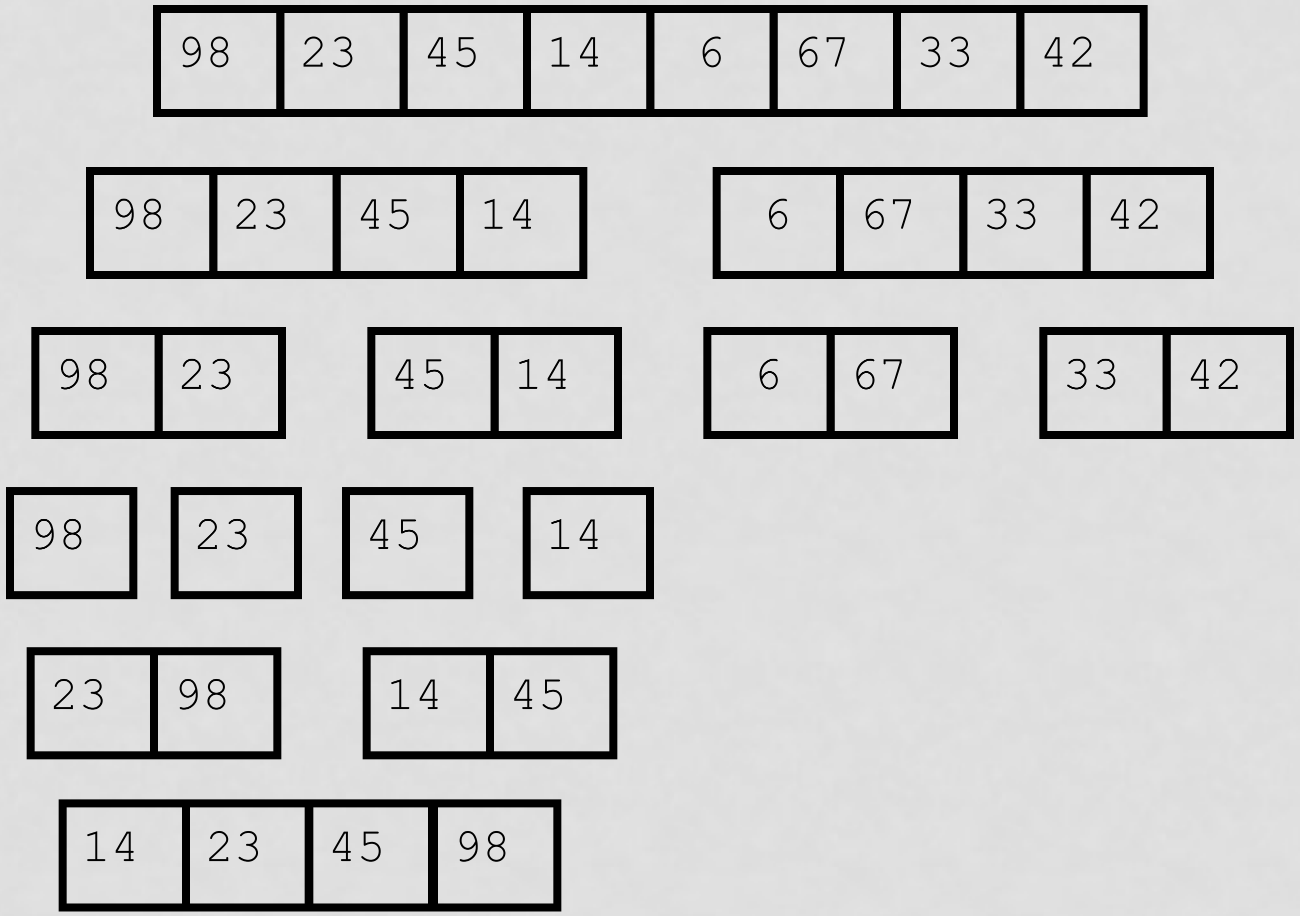

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
Merge

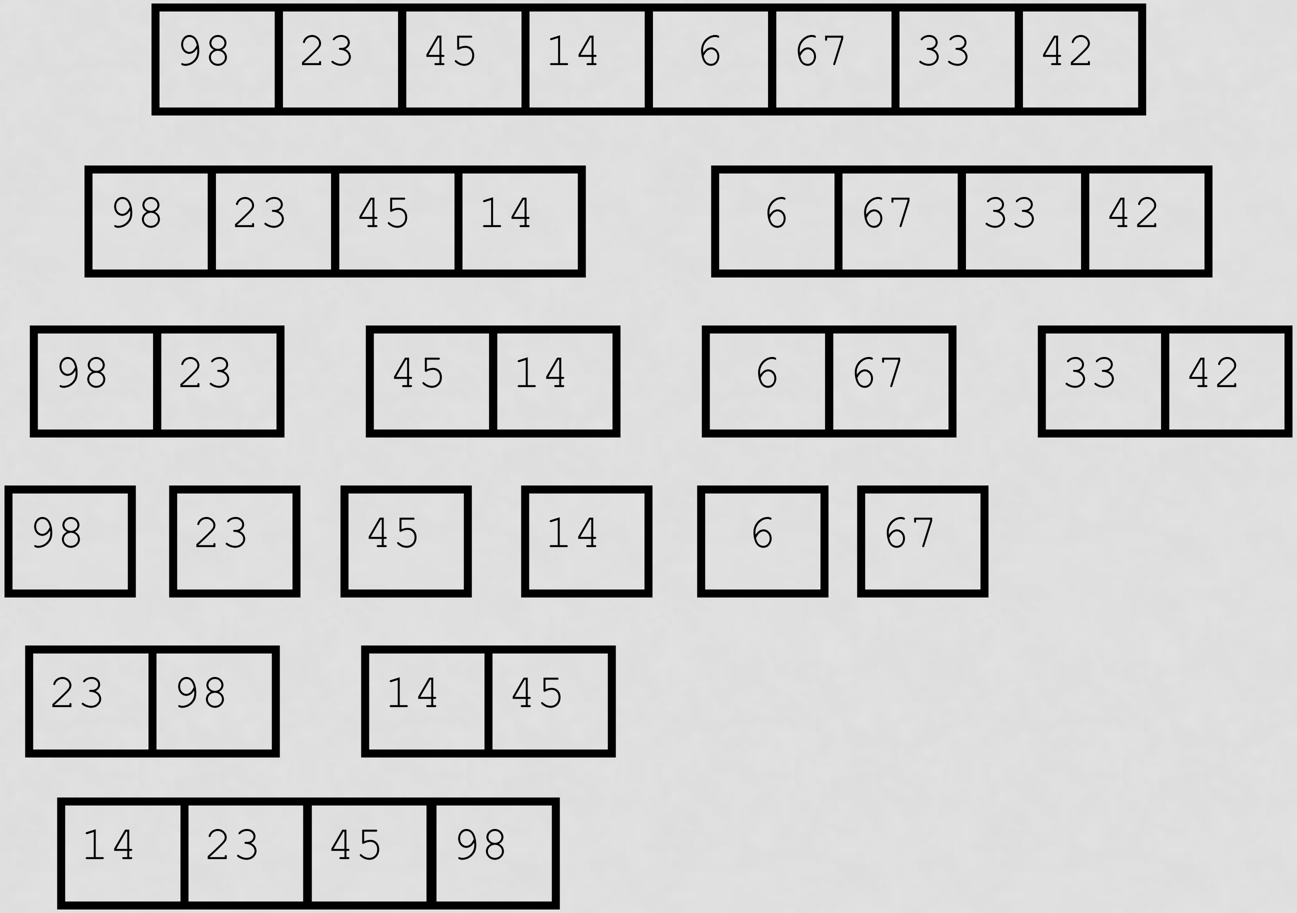

98
23
45
14
6
67
33
42
98
23
45
14
6
67
33
42
98
23
45
14
98
23
45
14
23
98
14
45
14
Merge

