Divide and Conquer

Kumkum Saxena

Algorithmic Paradigms

- Greedy. Build up a solution incrementally, myopically optimizing some local criterion.
- Divide-and-conquer. Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.
- Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

Divide and Conquer

- Introduction
- Binary Search
- Finding Minimum and Maximum
- Merge Sort
- Quick Sort
- Strassen's Matrix Multiplication
  - Analysis of all algorithms

Divide and Conquer

General idea:

Divide a problem into subprograms of the same kind; solve subprograms using the same approach, and combine partial solution (if necessary).

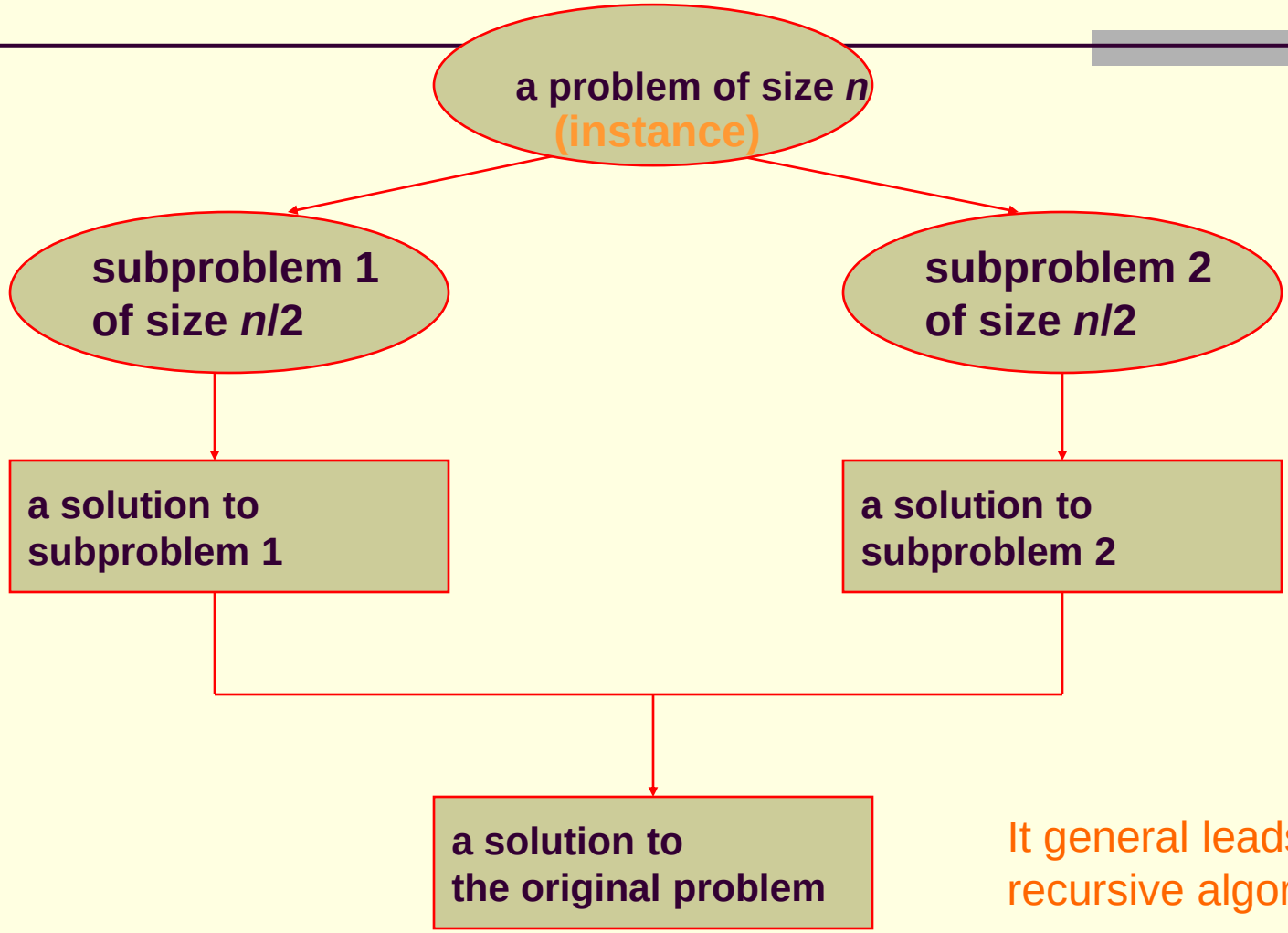
Divide and Conquer

- The most-well known algorithm design strategy:
- Divide instance of problem into two or more smaller instances
- Solve smaller instances recursively
- Obtain solution to original (larger) instance by combining these solutions

Divide and Conquer

- Divide and Conquer algorithms consist of two parts:
  - ***Divide:*** Smaller problems are solved recursively (except, of course, the base cases).
  - ***Conquer:*** The solution to the original problem is then formed from the solutions to the subproblems.

Divide-and-Conquer Technique



It general leads to a recursive algorithm!

Steps in Divide and Conquer

As its name implies ***divide-and-conquer*** involves dividing a problem into smaller problems that can be more easily solved. While the specifics vary from one application to another, ***divide-and-conquer*** always includes the following three steps in some form:

Divide - Typically this steps involves splitting one problem into two problems of approximately 1/2 the size of the original problem.

Conquer - The divide step is repeated (usually recursively) until individual problem sizes are small enough to be solved (conquered) directly.

Recombine - The solution to the original problem is obtained by combining all the solutions to the sub-problems.

Divide and Conquer is not applicable to every problem class. Even when D&C works it may not provide for an efficient solution.

Divide and Conquer

- **Traditionally**
  - Algorithms which contain at least 2 recursive calls are called *divide and conquer* algorithms, while algorithms with one recursive call are not.
- **Classic Examples**
  - Merge sort and Quick sort
  - The problem is divided into smaller sub-problems.
- **Examples of recursive algorithms that are not Divide and Conquer**
  - Even though the recursive method to compute the Fibonacci numbers has 2 recursive calls
  - It's really not divide and conquer because it doesn't divide the problem.

Divide and Conquer

- Given a function to compute on *n* inputs, the divide-and-conquer strategy consists of:
  - splitting the inputs into k distinct subsets, $1<k\leq n$, yielding k subproblems.
  - solving these subproblems
  - combining the sub solutions into solution of the whole.
- if the subproblems are relatively large, then divide_Conquer is applied again.
- if the subproblems are small, the are solved without splitting.

General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

Master Theorem: If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Note: The same results hold with O instead of Θ.

Examples: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$ Θ(n^2)

$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$ Θ(n^2log n)

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$ Θ(n^3)

General Divide-and-Conquer Recurrence

- **Examples:**
  - $T(n) = T(n/2) + n \Rightarrow T(n) \in \Theta(n)$

    Here a = 1, b = 2, d = 1, $a < b^d$

  - $T(n) = 2T(n/2) + 1 \Rightarrow T(n) \in \Theta(n^{\log_2 2}) = \Theta(n)$

    Here a = 2, b = 2, d = 0, $a > b^d$

Examples

- $T(n) = T(n/2) + 1 \Rightarrow T(n) \in \Theta(log(n))$

  Here a = 1, b = 2, d = 0, $a = b^d$

- $T(n) = 4T(n/2) + n \Rightarrow T(n) \in \Theta(n^{\log_2 4}) = \Theta(n^2)$

  Here a = 4, b = 2, d = 1, $a > b^d$

- $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in \Theta(n^2 \log n)$

  Here a = 4, b = 2, d = 2, $a = b^d$

- $T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in \Theta(n^3)$

  Here a = 4, b = 2, d = 3, $a < b^d$

Binary Search

Binary Search

- Number Guessing Game from childhood
  - Remember the game you most likely played as a child
    - I have a secret number between 1 and 100.
    - Make a guess and I'll tell you whether your guess is too high or too low.
    - Then you guess again. The process continues until you guess the correct number.
    - Your job is to MINIMIZE the number of guesses you make.

Binary Search

- Number Guessing Game from childhood
  - What is the first guess of most people?
    - 50.
  - Why?
    - No matter the response (too high or too low), the most number of possible values for your remaining search is 50 (either from 1-49 or 51-100)
    - Any other first guess results in the risk that the possible remaining values is greater than 50.
      - Example: you guess 75
      - I respond: too high
      - So now you have to guess between 1 and 74
        - 74 values to guess from instead of 50

Binary Search

- Number Guessing Game from childhood
  - Basic Winning Strategy
    - Always guess the number that is halfway between the lowest possible value in your search range and the highest possible value in your search range
- Can we now adapt this idea to work for searching for a given value in an array?

Binary Search

- Array Search
  - We are given the following sorted array:

index	0	1	2	3	4	5	6	7	8
value	2	6	19	27	33	37	38	41	118

  - We are searching for the value, 19
  - So where is halfway between?
    - One guess would be to look at 2 and 118 and take their average (60).
    - But 60 isn't even in the list
    - And if we look at the number closest to 60
      - It is almost at the end of the array

Binary Search

- Array Search
  - We quickly realize that if we want to adapt the number guessing game strategy to searching an array, we MUST search in the middle INDEX of the array.
  - In this case:
    - The lowest index is 0
    - The highest index is 8
    - So the middle index is 4

Binary Search

- Array Search
  - Correct Strategy
    - We would ask, “is the number I am searching for, 19, greater or less than the number stored in index 4?
      - Index 4 stores 33
    - The answer would be “less than”
    - So we would modify our search range to in between index 0 and index 3
      - Note that index 4 is no longer in the search space
    - We then continue this process
      - The second index we’d look at is index 1, since (0+3)/2=1
      - Then we’d finally get to index 2, since (2+3)/2 = 2
      - And at index 2, we would find the value, 19, in the array

Binary Search

- Binary Search code:

```
int binsearch(int a[], int len, int value) {

        int low = 0, high = len-1;
        while (low <= high) {
                int mid = (low+high)/2;
                if (value < a[mid])
                        high = mid-1;
                else if (value > a[mid])
                        low = mid+1;
                else
                        return 1;
        }

        return 0;
}
```

Recursive Binary Search

- int rBinarySearch(int array[], int start_index, int end_index, int element)

```
{
        if (end_index >= start_index)
        {
        int middle = start_index + (end_index - start_index )/2;
        if (array[middle] == element)
                return middle;
        if (array[middle] > element)
            return rBinarySearch(array, start_index, middle-1, element);
         return rBinarySearch(array, middle+1, end_index, element)
        }
        return -1;}
```

Binary Search

- Binary Search code:
  - At the end of each array iteration, all we do is update either low or high
  - This modifies our search region
    - Essentially halving it

Binary Search

- Efficiency of Binary Search
  - Analysis:
    - Let's analyze how many comparisons (guesses) are necessary when running this algorithm on an array of n items

      First, let's try n = 100
      - After 1 guess, we have 50 items left,
      - After 2 guesses, we have 25 items left,
      - After 3 guesses, we have 12 items left,
      - After 4 guesses, we have 6 items left,
      - After 5 guesses, we have 3 items left,
      - After 6 guesses, we have 1 item left
      - After 7 guesses, we have 0 items left.