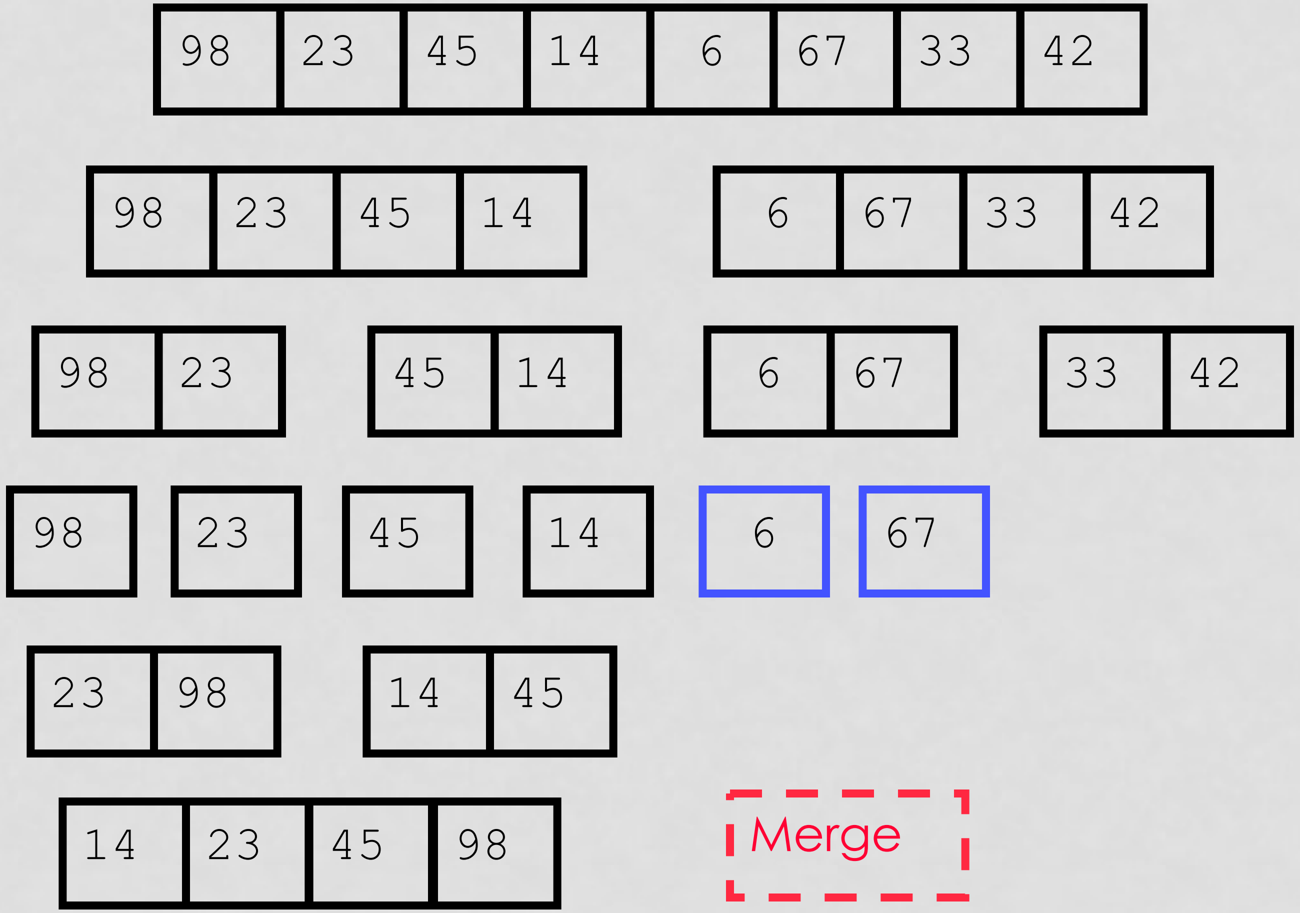

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23
Merge

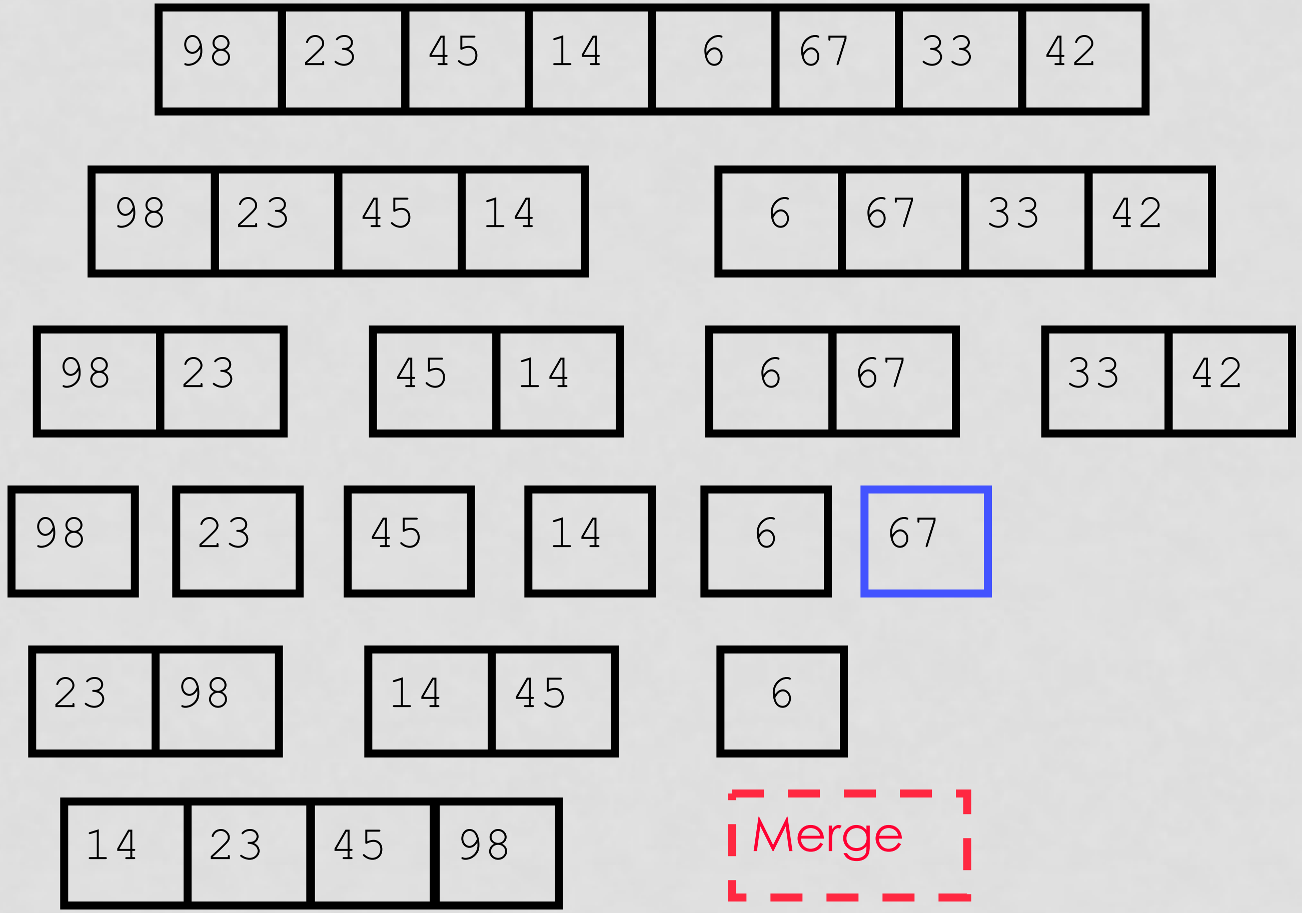

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45
Merge

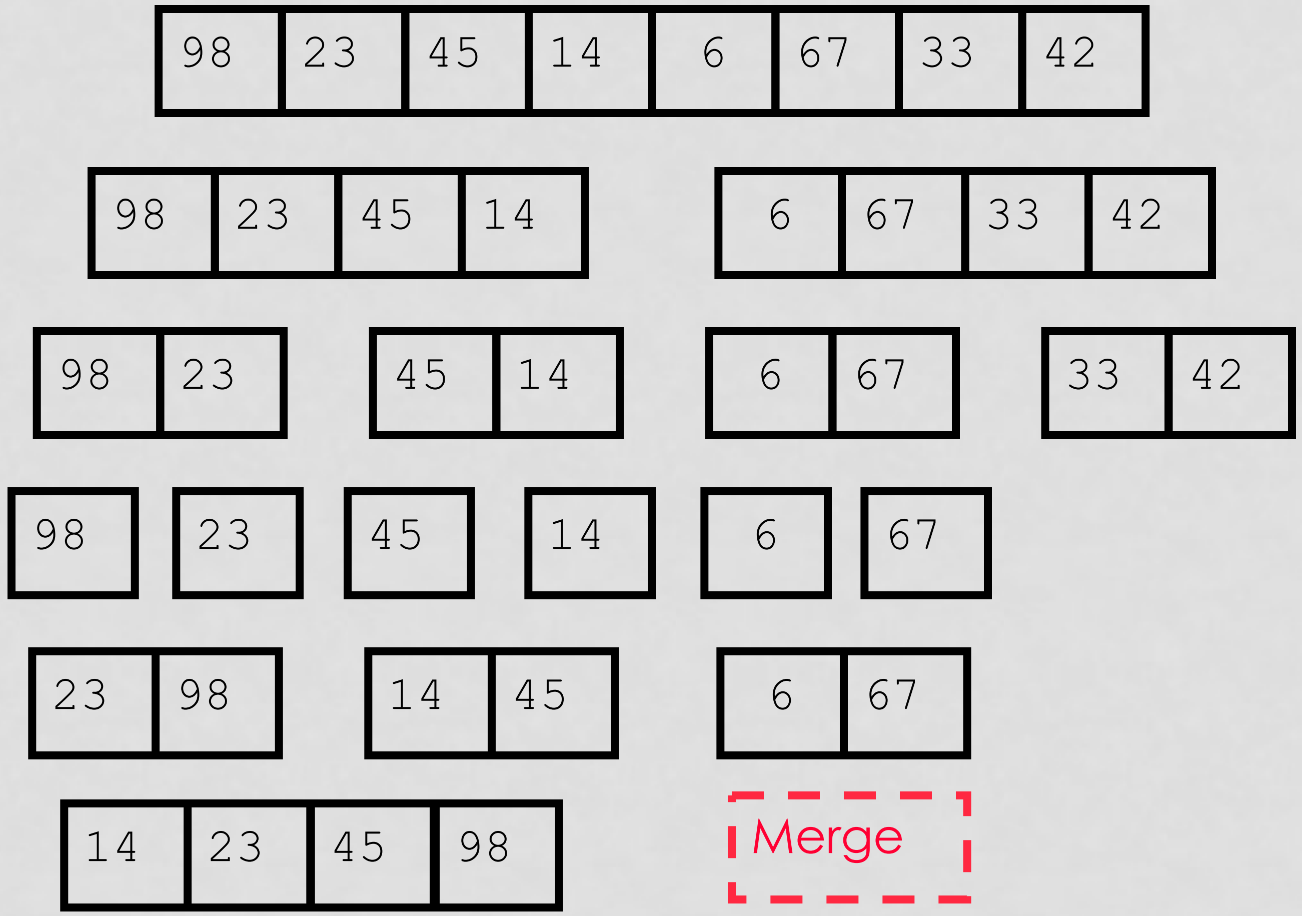

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45 98
Merge

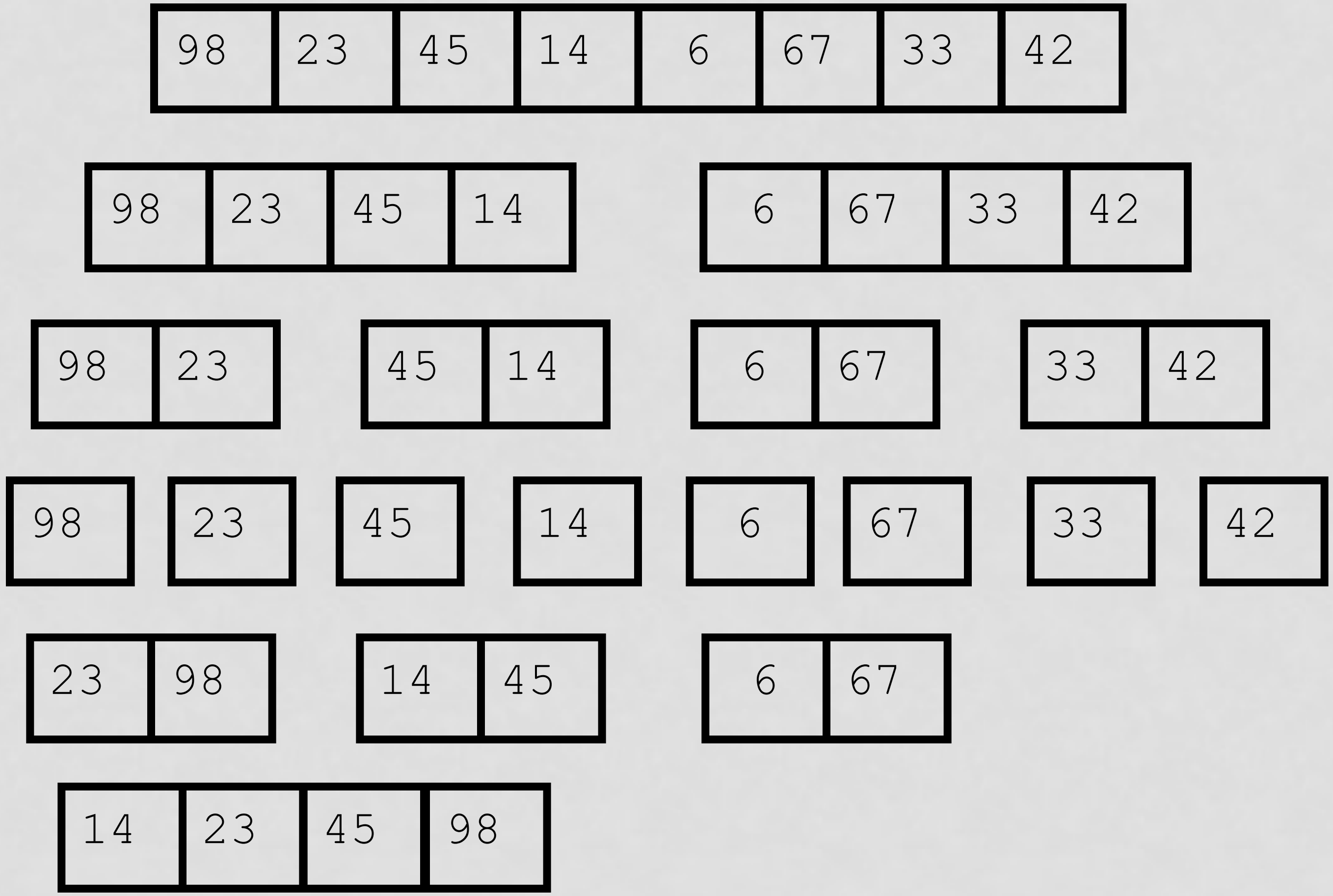

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
23 98
14 45
14 23 45 98

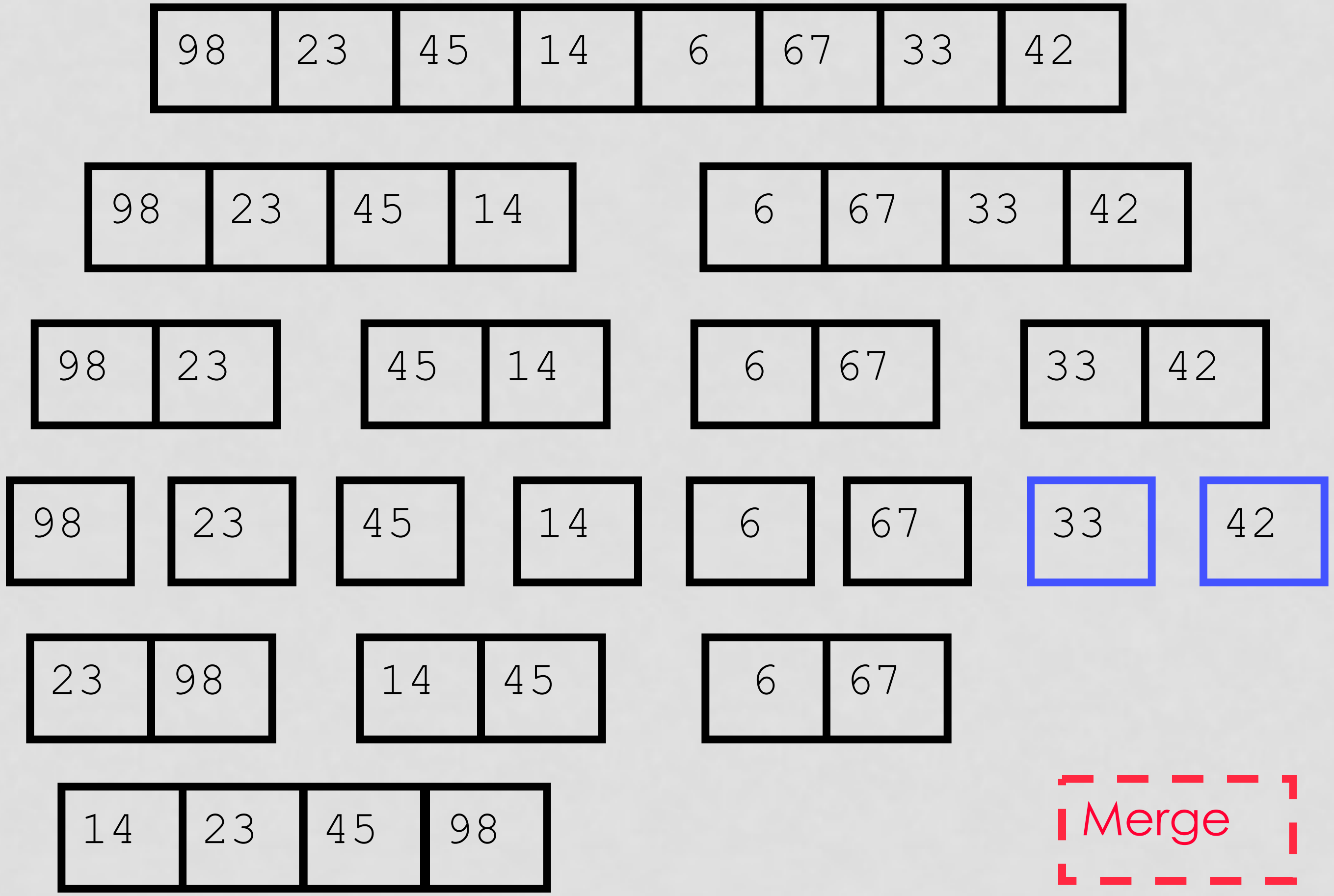

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98

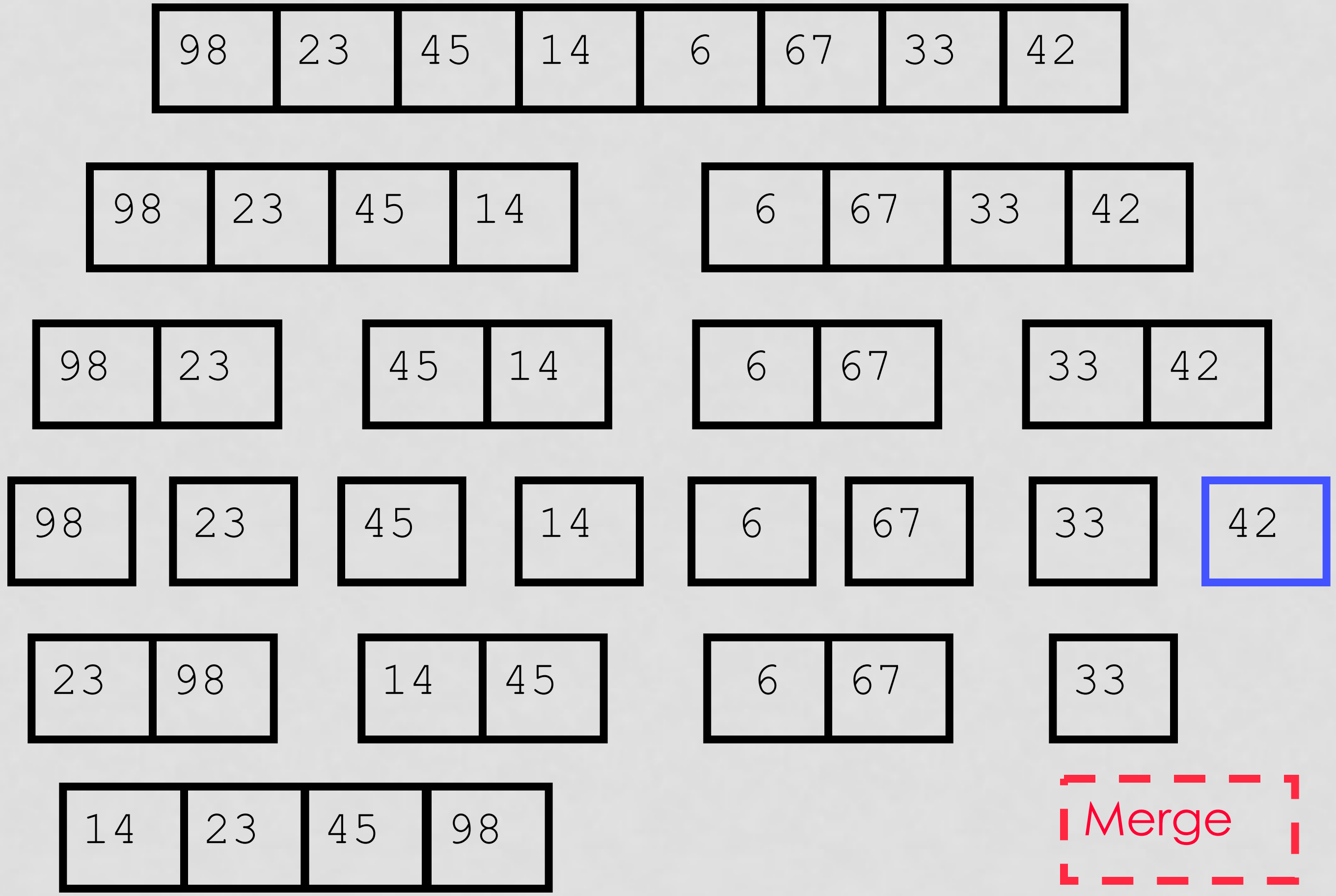

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98
Merge

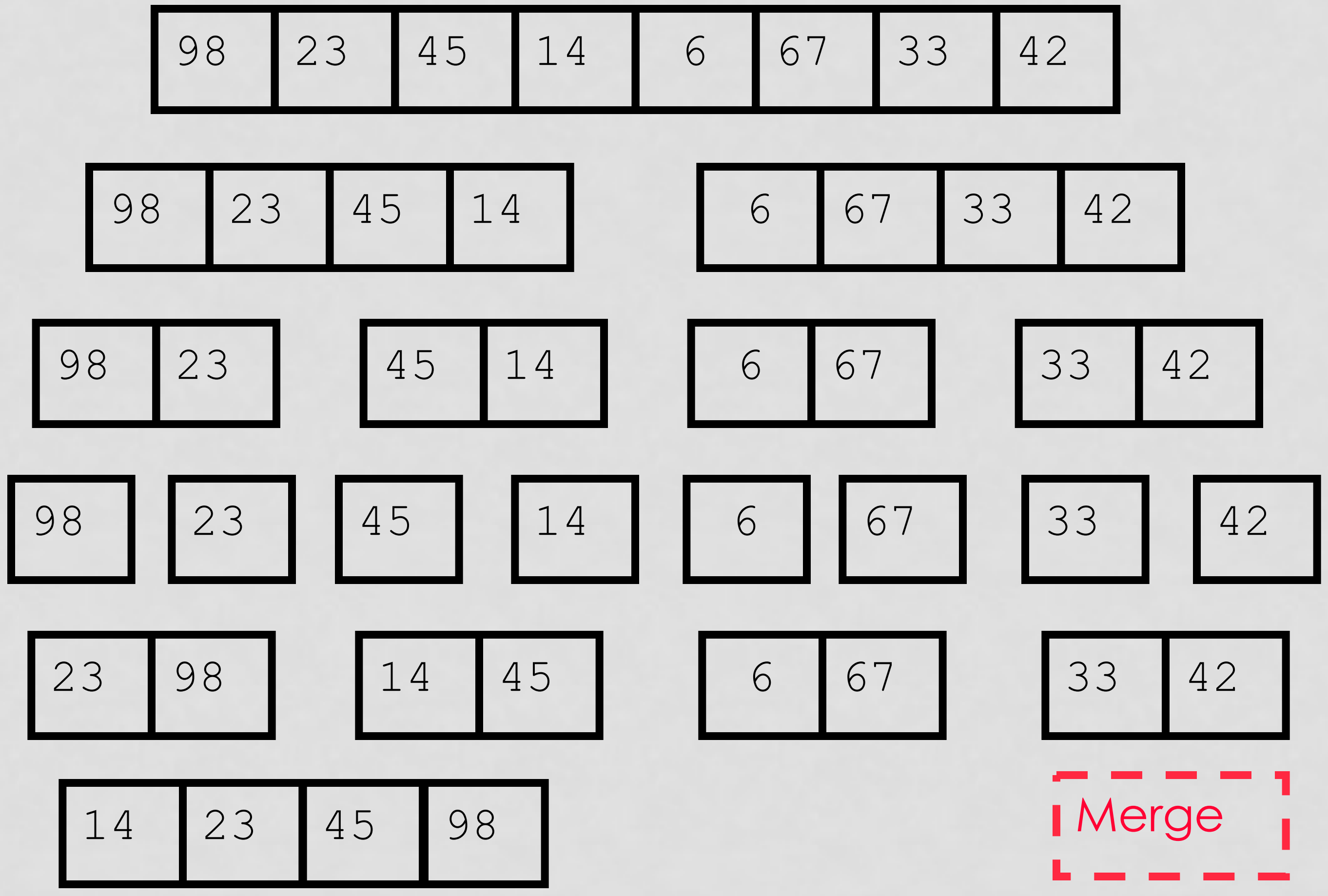

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
6
14 23 45 98
Merge

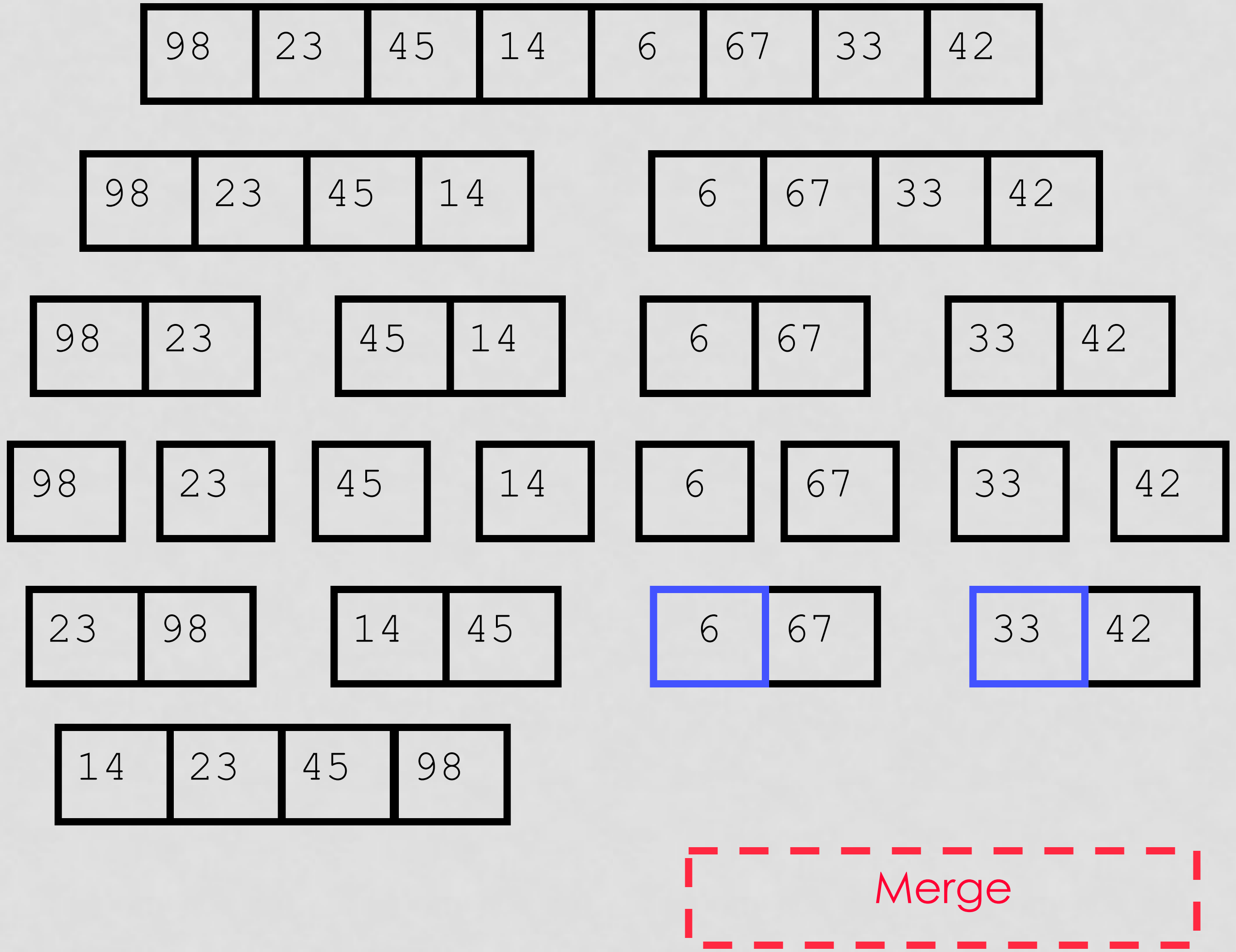

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
6 67
14 23 45 98
Merge

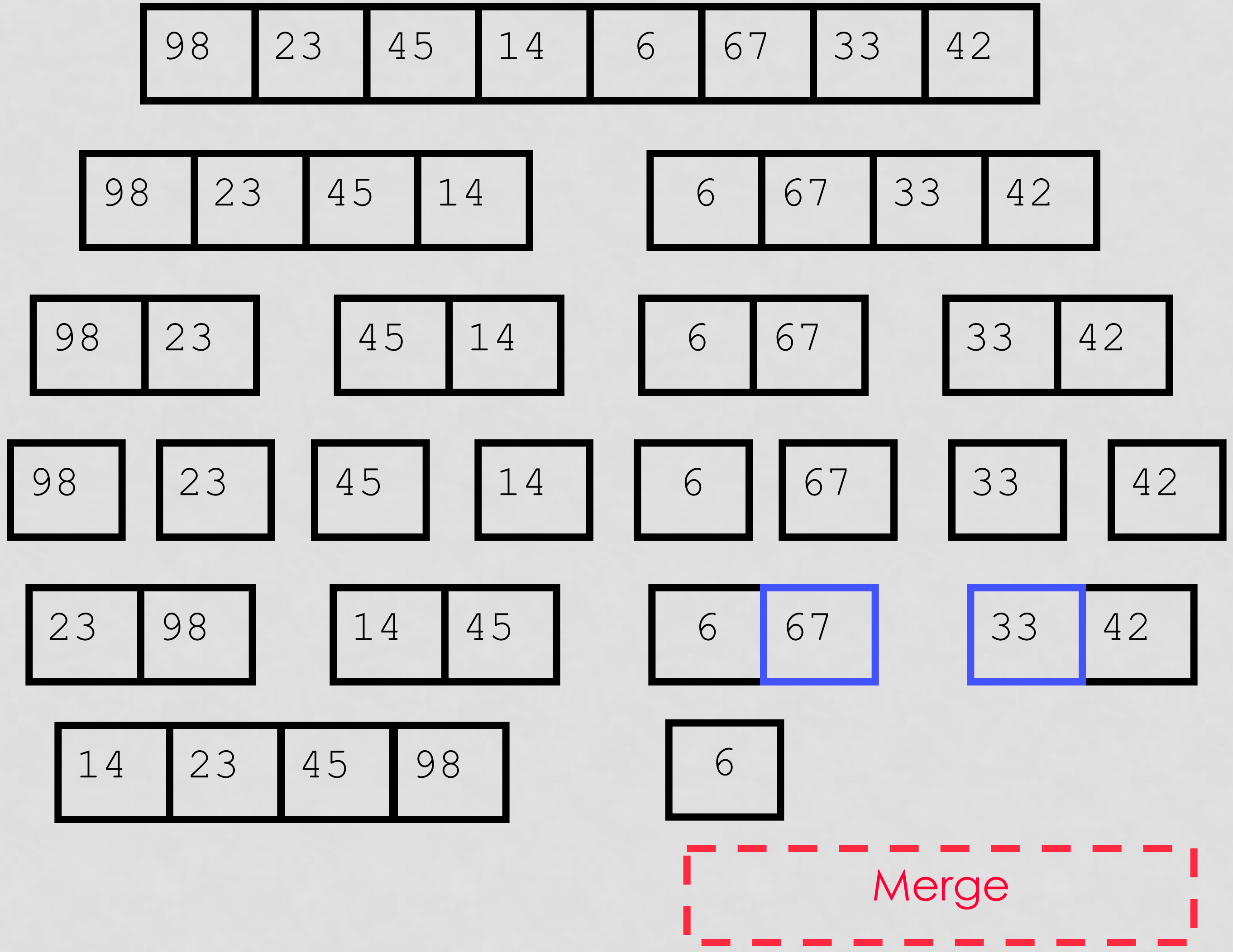

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
14 23 45 98