Binary Search

- Efficiency of Binary Search
  - Analysis:
    - Notes:
      - The reason for the last iteration is because the number of items left represent the number of other possible values to search
        - We need to reduce this to 0.
      - Also, when n is odd, such as when n=25
        - We search the middle element, # 13
        - There are 12 elements smaller than 13
        - And 12 elements bigger than 13
        - This is why the number of items is slightly less than ½ in those cases

Binary Search

- Efficiency of Binary Search
  - Analysis:
    - General case:
    - After 1 guess, we have n/2 items left
    - After 2 guesses, we have n/4 items left
    - After 3 guesses, we have n/8 items left
    - After 4 guesses, we have n/16 items left
    - …
    - After k guesses, we have $n/2^k$ items left

Binary Search

- Efficiency of Binary Search
  - Analysis:
    - General case:
    - So, after k guesses, we have $n/2^k$ items left
    - The question is:
      - How many k guesses do we need to make in order to find our answer?
      - Or until we have one and only one guess left to make?
    - So we want to get only 1 item left
    - If we can find the value that makes the above fraction equal to 1, then we know that in one more guess, we'll narrow down the item

Binary Search

- Efficiency of Binary Search
  - Analysis:
    - General case:
    - So, after k guesses, we have $n/2^k$ items left
      - Again, we want only 1 item left
      - So set this equal to 1 and solve for k

$$\frac{n}{2^k} = 1 \qquad n = 2^k \qquad k = \log_2 n$$

    - This means that a binary search roughly takes $\log_2 n$ comparisons when searching in a sorted array of n items

Binary Search

- Efficiency of Binary Search
  - Analysis:
    - Runs in logarithmic (log n) time
    - This is MUCH faster than searching linearly
    - Consider the following chart:

n	log n
8	3
1024	10
65536	16
1048576	20
33554432	25
1073741824	30

    - Basically, any log n algorithm is SUPER FAST.

Binary Search - Iterative Version

```
procedure bin1(n:integer,x:keytype,S:list,loc:index)is
  lo,hi,mid : index;
begin
  lo:=1; hi:=n; loc:=0;
  while lo<=hi and loc=0 loop
    mid:=(lo+hi)/2;
    if x=S(mid) then
      loc:=mid;
    elsif x<S(mid) then
      hi:=mid-1;
    else
      lo:=mid+1;
    end if;
  end loop;
end bin1;
```

x=42
lo=1 hi=9
mid=4 S(4)=19

lo=5 hi=9
mid=7 S(7)=45

lo=5 hi=6
mid=5 S(5)=24

lo=6 hi=6
mid=6 S(6)=39

lo=7 hi=6

	S
0	10
1	12
2	15
3	19
4	19
5	24
6	39
7	45
8	53
9	77

$T(n) = C + \log_2 n\, D \rightarrow \mathbf{O}(\log_2 n)$

Binary Search - Recursive Version

```
function location(lo,hi : index) return index is
begin
  if lo>hi then
    return 0;
  else
    mid:=(lo+hi)/2;
    if x=S(mid) then
      return mid;
    elsif x<S(mid) then
      return location(lo,mid-1);
    else
      return location(mid+1,hi);
    end if;
  end if;
end location;
```

```
x=42
lo=0 hi=9
mid=4 S(4)=19
```

```
lo=5 hi=9
mid=7 S(7)=45
```

```
lo=5 hi=6
mid=5 S(5)=24
```

```
lo=6 hi=6
mid=6 S(6)=39
```

```
lo=7 hi=6
return 0
```

	S
0	10
1	12
2	15
3	19
4	19
5	24
6	39
7	45
8	53
9	77

Analysis of Function *location( )*

$$T(n) = \begin{cases} C & n<=1 \\ T(n/2) + D & \text{otherwise} \end{cases}$$

We need a closed form expression for T(n) that does not contain a term involving T.

We assume that n>1 and note that we need to replace T(n/2) with an explicit function of n.

$T(n/2)=T(n/4) + D$

$T(n)=(T(n/4) + D) + D$

$T(n)=T(n/4) + 2D$

$T(n)=T(n/8) + 3D$

:

We can use the recurrence relation to express the term T(n/2) in another form that can, in turn be substituted back into our expression for T(n). Eventually we can write a general expression for the kth substitution.

$T(n)=T(n/2^k) + kD$

$T(n)=T(1) + (\log_2 n)D$

$T(n)=C + (\log_2 n)D$

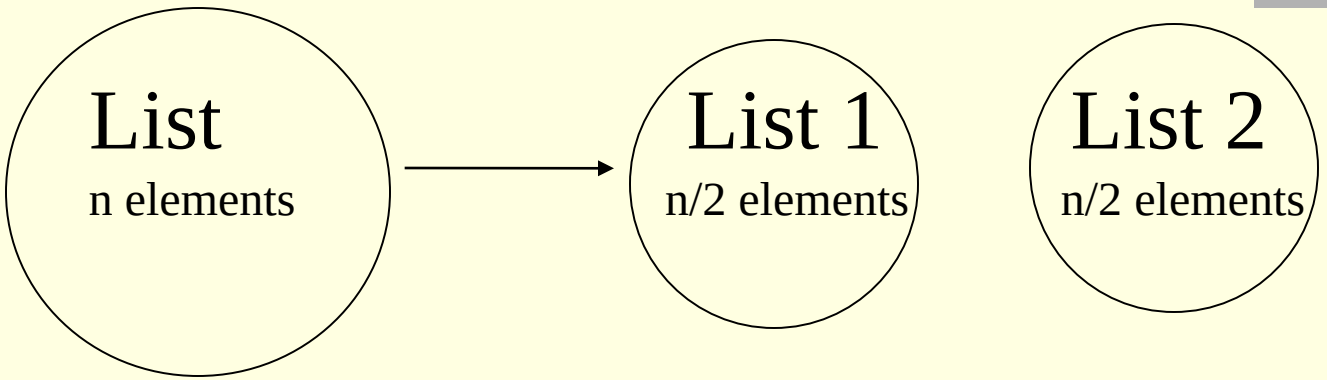
Then we can determine a value for k (in terms of n) that sets the parameter of T equal to one. This lets us substitute T(1)=C in our recurrence and replace any other occurrences of k with terms involving n.

$T(n) \rightarrow \mathbf{O}(\log_2 n)$

We can then determine the order of complexity.

Finding maximum and minimum

Finding maximum and minimum



min = MIN (min1, min2)
max = MAX (max1, max2)

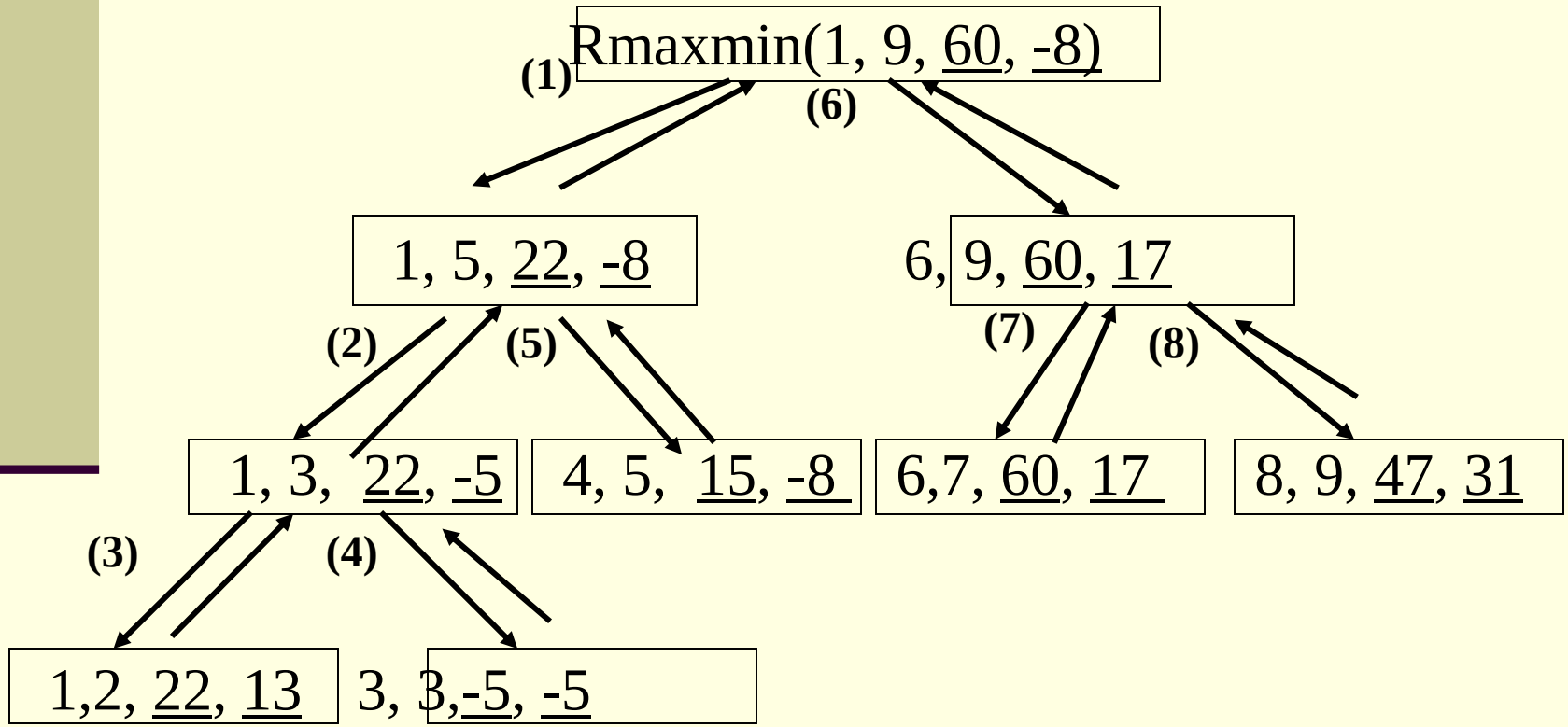
The Divide-and-Conquer algorithm:

```
procedure Rmaxmin (i, j, fmax, fmin);    // i, j are index #, fmax,
  begin                                  // fmin are output parameters
case:
  i = j:        fmax ← fmin ← A (i);
  i = j –1:     if A (i) < A (j) then
                        fmax ← A (j);
                        fmin ← A (i);
          else      fmax ← A (i);
                    fmin ← A (j);
  else:         mid ← (i+j)/2        ;
          call Rmaxmin (i, mid, gmax, gmin);
          call Rmaxmin (mid+1, j, hmax, hmin);
          fmax ← MAX (gmax, hmax);
          fmin ← MIN (gmin, hmin);
end
end;
```