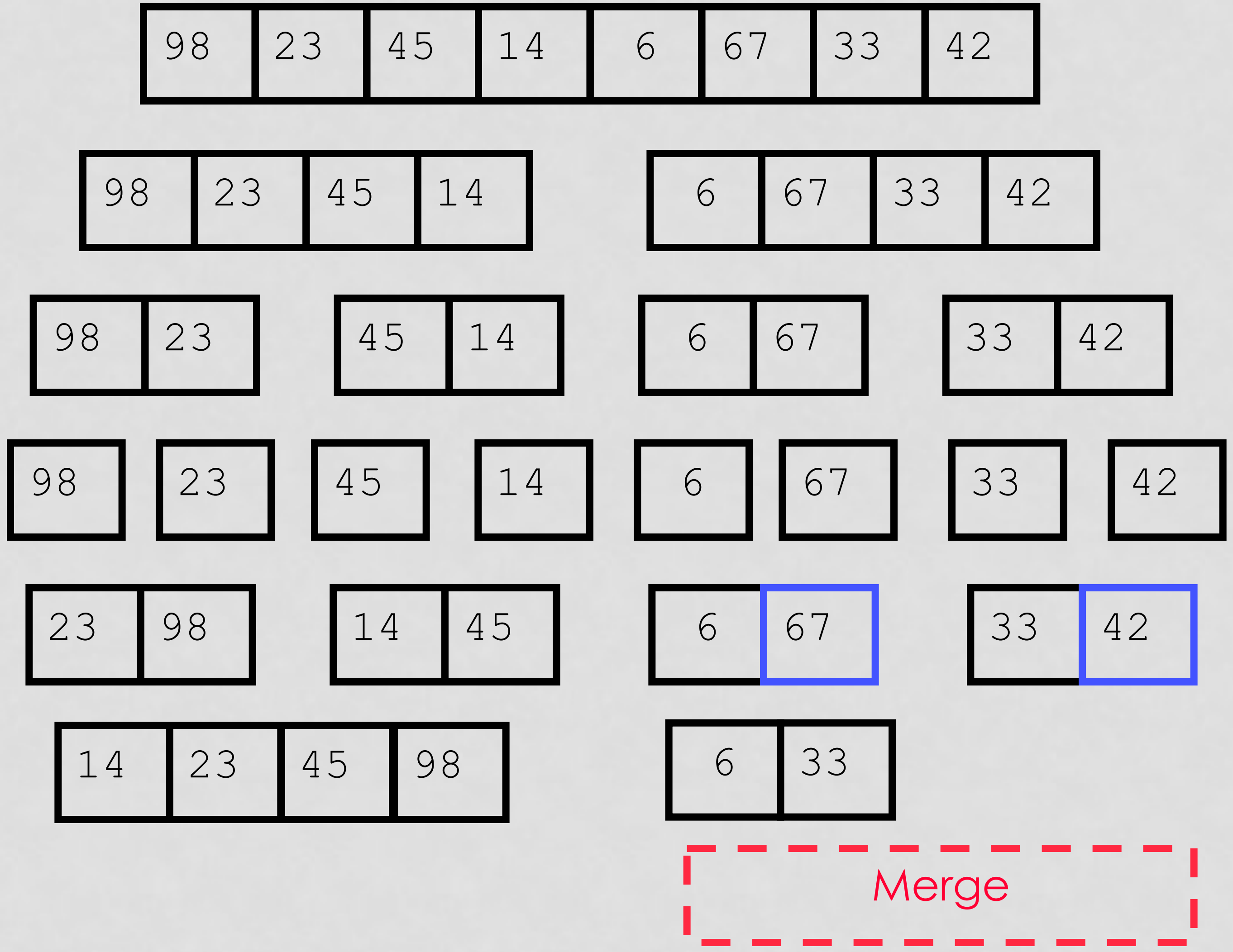

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
14 23 45 98
Merge

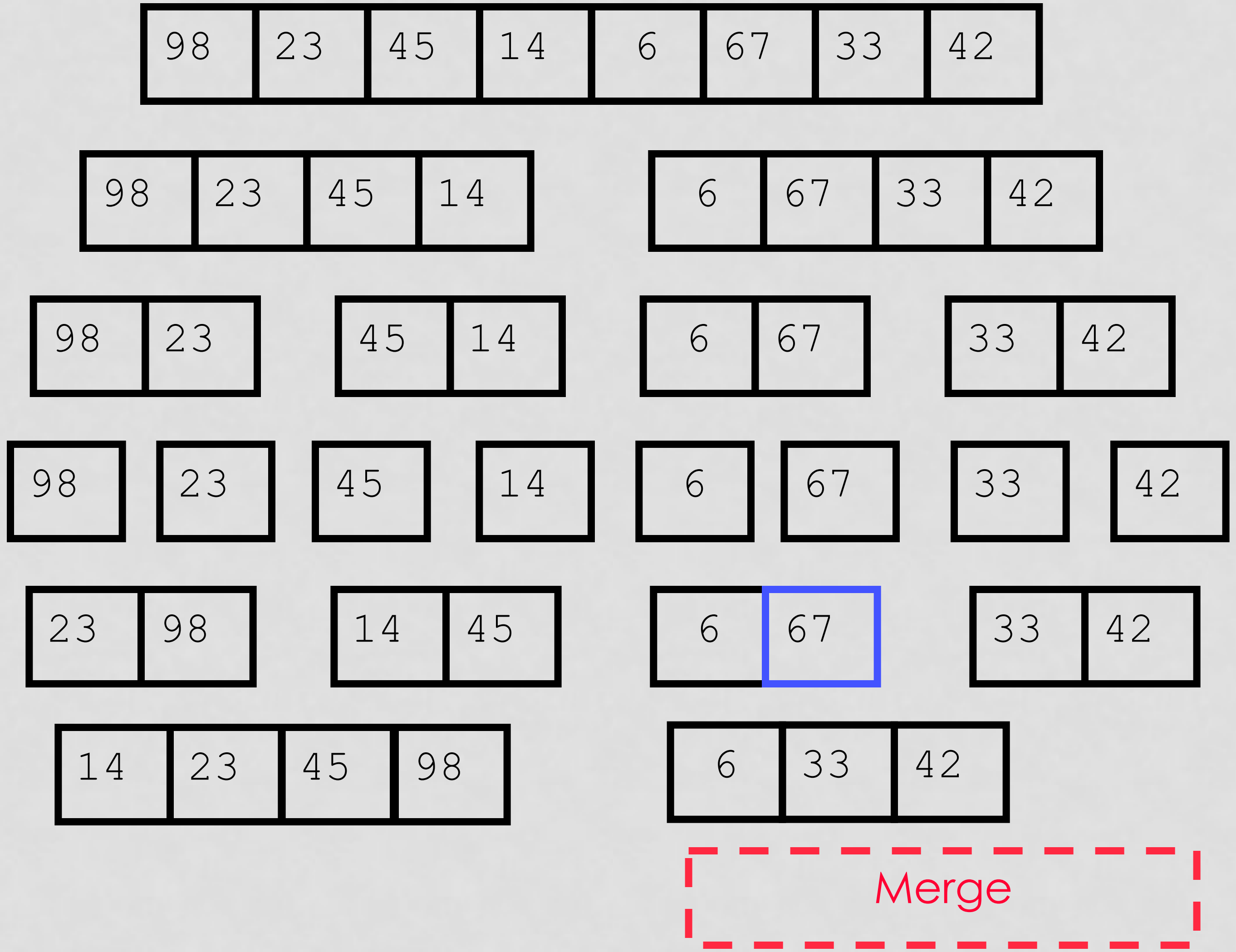

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33
14 23 45 98
Merge

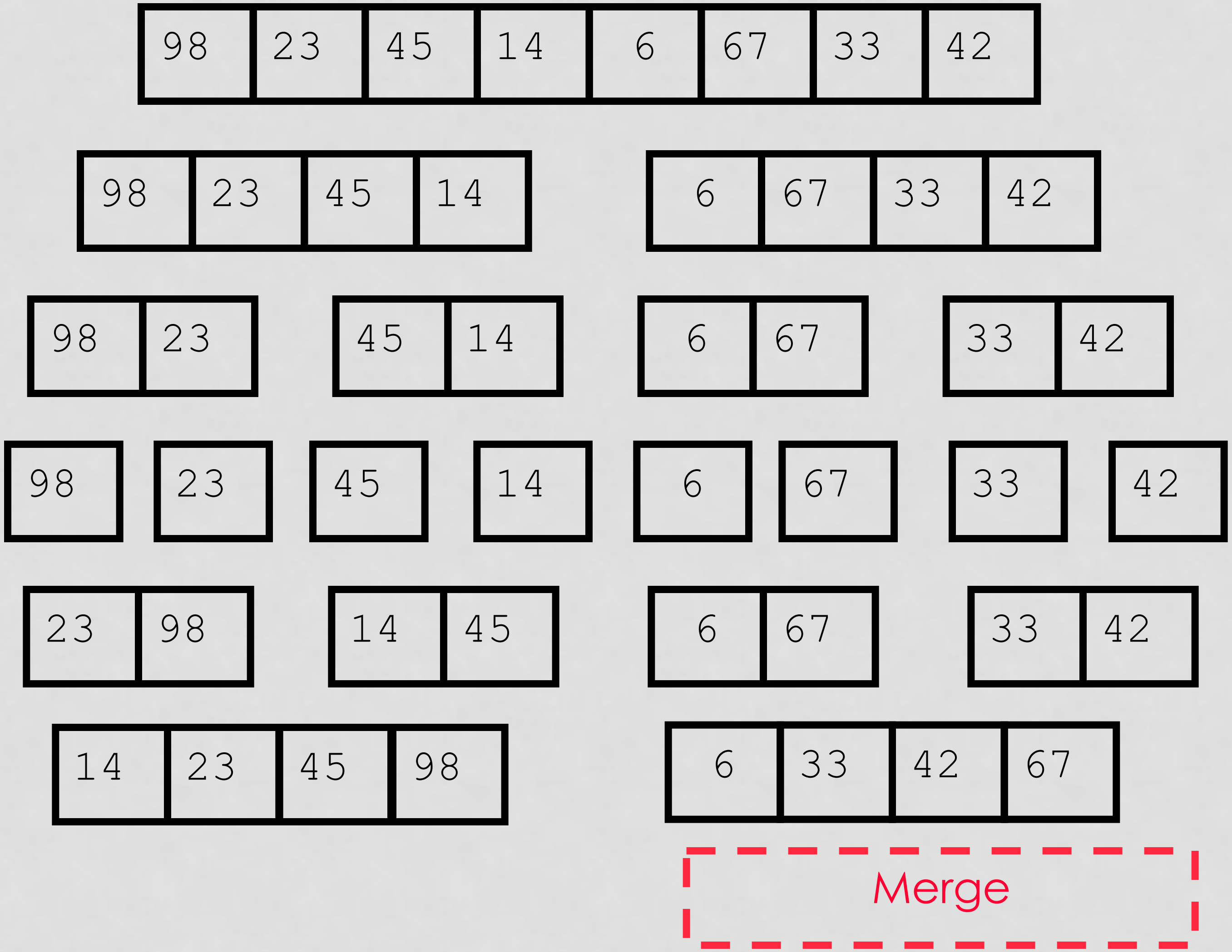

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
Merge


98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
Merge

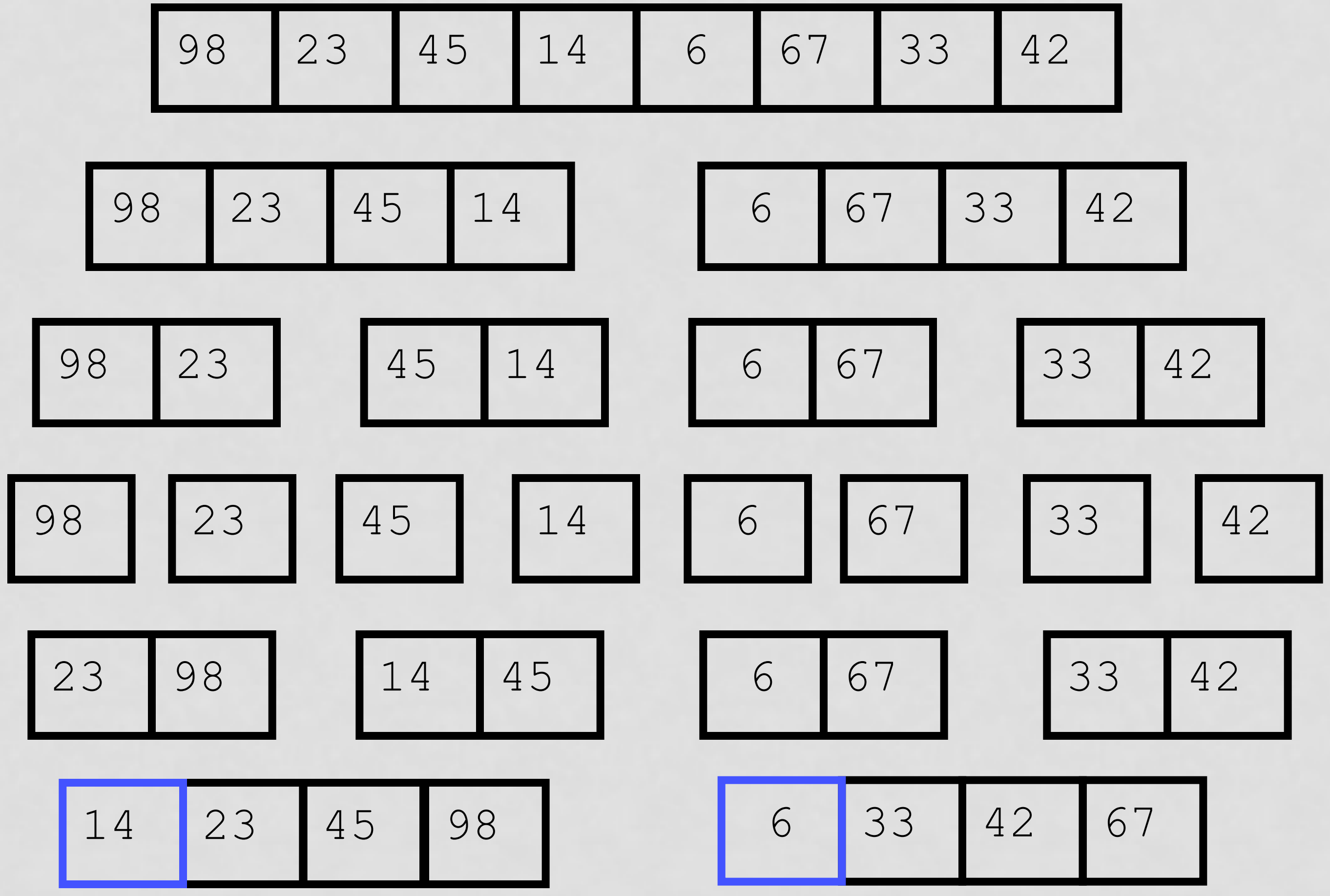

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6