Example: find max and min in the array:

22, 13, -5, -8, 15, 60, 17, 31, 47 (n = 9)

Index:	1	2	3	4	5	6	7	8	9
Array:	22	13	-5	-8	15	60	17	31	47



1. Find the maximum and minimum

The problem: Given a list of unordered *n* elements, find max and min

The straightforward algorithm:

max ← min ← A (1);
for *i* ← 2 to n do
    if A (*i*) > max, max ← A (*i*);
    if A (*i*) < min, min ← A (*i*);

Key comparisons: 2(n – 1)

Analysis: *For algorithm containing recursive calls, we can use recurrence relation to find its complexity*

T(n) - # of comparisons needed for Rmaxmin

Recurrence relation:

$$T(n) = 0 \quad n = 1$$

$$T(n) = 1 \quad n = 2$$

$$T(n) = 2T(\frac{n}{2}) + 2 \quad otherwise$$

$$T(n) = 2T(\frac{n}{2}) + 2$$

$$2(2T(\frac{n}{4}) + 2) + 2 = 2^2 T(\frac{n}{2^2}) + 2^2 + 2$$

$$2^2(2T(\frac{n}{8}) + 2) + 2^2 + 2 = 2^3 T(\frac{n}{2^3}) + 2^3 + 2^2 + 2$$

$\ldots$

Assume $n = 2^k$ for some integer k

$$2^{k-1}T\left(\frac{n}{2^{k-1}}\right)+\left(2^{k-1}+2^{k-2}+\cdots+2^1\right)$$

$$2^{k-1}T(2)+\left(2^k-2\right)=\frac{n}{2}\cdot 1+n-2$$

$$1.5n-2$$

Merge Sort

Kumkum Saxena

Merge Sort

- Problem with Bubble/Insertion/Selection Sorts:
  - All of these sorts make a large number of comparisons and swaps between elements
  - Any algorithm that swaps adjacent elements can only run so fast
  - So one might ask is there a more clever way to sort numbers
    - A way that does not require looking at all these pairs
  - Indeed, there are several ways to do this
  - And one of them is Merge Sort

- **Merge sort** is an *O*(*n* log *n*) comparison-based sorting algorithm.
- Most implementations produce a stable sort, meaning that the implementation preserves the input order of equal elements in the sorted output.
- It is a divide and conquer algorithm.

Sorting: Merge Sort

- Merge Sort
  - Conceptually, Merge Sort works as follows:
    - If the "list" is of length 0 or 1, then it is already sorted!
    - Otherwise:
    1. Divide the unsorted list into two sub-lists of about half the size
       - So if your list has n elements, you will divide that list into two sub-lists, each having approximately n/2 elements:
    2. Recursively sort each sub-list by calling recursively calling Merge Sort on the two smaller lists
    3. **Merge** the two sub-lists back into one sorted list
       - This Merge is a function that we study on its own
         - In a bit…

Sorting: Merge Sort

- Merge Sort
  - Basically, given a list:
    - You will split this list into two lists of about half the size
    - Then you recursively call Merge Sort on each list
    - What does that do?
      - Each of these new lists will, individually, be split into two lists of about half the size.
      - So now we have four lists, each about ¼ the size of the original list
    - This keeps happening…the lists keep getting split into smaller and smaller lists
      - Until you get to a list of size 1 or size 0…which is sorted!
    - Then we Merge them into a larger, sorted list

Sorting: Merge Sort

- Merge Sort
  - Incorporates two main ideas to improve its runtime:
  1) A small list will take fewer steps to sort than a large list
  2) Fewer steps are required to construct a sorted list from two sorted lists than two unsorted lists
     - For example:
       - You only have to traverse each list once if they're already sorted

Sorting: Merge Sort

- **Merge** function
  - The key to Merge Sort: the **Merge** function
  - Given two sorted lists, **Merge** them into one sorted list
  - Problem:
    - You are given two arrays, each of which is already sorted
    - Your job is to efficiently combine the two arrays into one larger array
    - The larger array should contain all the values of the two smaller arrays
    - Finally, the larger array should be in sorted order

Sorting: Merge Sort

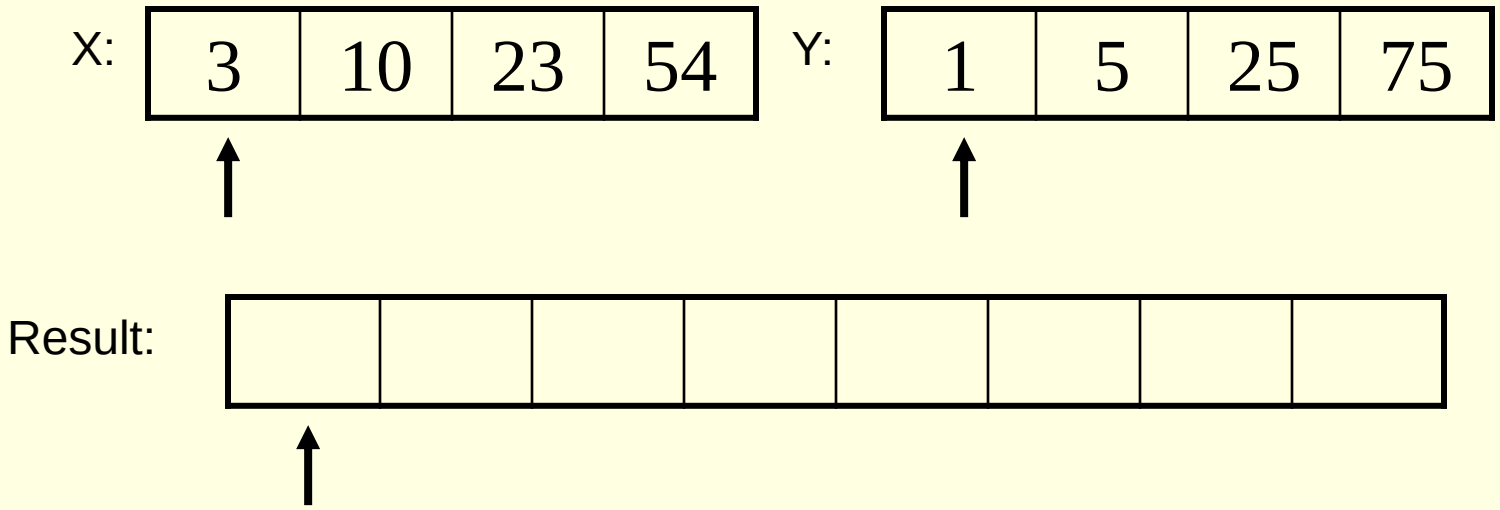
- **Merge** function
  - The key to Merge Sort: the **Merge** function
  - Given two sorted lists, **Merge** them into one sorted list
  - If you have two lists:
    - X ($x_1 < x_2 < \ldots < x_m$) and Y ($y_1 < y_2 < \ldots < y_n$)
    - Merge these into one list: Z ($z_1 < z_2 < \ldots < z_{m+n}$)
  - Example:
    - List 1 = {3, 8, 9} and List 2 = {1, 5, 7}
    - Merge(List 1, List 2) = {1, 3, 5, 7, 8, 9}

Sorting: Merge Sort

- **Merge** function
  - Solution:
    - Keep track of the smallest value in each array that hasn't been placed, in order, in the larger array yet
    - Compare these two smallest values from each array
      - One of these MUST be the smallest of all the values in both arrays that are left
      - Place the smallest of the two values in the next location in the larger array
    - Adjust the smallest value for the appropriate array
    - Continue this process until all values have been placed in the large array

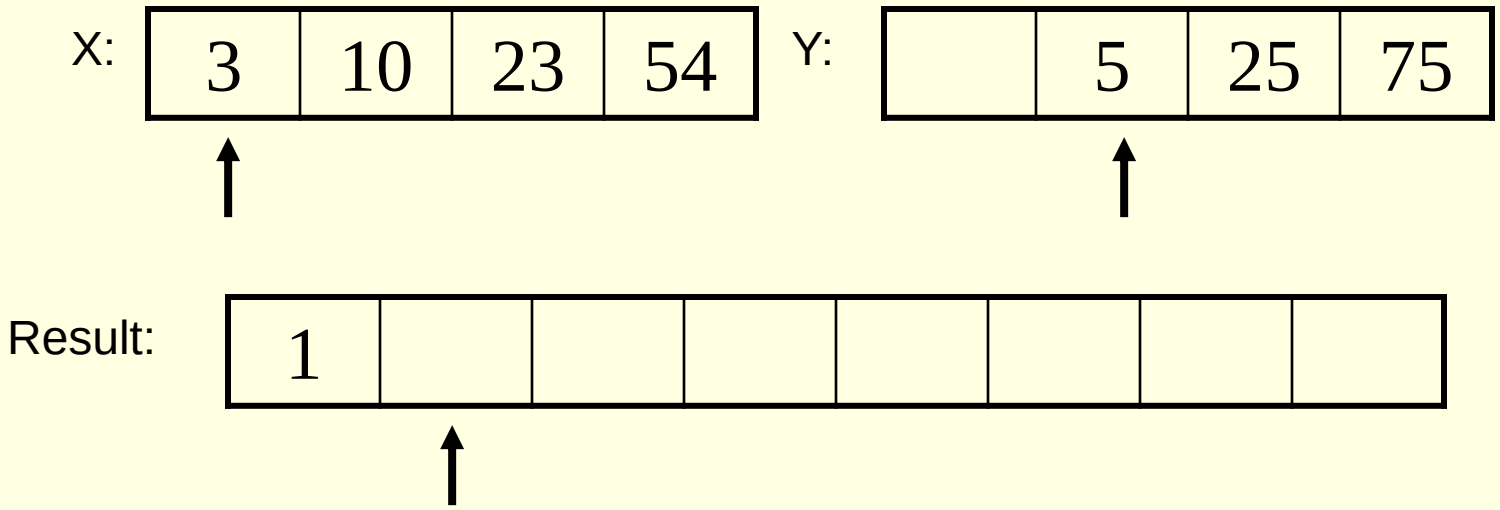
Sorting: Merge Sort

- Example of **Merge** function:



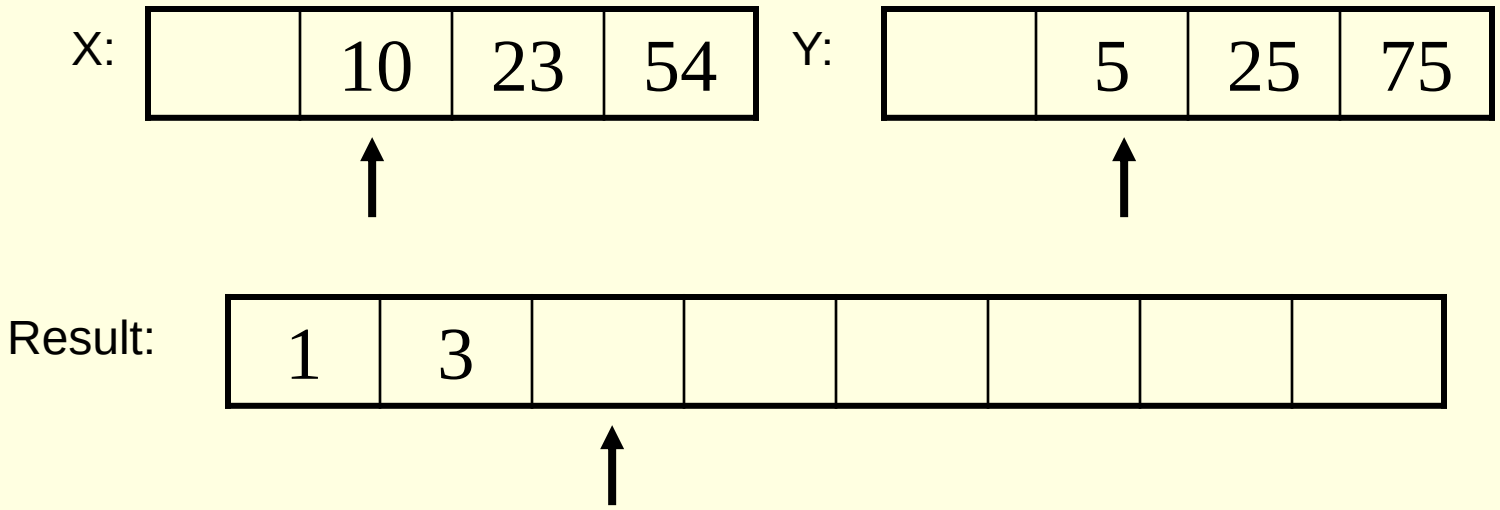
Sorting: Merge Sort

- Example of **Merge** function:



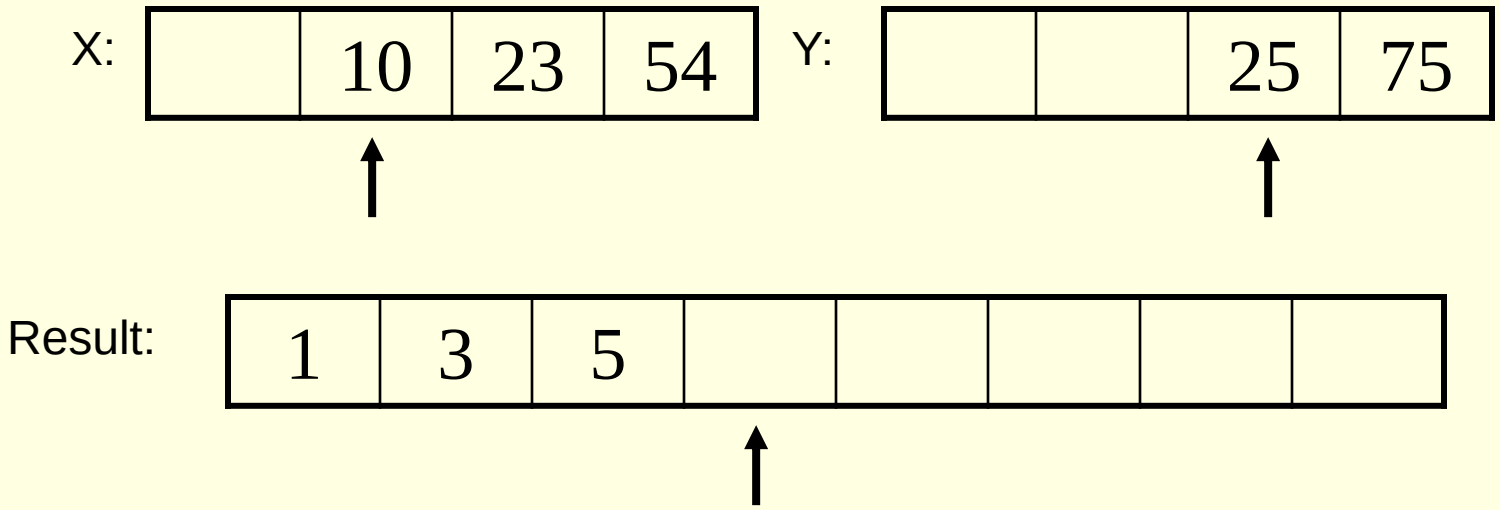
Sorting: Merge Sort

- Example of **Merge** function:



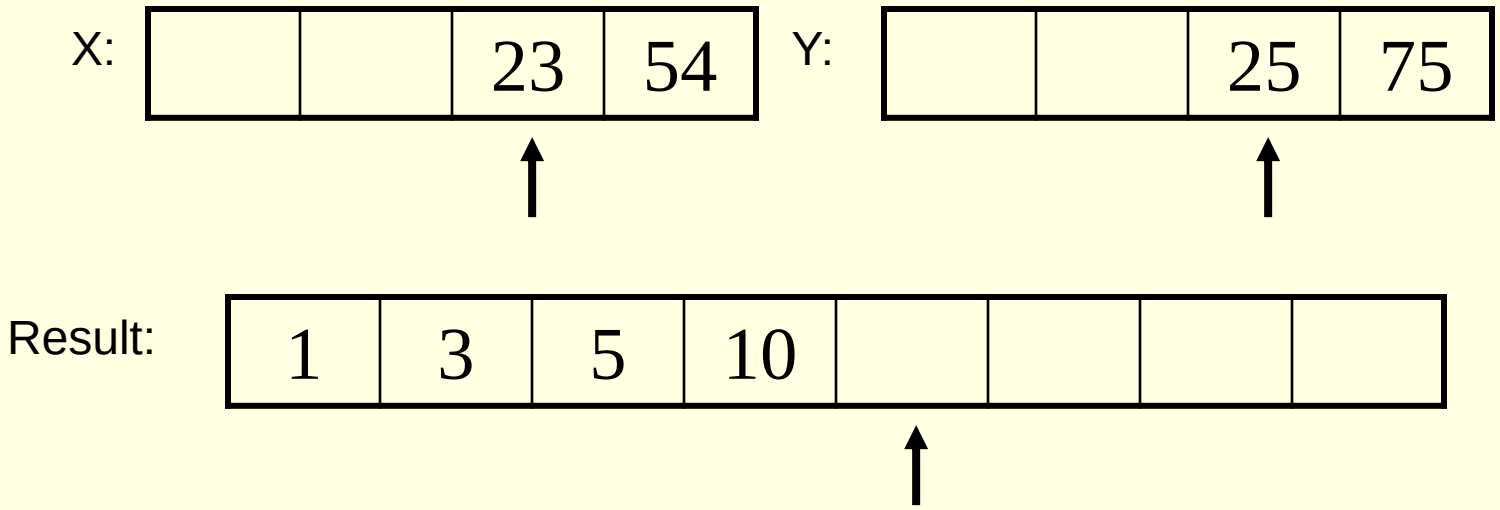
Sorting: Merge Sort

- Example of **Merge** function:



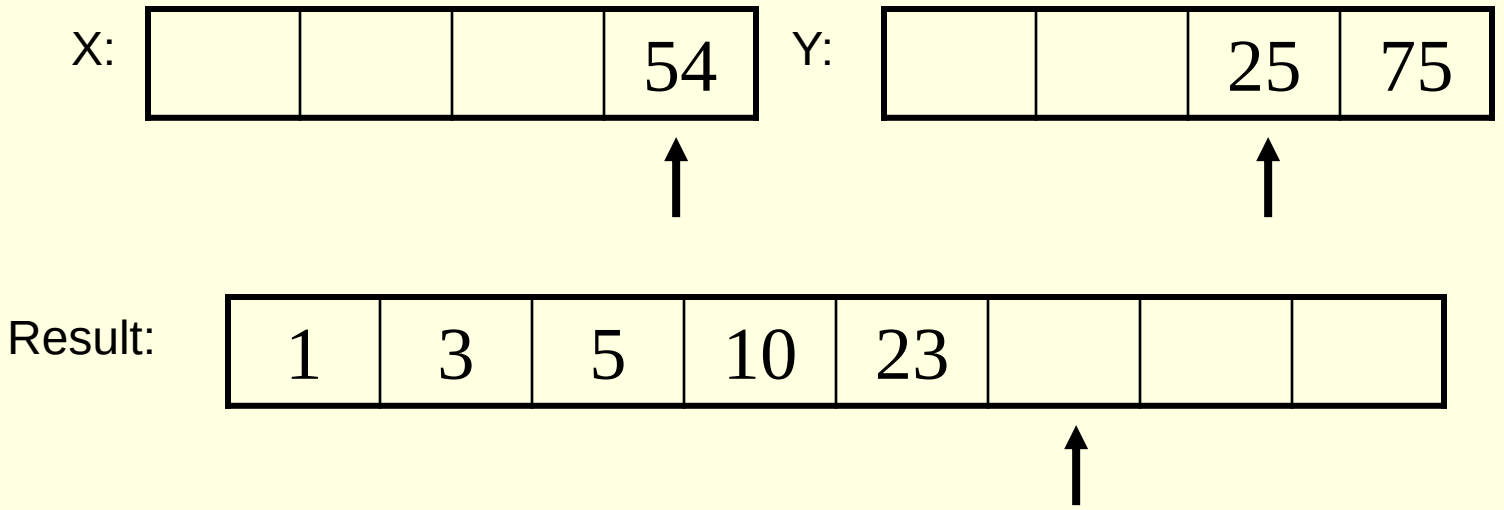
Sorting: Merge Sort

- Example of **Merge** function:



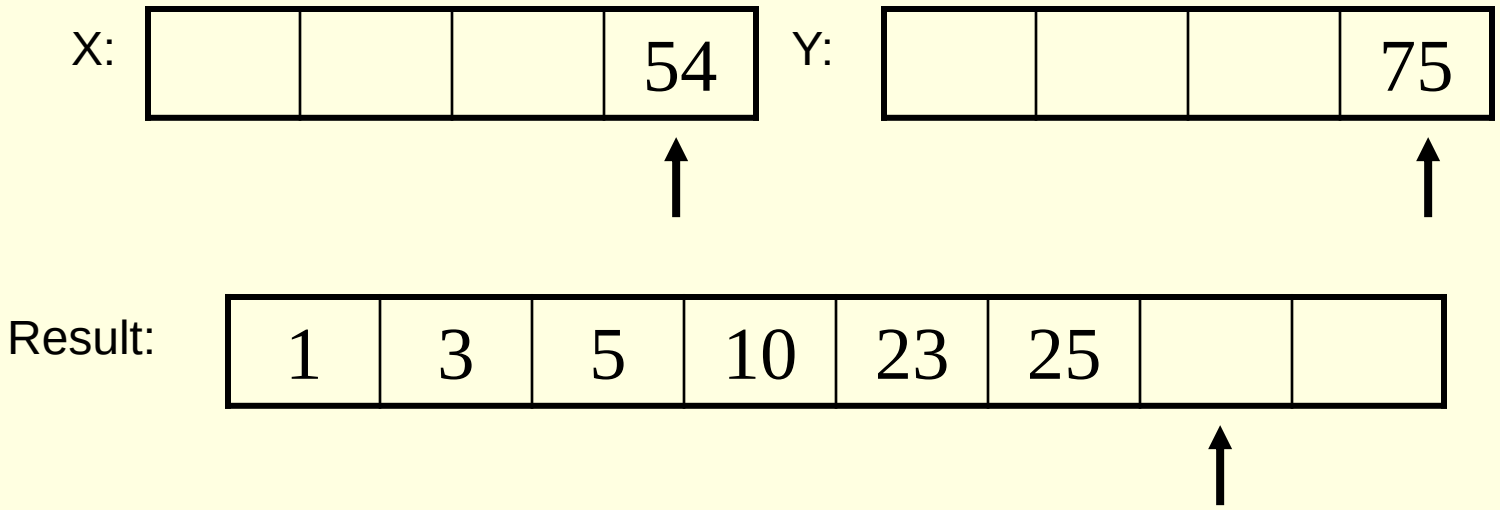
Sorting: Merge Sort

- Example of **Merge** function:



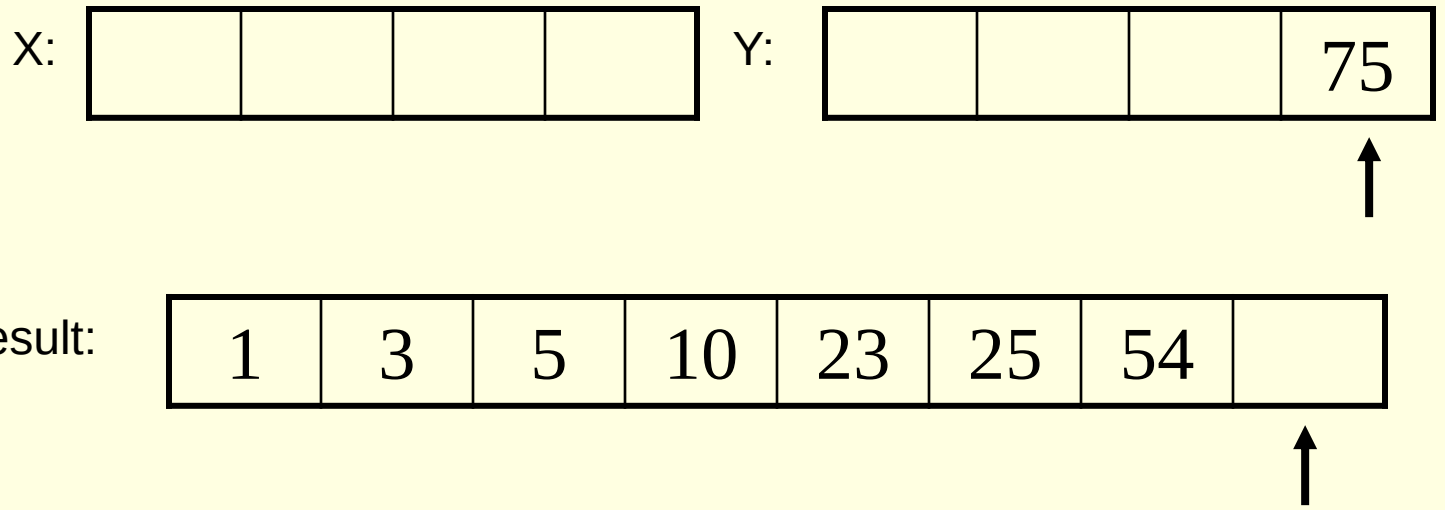
Sorting: Merge Sort

- Example of **Merge** function:



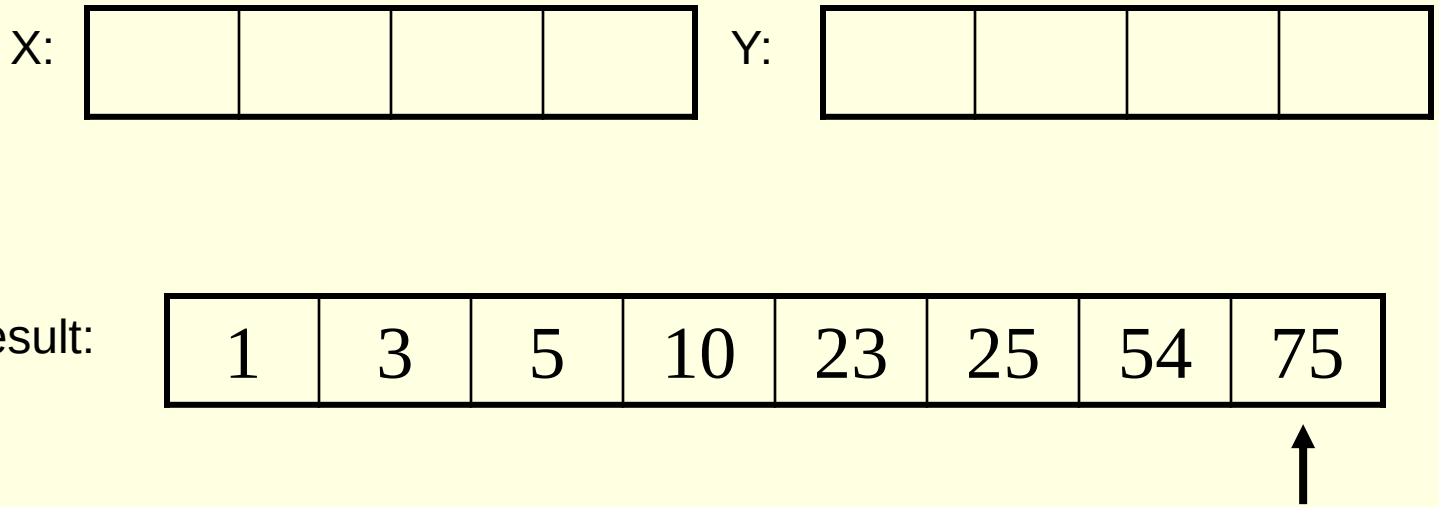
Sorting: Merge Sort

- Example of **Merge** function:



Sorting: Merge Sort

- Example of **Merge** function:



Sorting: Merge Sort

- **Merge** function
  - The big question:
    - How can we use this Merge function to sort an entire, unsorted array?
    - This function only "sorts" a specific scenario:
      - You have to have two, **already sorted**, arrays
    - **Merge** can then "sort" (merge) them into one larger array
    - So can we use this Merge function to somehow sort a large, unsorted array???
  - This brings us back to Merge Sort

Sorting: Merge Sort

- Merge Sort
  - Again, here is the main idea for Merge Sort:
  1) Sort the first half of the array, using Merge Sort
  2) Sort the second half of the array, using Merge Sort
     - Now, we do indeed have a situation where we can use the Merge function!
     - Each half is already sorted!
  3) So simply merge the first half of the array with the second half.
  - And this points to a recursive solution…

Sorting: Merge Sort

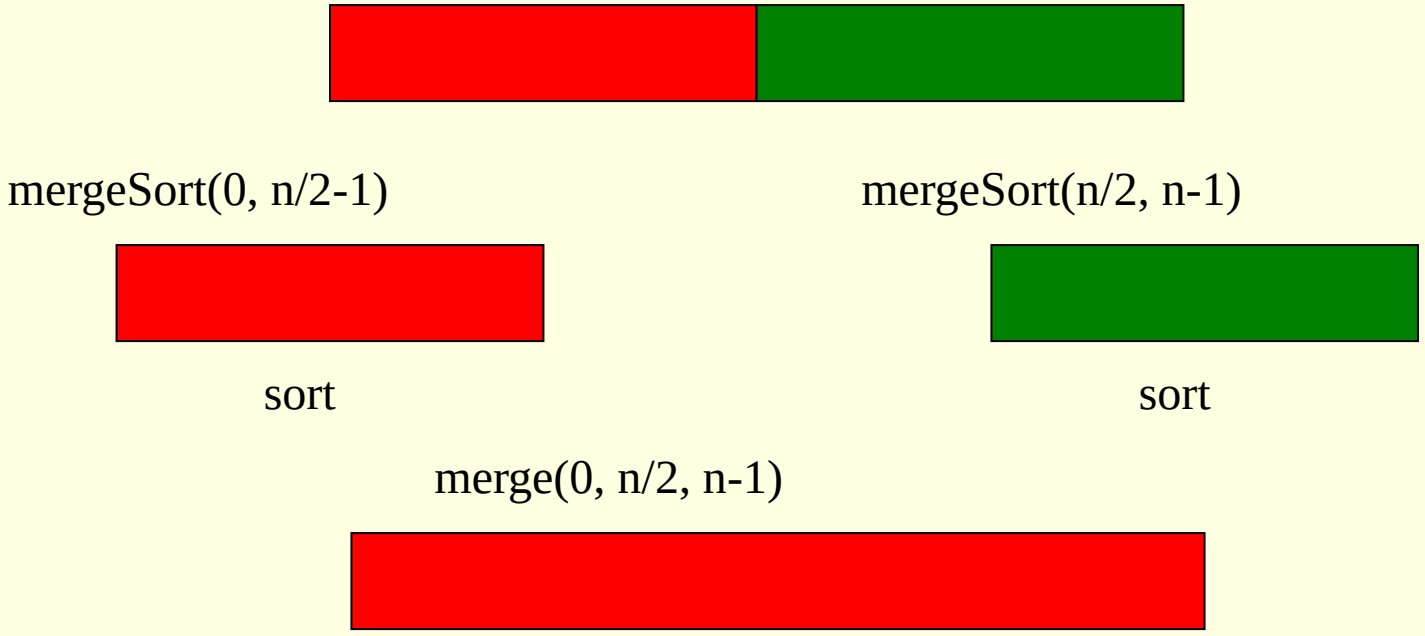
- Merge Sort
  - Conceptually, Merge Sort works as follows:
    - If the “list” is of length 0 or 1, then it is already sorted!
    - Otherwise:
    1. Divide the unsorted list into two sub-lists of about half the size
       - So if your list has n elements, you will divide that list into two sub-lists, each having approximately n/2 elements:
    2. Recursively sort each sub-list by calling recursively calling Merge Sort on the two smaller lists
    3. **Merge** the two sub-lists back into one sorted list