Merge


98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33
Merge


98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42
Merge

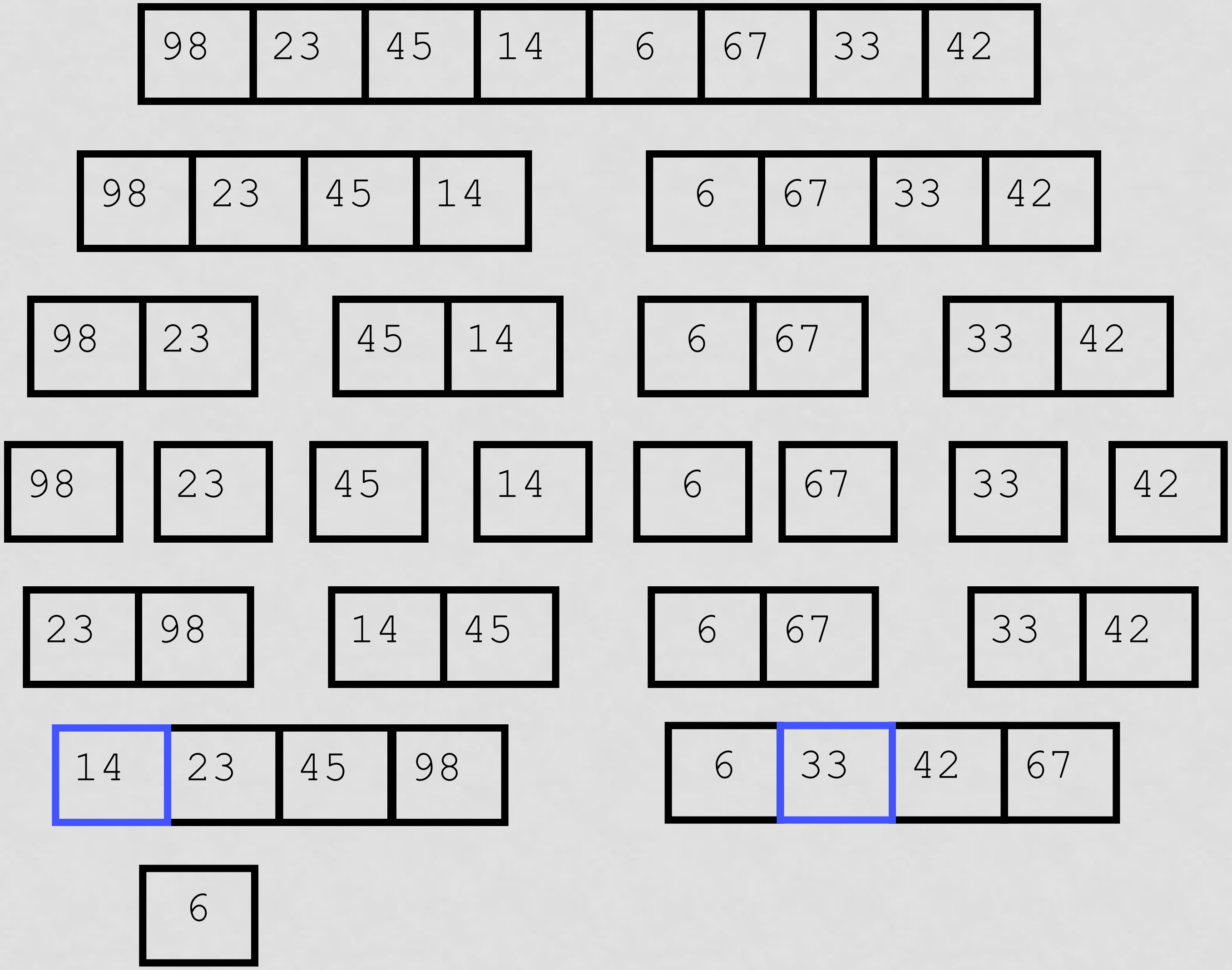


98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
Merge

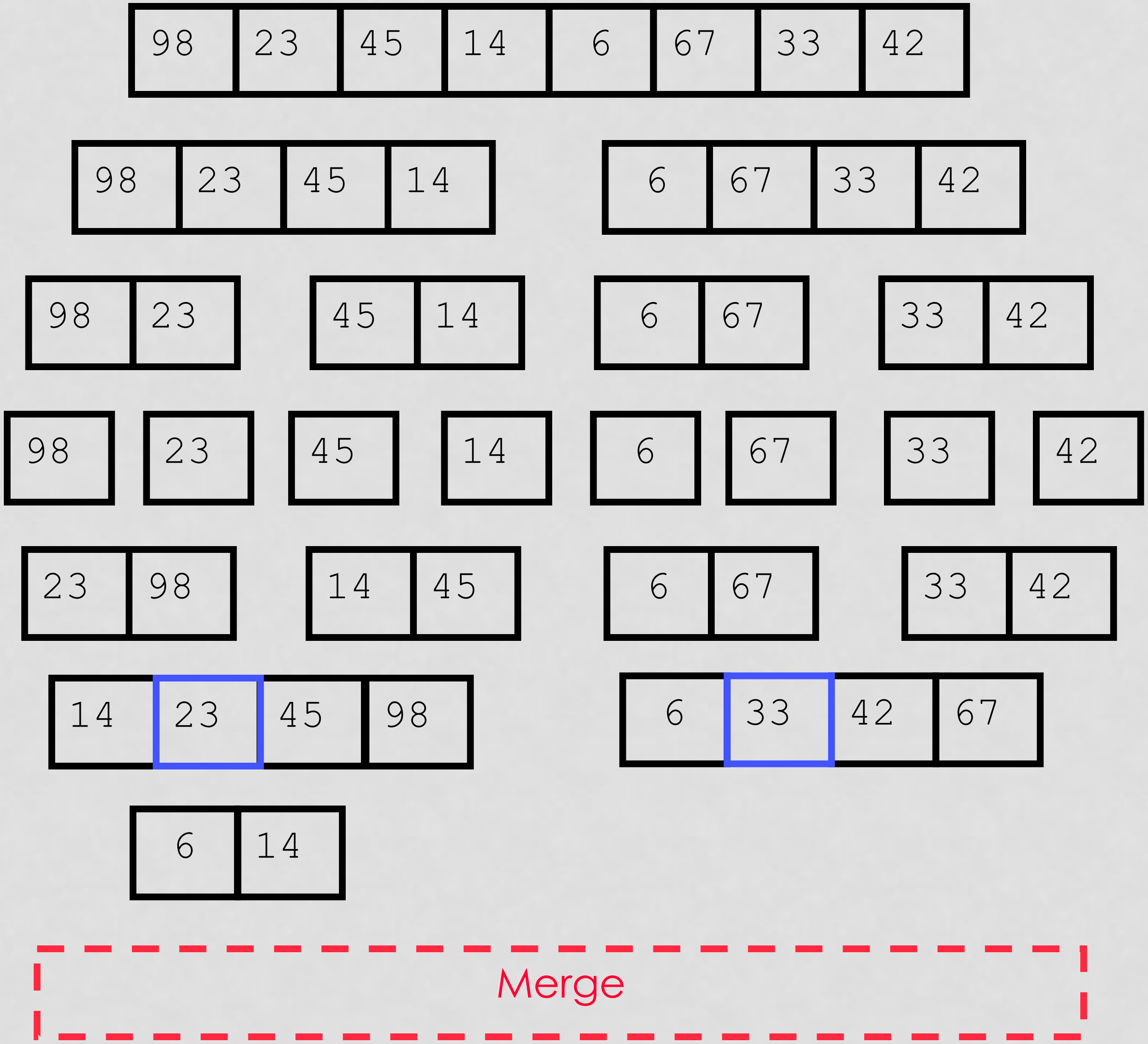


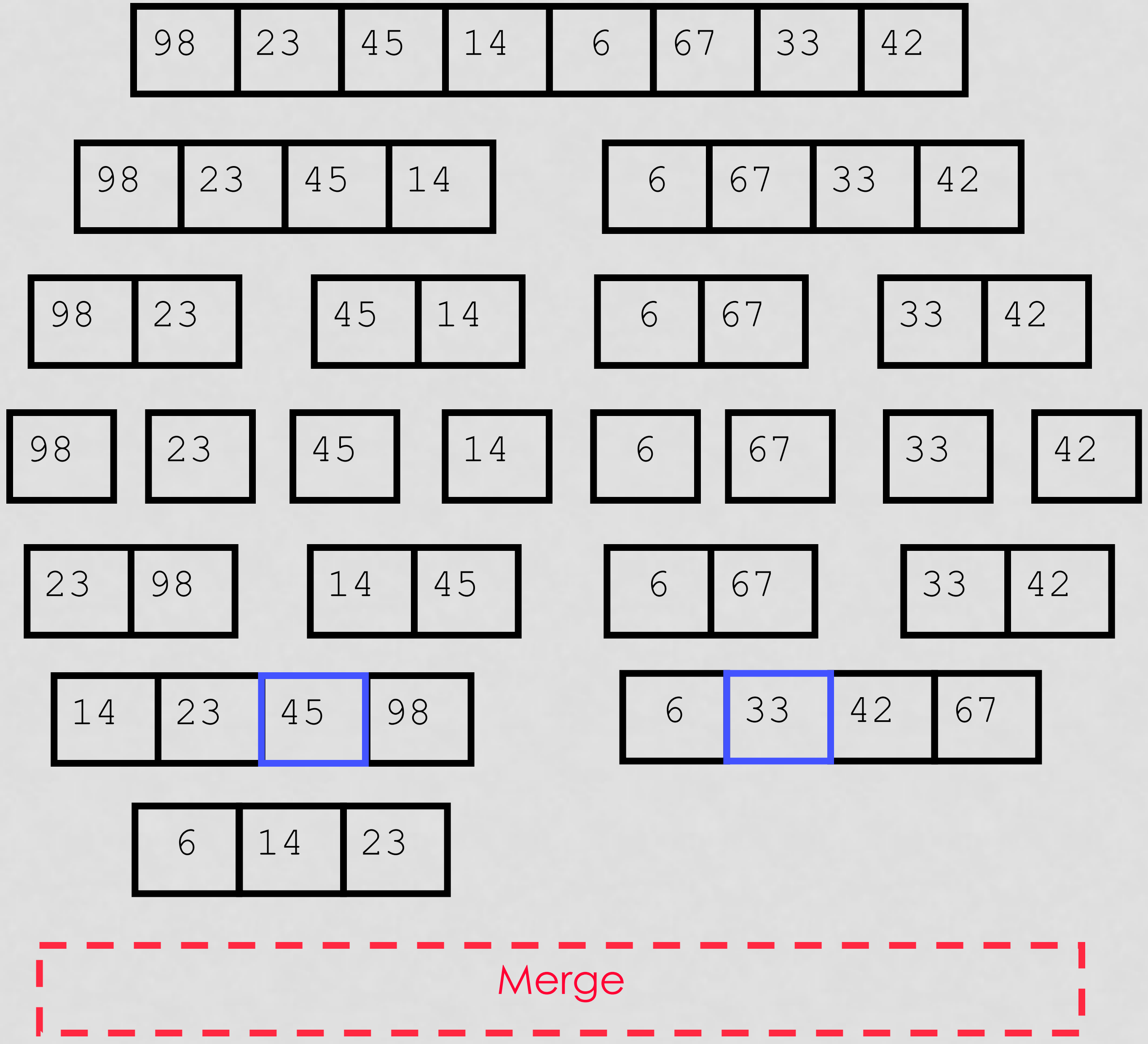
98 23 45 14 6 67 33 42
98 23 45 14 | 6 67 33 42
98 23 | 45 14 | 6 67 | 33 42
98 | 23 | 45 | 14 | 6 | 67 | 33 | 42
23 98 | 14 45 | 6 67 | 33 42
14 23 45 98 | 6 33 42 67