Sorting: Merge Sort

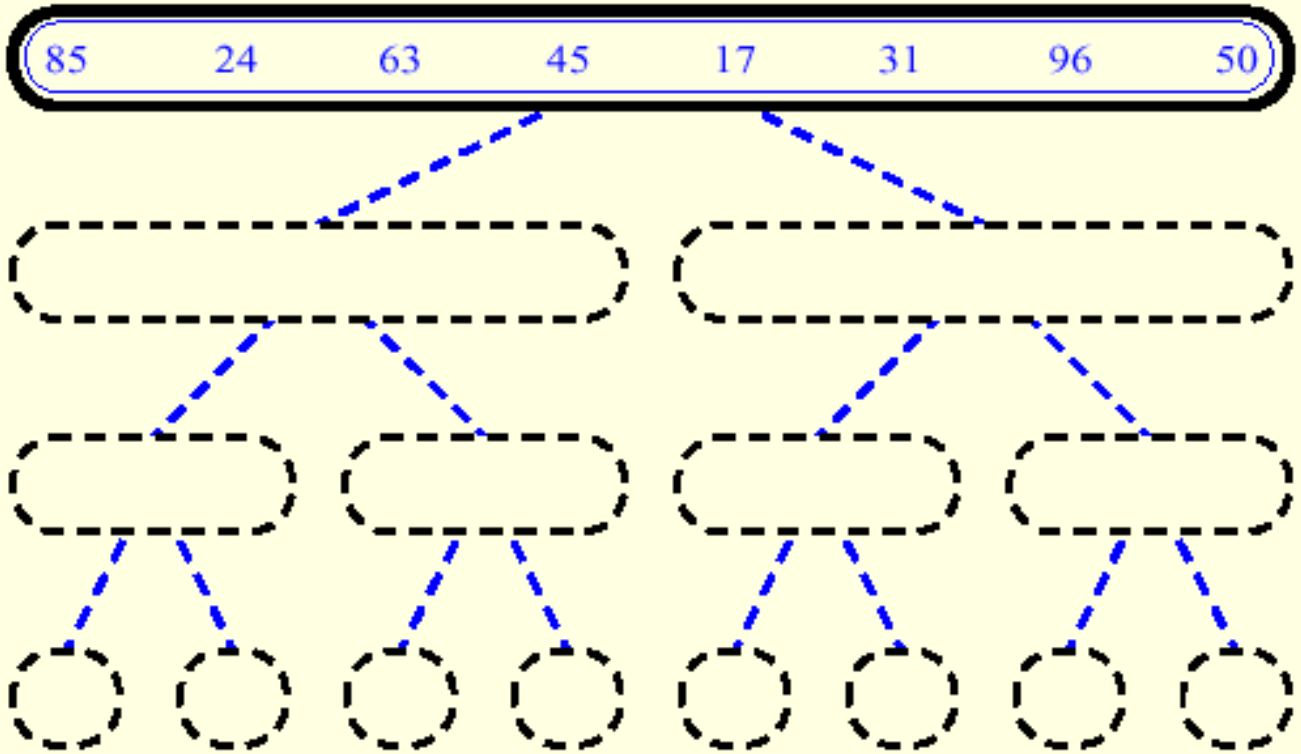
- Merge Sort
  - Basically, given a list:
    - You will split this list into two lists of about half the size
    - Then you recursively call Merge Sort on each list
    - What does that do?
      - Each of these new lists will, individually, be split into two lists of about half the size.
      - So now we have four lists, each about ¼ the size of the original list
    - This keeps happening…the lists keep getting split into smaller and smaller lists
      - Until you get to a list of size 1 or size 0
    - Then we Merge them into a larger, sorted list

Sorting: Merge Sort

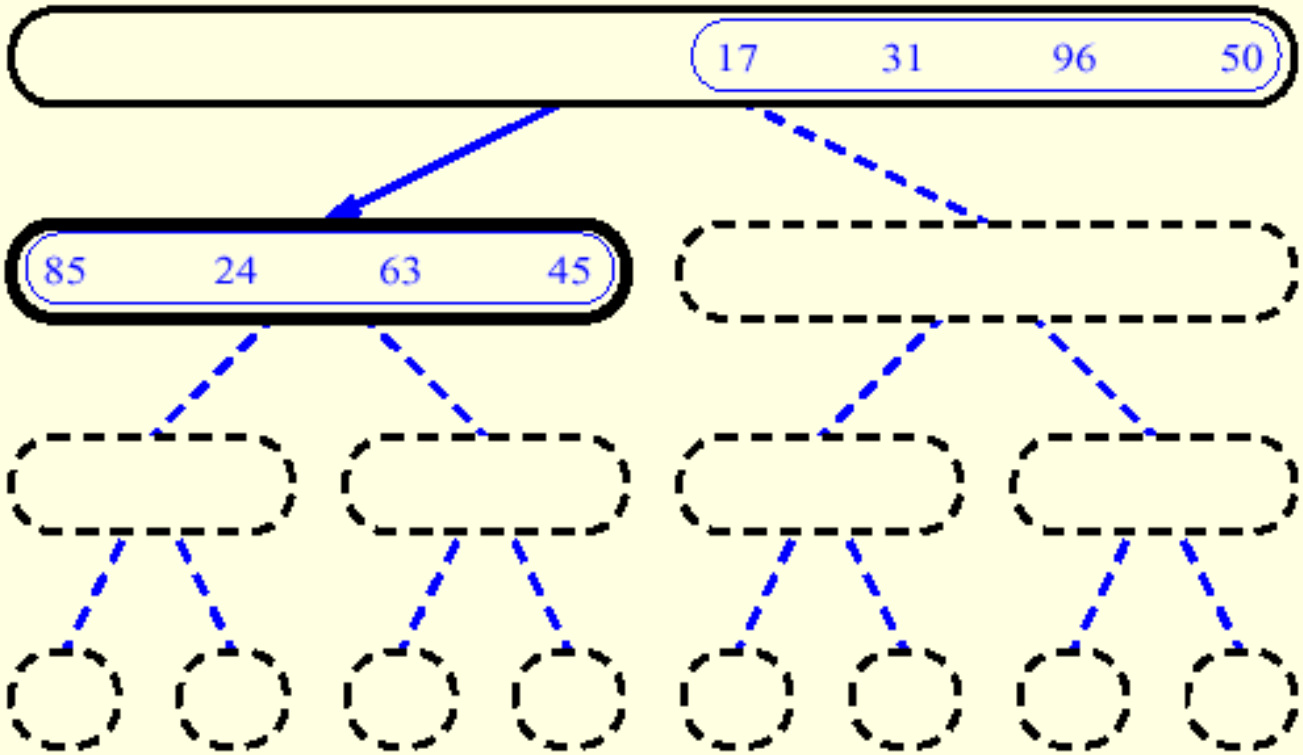
- Merge sort idea:
  - Divide the array into two halves.
  - Recursively sort the two halves (using merge sort).
  - Use **Merge** to combine the two arrays.