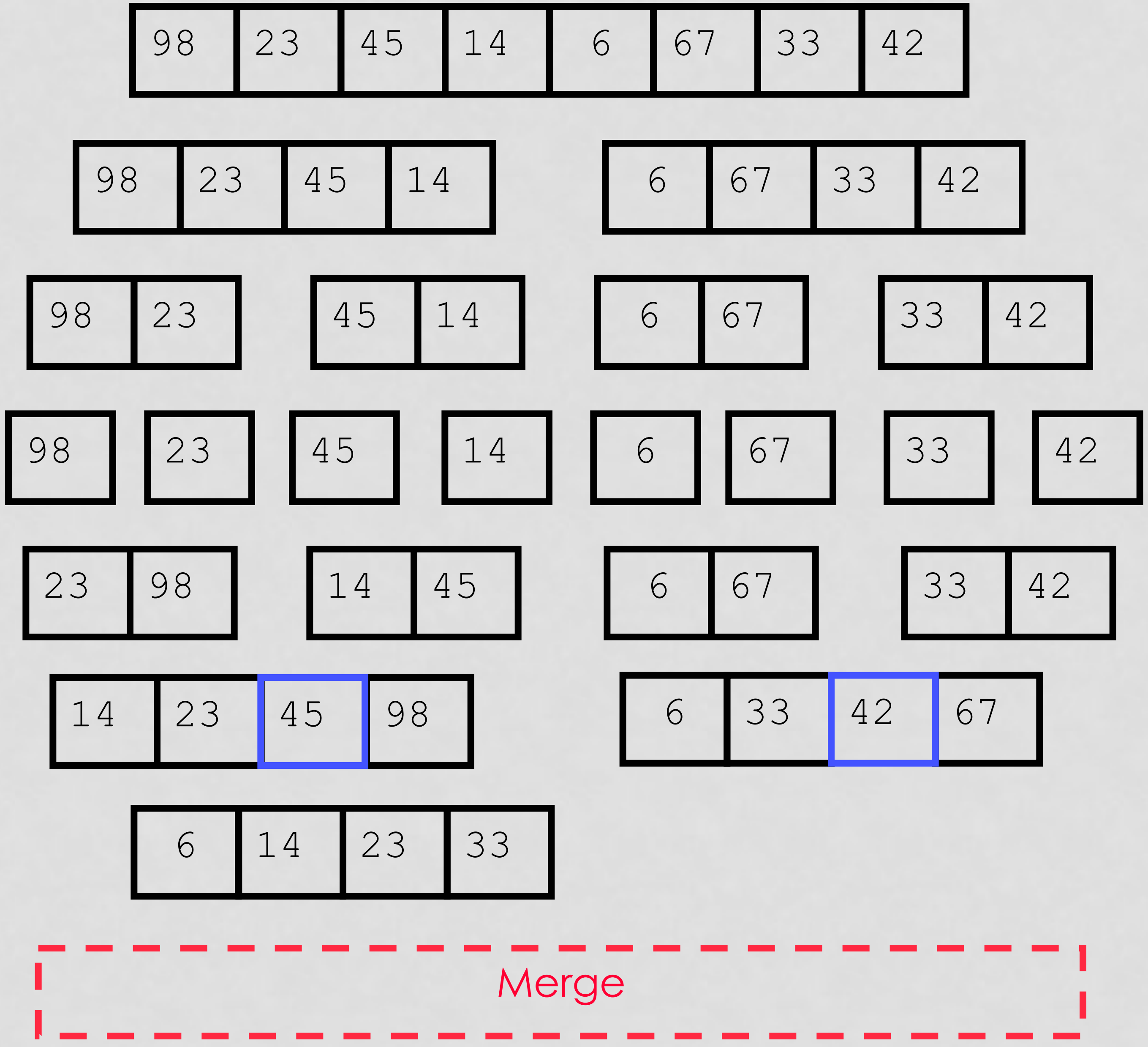
Merge

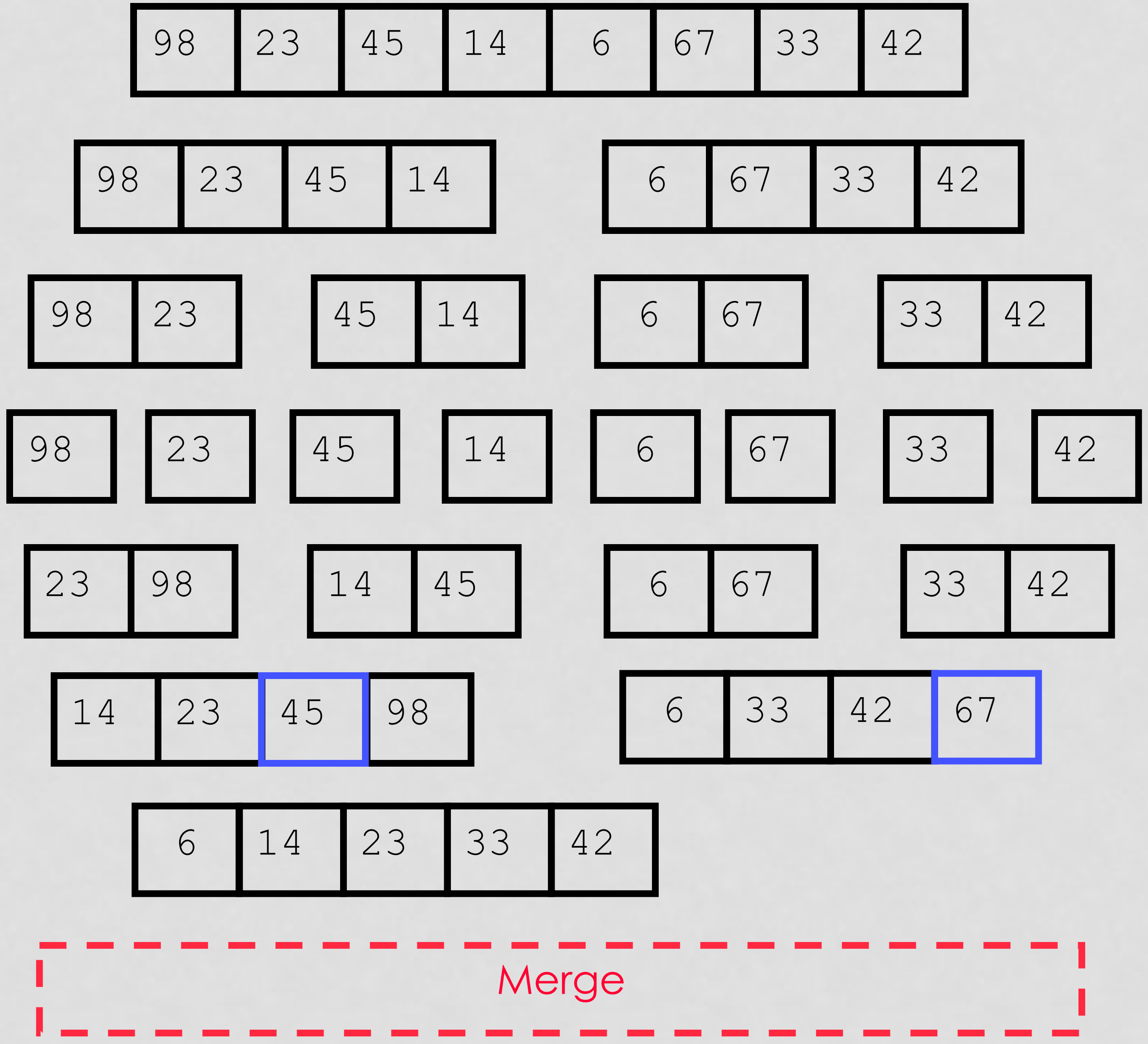


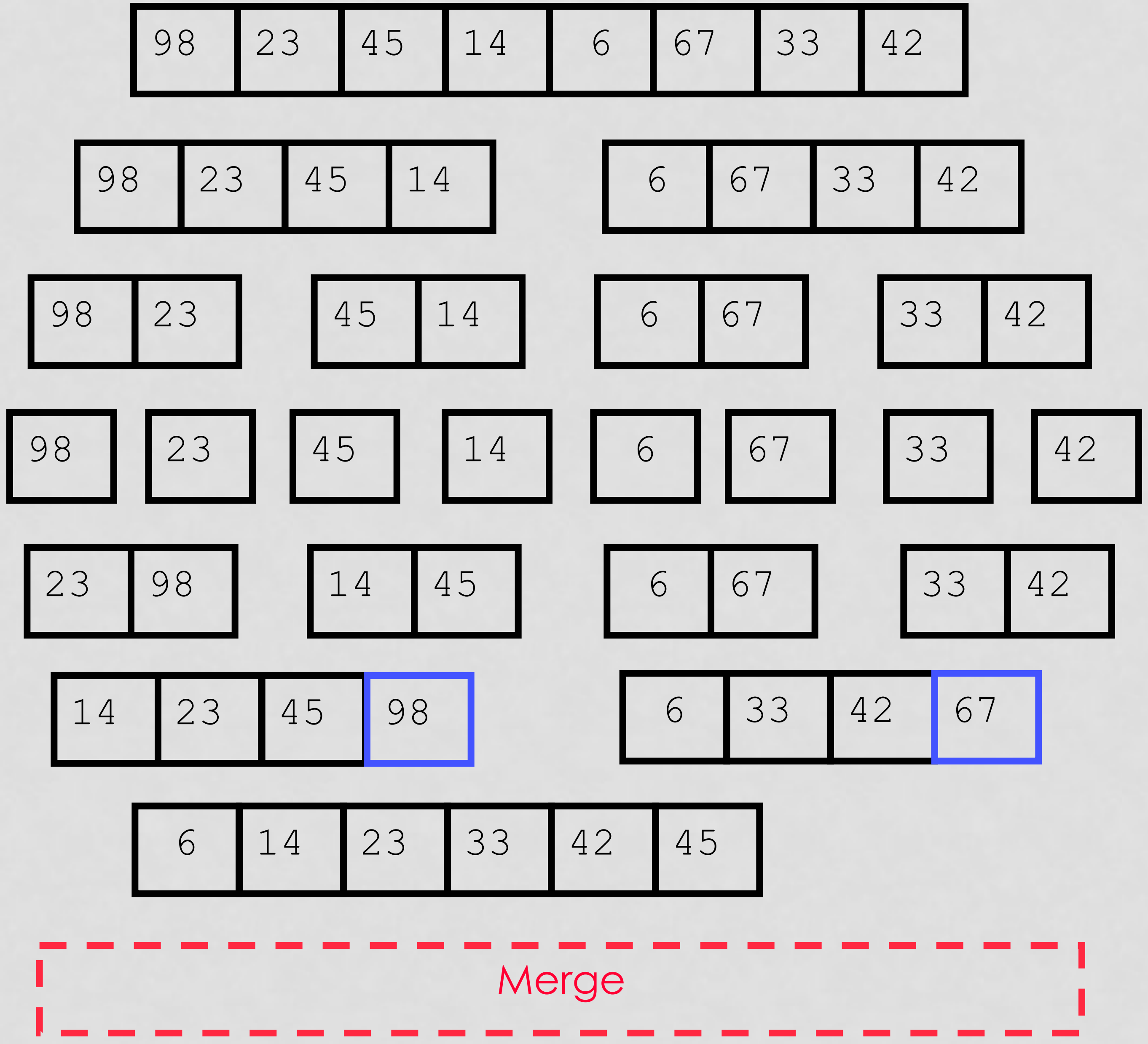
Merge

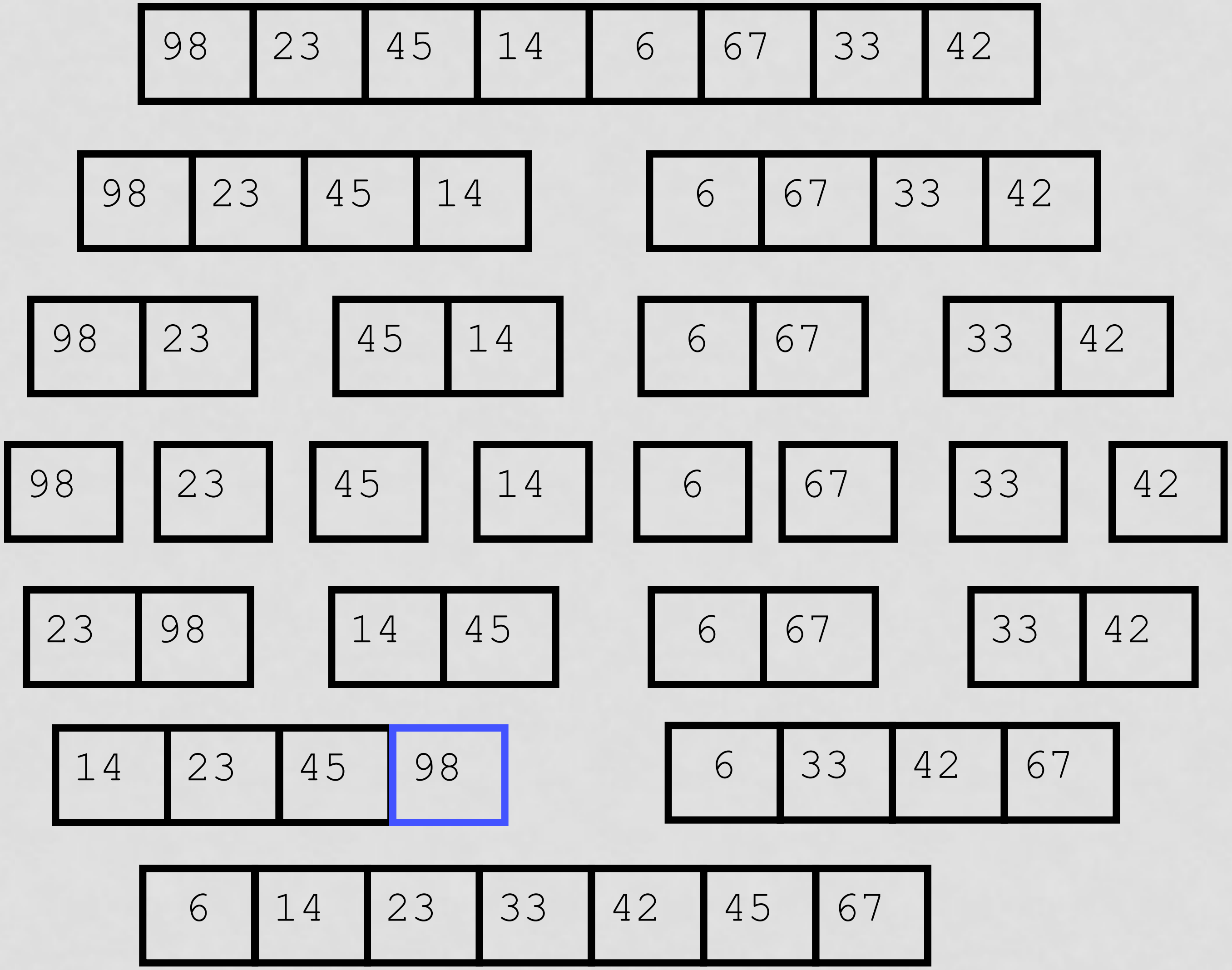

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14
Merge


98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23
Merge

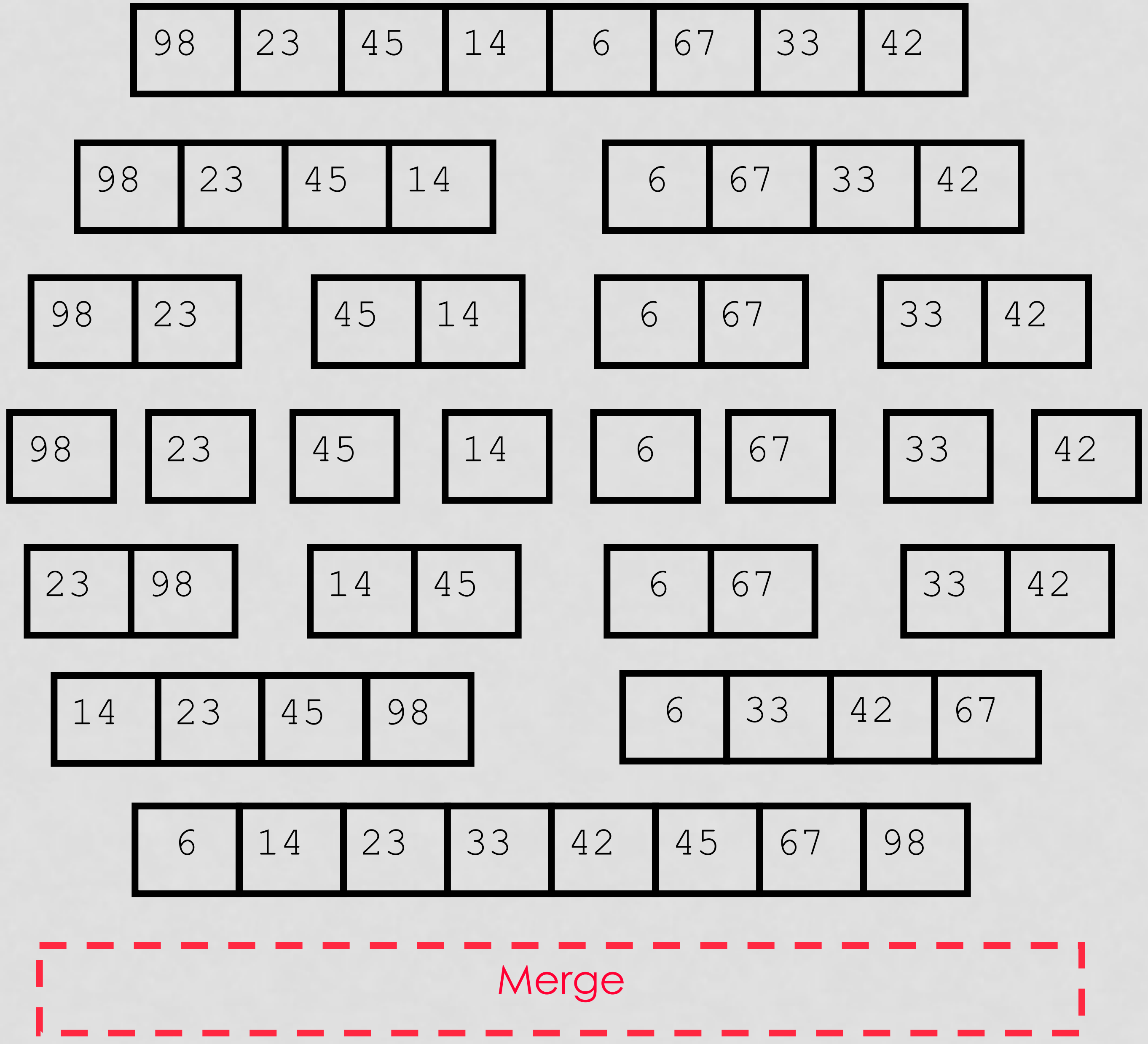

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33
Merge

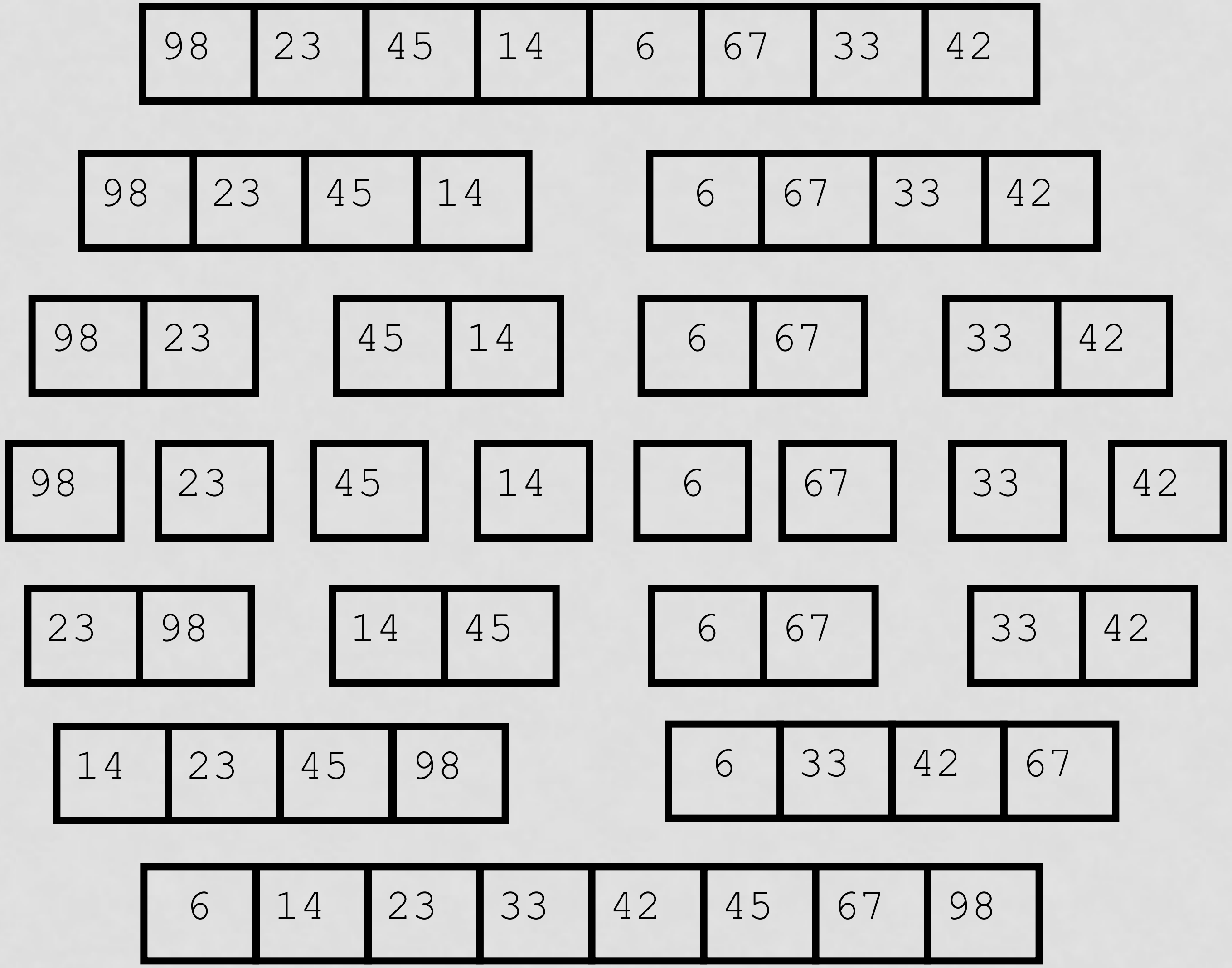

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42
Merge

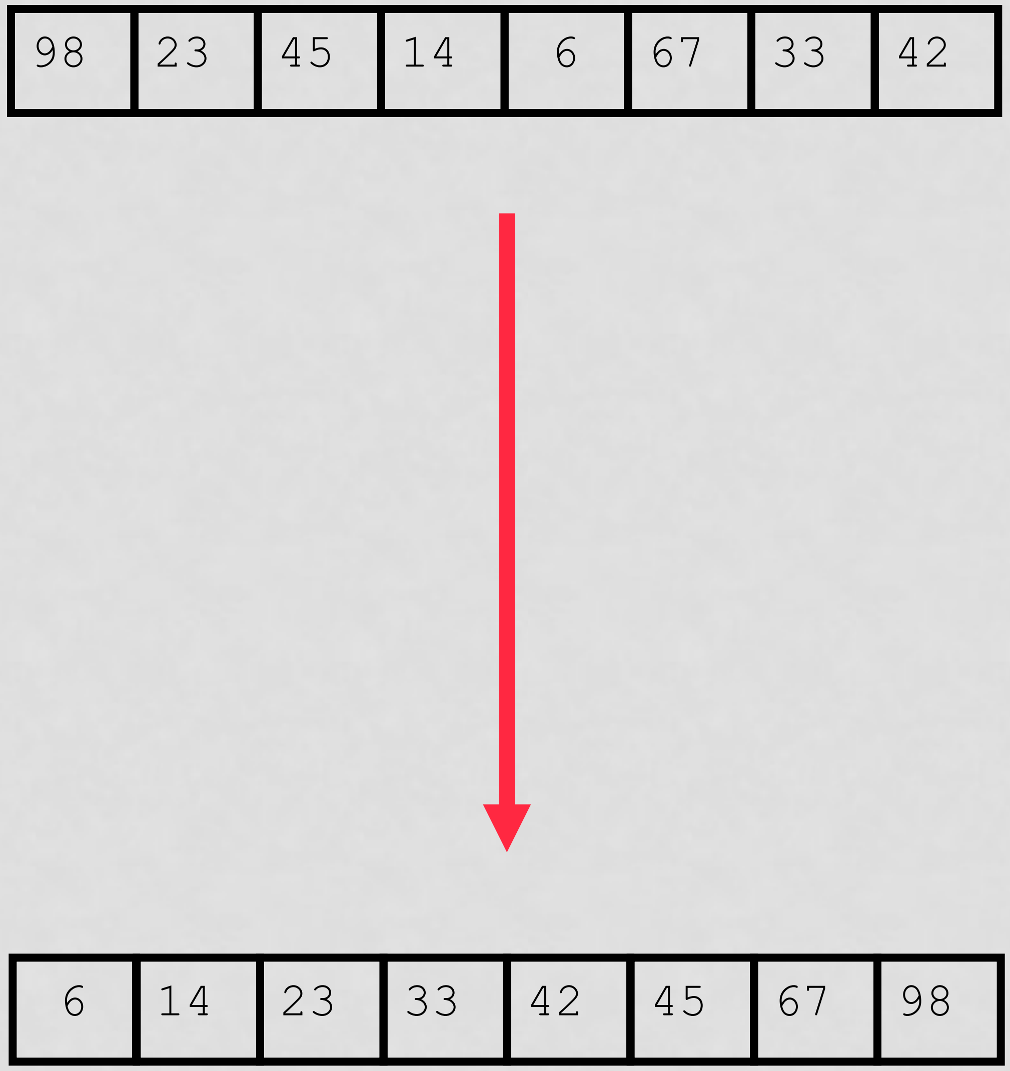

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42 45
Merge


98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42 45 67

Merge


98 23 45 14 6 67 33 42
98 23 45 14 | 6 67 33 42
98 23 | 45 14 | 6 67 | 33 42
98 | 23 | 45 | 14 | 6 | 67 | 33 | 42
23 98 | 14 45 | 6 67 | 33 42
14 23 45 98 | 6 33 42 67
6 14 23 33 42 45 67 98
Merge