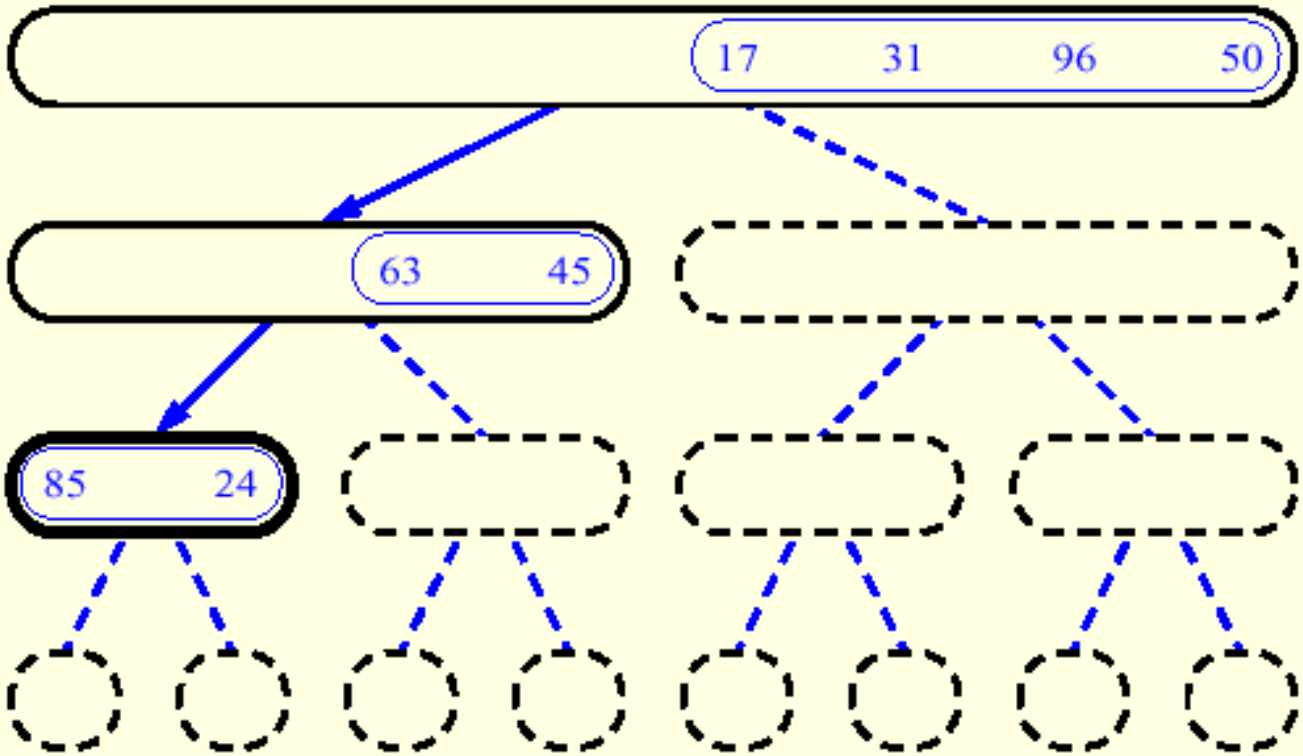
MergeSort (Example) - 1



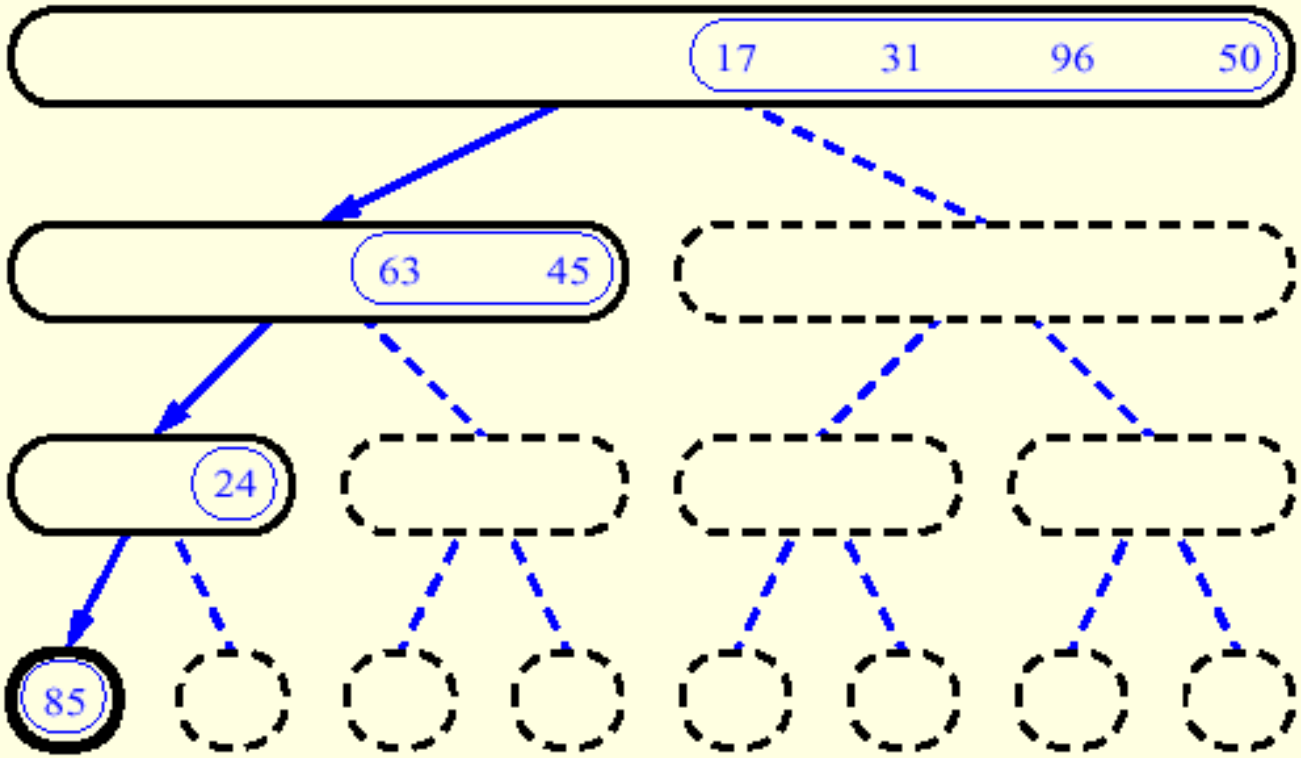
MergeSort (Example) - 2



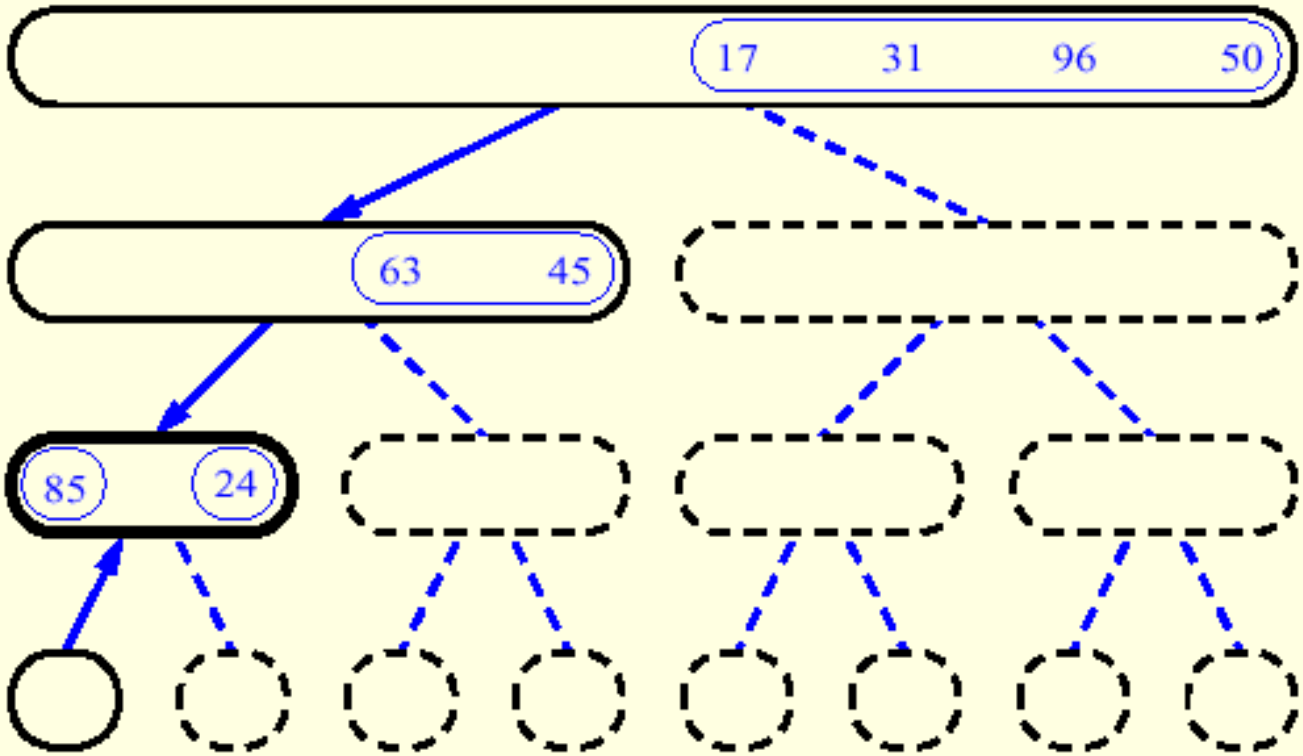
MergeSort (Example) - 3



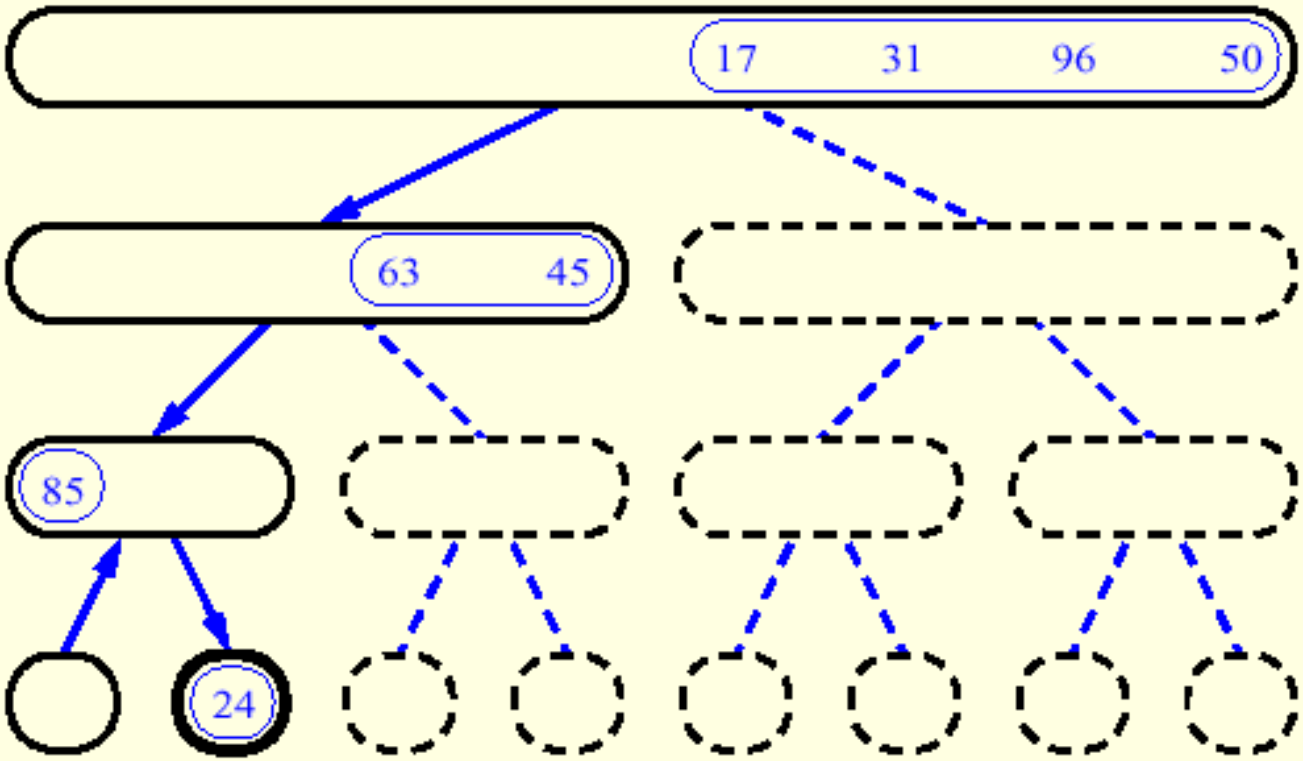
MergeSort (Example) - 4