98 23 45 14 6 67 33 42
98 23 45 14 | 6 67 33 42
98 23 | 45 14 | 6 67 | 33 42
98 | 23 | 45 | 14 | 6 | 67 | 33 | 42
23 98 | 14 45 | 6 67 | 33 42
14 23 45 98 | 6 33 42 67
6 14 23 33 42 45 67 98


98
23
45
14
6
67
33
42
6
14
23
33
42
45
67
98

## RECAP

- Divide the unsorted collection into two
- Until the sub-arrays only contain one element
- Then merge the sub-problem solutions together

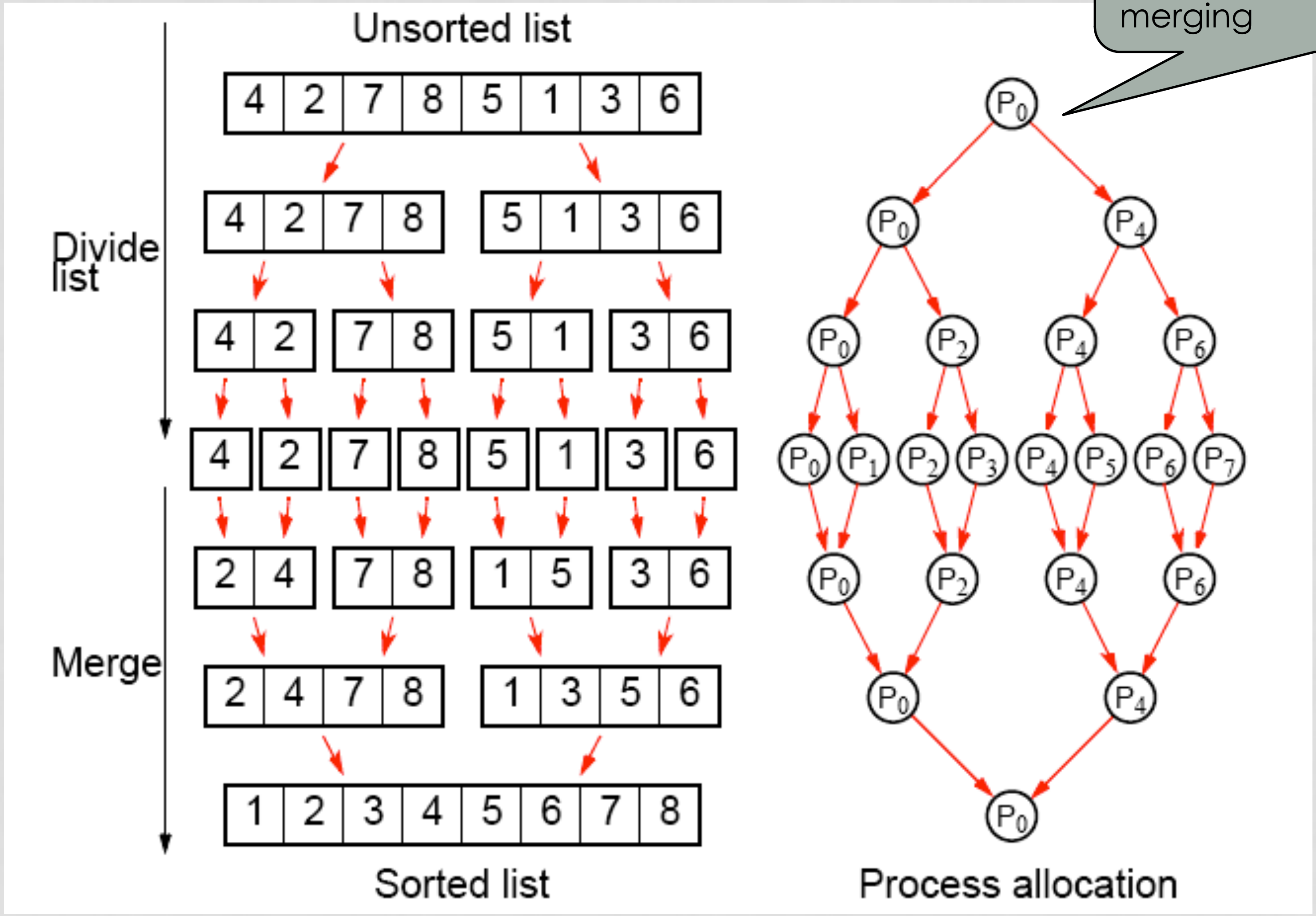
## HOW TO MAKE PARALLELIZED?

**Many-Processor MergeSort**

# Parallelizing Mergesort

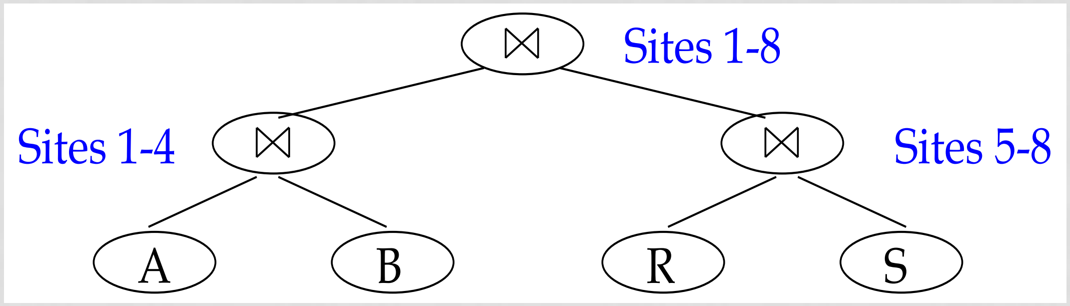
Using tree allocation of processes

Parent processor who makes the division, also makes the merging

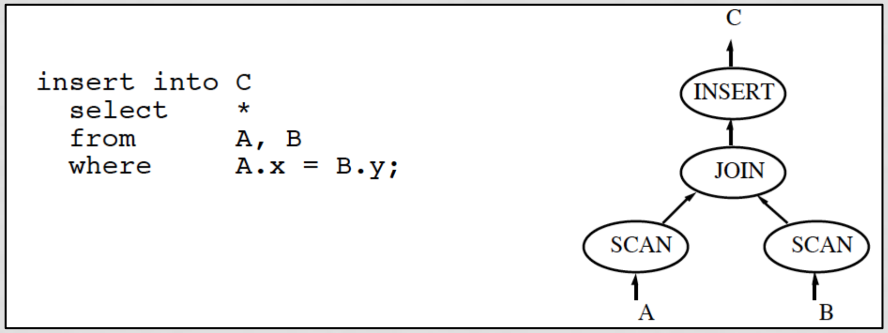


## COMPLEX PARALLEL QUERY PLANS

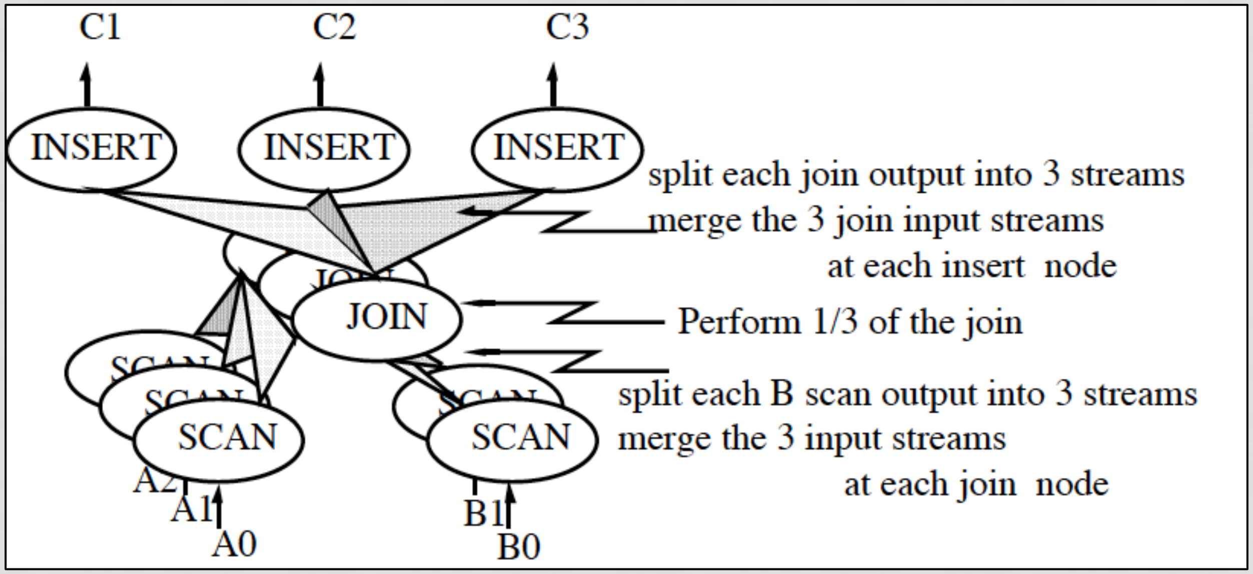
- All previous examples are ***intra-operator parallelism***
- **Complex queries can have *inter-operator parallelism***
  - Different machines perform different tasks



# EXAMPLE



```
insert into C
  select     *
  from       A, B
  where      A.x = B.y;
```



split each join output into 3 streams
merge the 3 join input streams
at each insert node

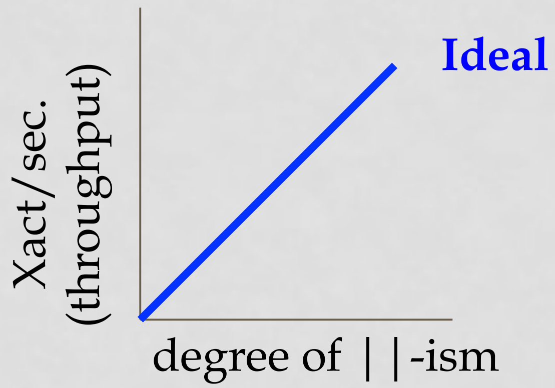
Perform 1/3 of the join

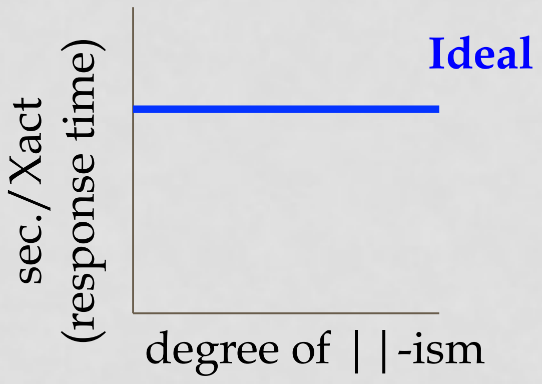
split each B scan output into 3 streams
merge the 3 input streams
at each join node

# PERFORMANCE OF PARALLEL ALGORITHMS

- **In many cases, parallel algorithms reach their expected lower bound (or close to)**
  - If parallelism degree is m, then the parallel cost is 1/m of the sequential cost
  - Cost mostly refers to query's response time

- **Example**
  - Parallel selection or projection is 1/m of the sequential cost





## PERFORMANCE OF PARALLEL ALGORITHMS (CONT'D)

- **Total disk I/Os (sum over all machines) of parallel algorithms can be larger than that of sequential counterpart**
    - But we get the benefit of being done in parallel

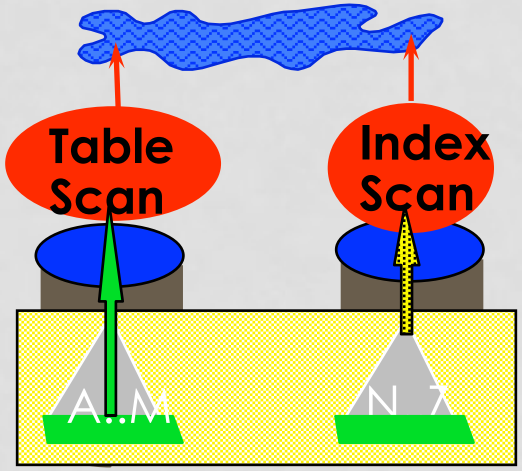
- **Example**
    - Merge-sort join **(serial case)** has I/O cost = 3(B(R) + B(S))
    - Merge-sort join **(parallel case)** has total (sum) I/O cost = 5(B(R) + B(S))
        - **Considering the parallelism = 5(B(R) + B(S)) / m**

**Number of pages of relations R and S**

# OPTIMIZING PARALLEL ALGORITHMS

- **Best serial plan != the best parallel one**

- **Trivial counter-example:**
  - Table partitioned with local secondary index at two nodes
  - **Range query:** all data of node 1 and 1% of node 2.
  - Node 1 should do a scan of its partition.
  - Node 2 should use secondary index.



- Different optimization algorithms for parallel plans (more candidate plans)

- Different machines may perform the same operation but using different plans

## SUMMARY OF PARALLEL DATABASES

- **Three possible architectures**
  - Shared-memory
  - Shared-disk
  - Shared-nothing (the most common one)

- **Parallel algorithms**
  - Intra-operator
    - Scans, projections, joins, sorting, set operators, etc.
  - Inter-operator
    - Distributing different operators in a complex query to different nodes

- **Partitioning and data layout is important and affect the performance**

- **Optimization of parallel algorithms is a challenge**