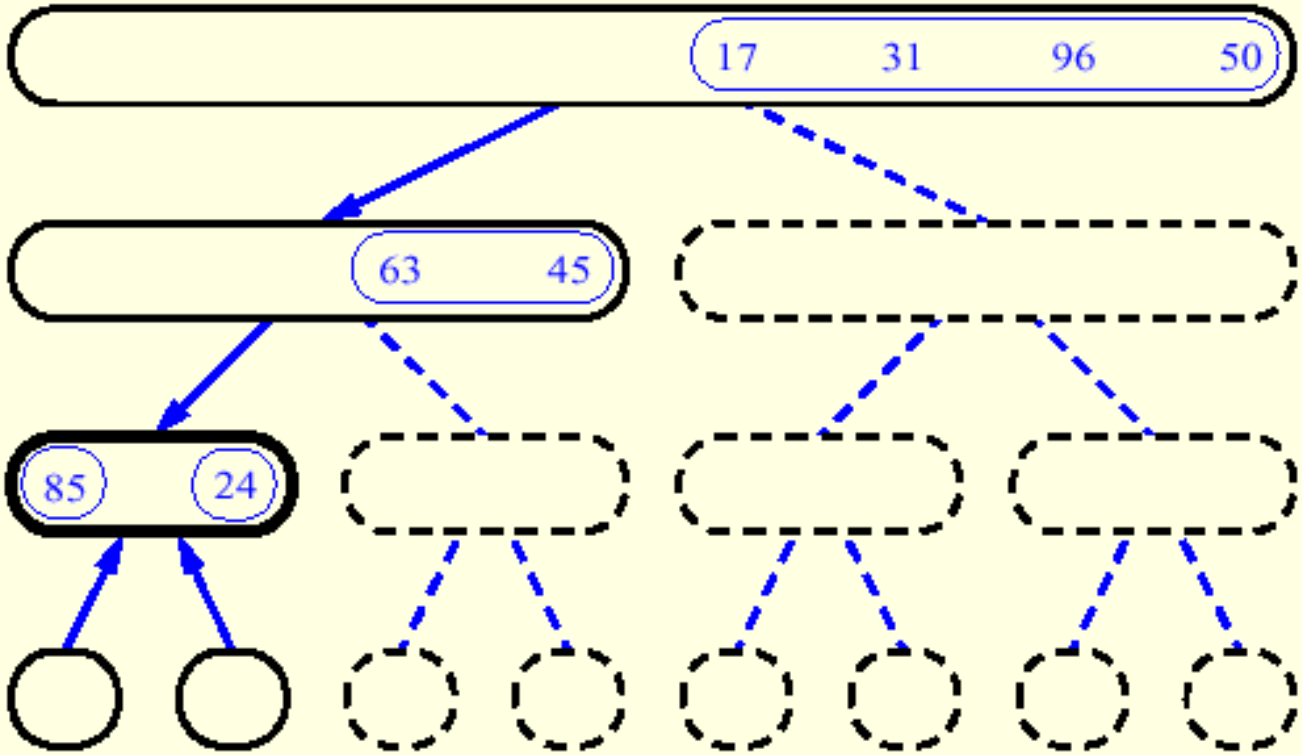
MergeSort (Example) - 5



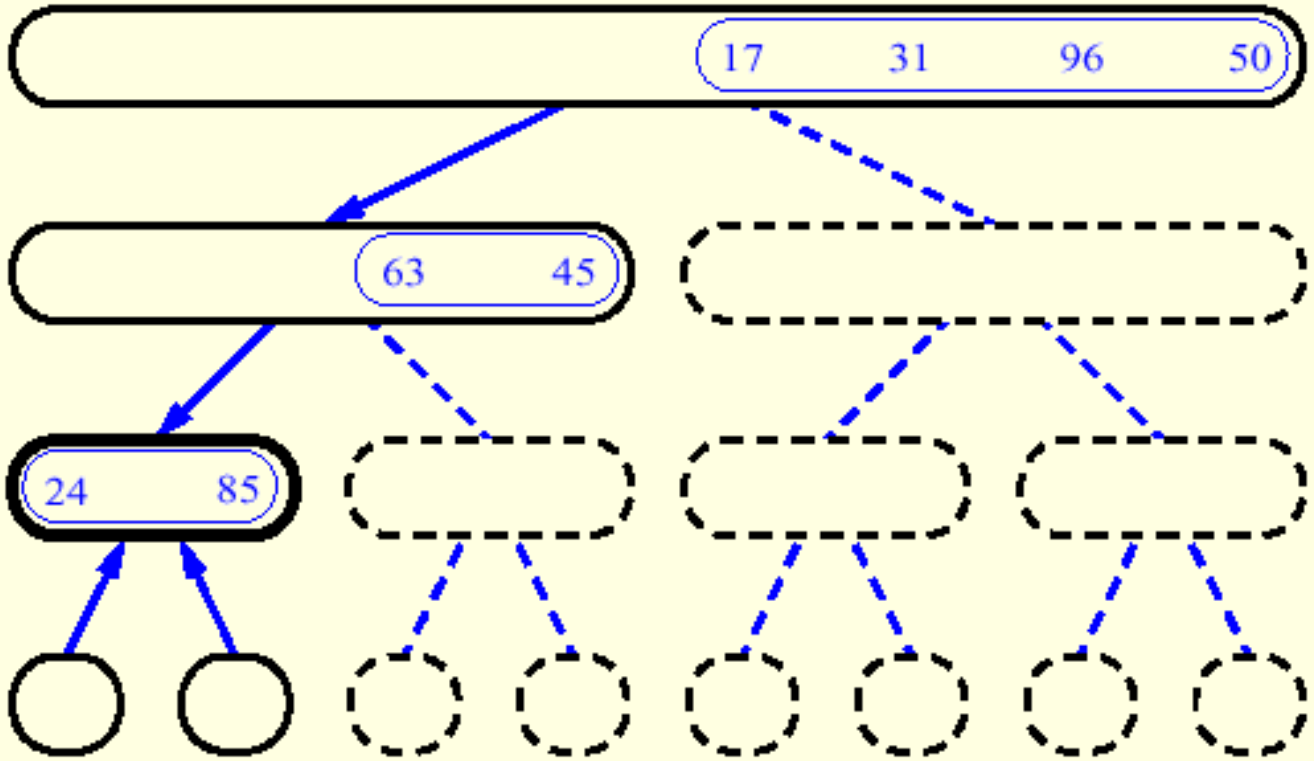
MergeSort (Example) - 6



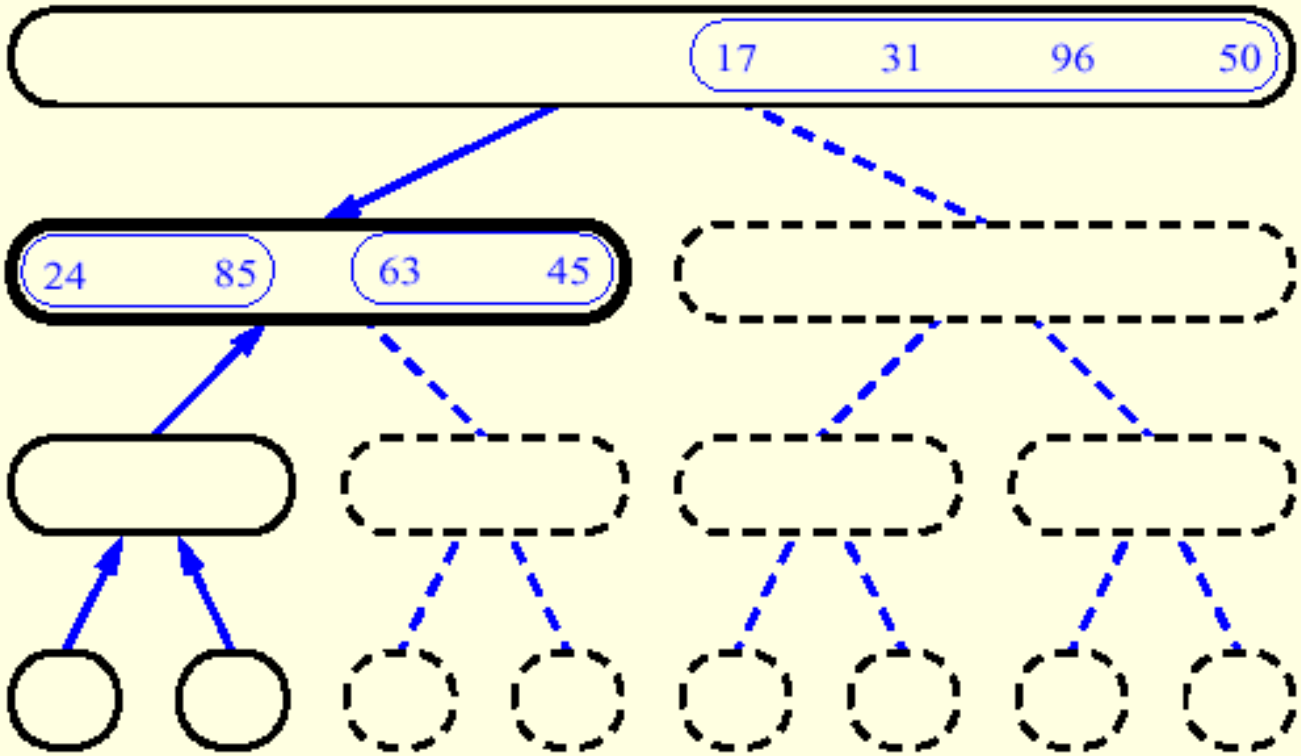
MergeSort (Example) - 7



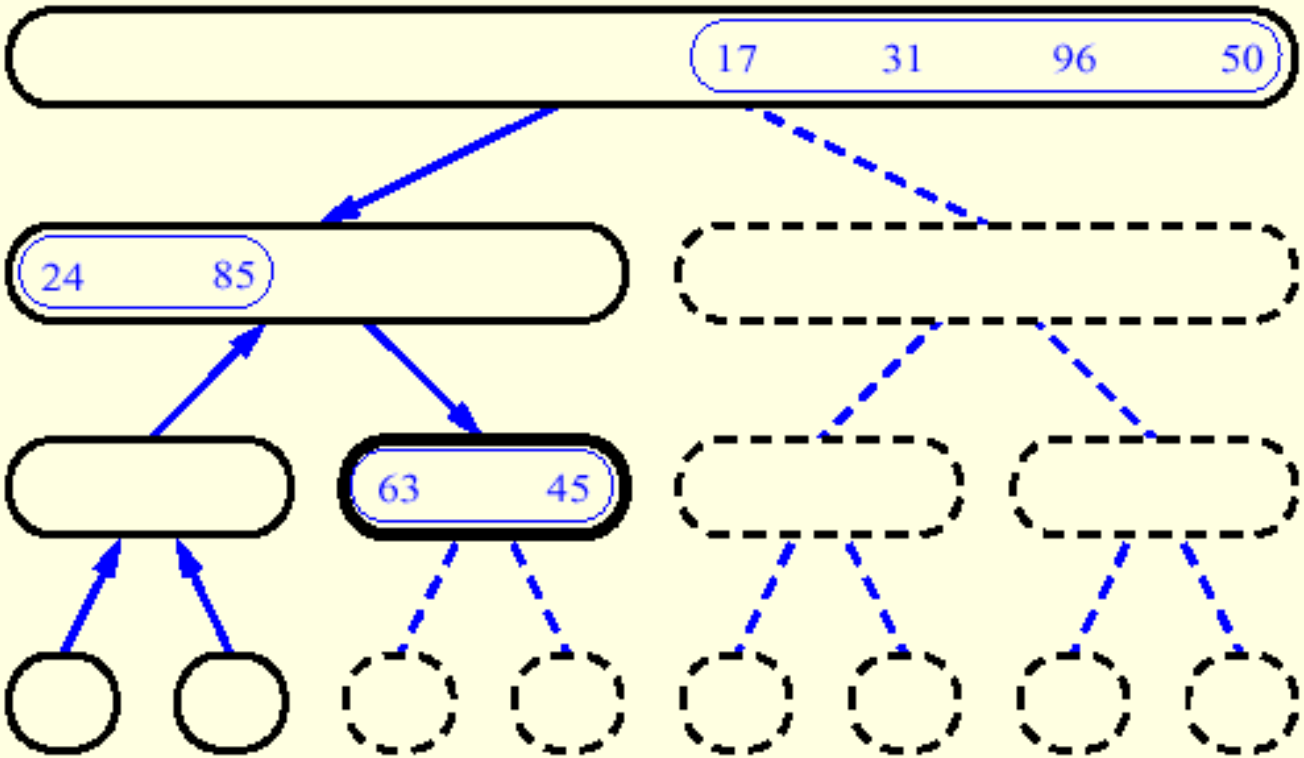
MergeSort (Example) - 8



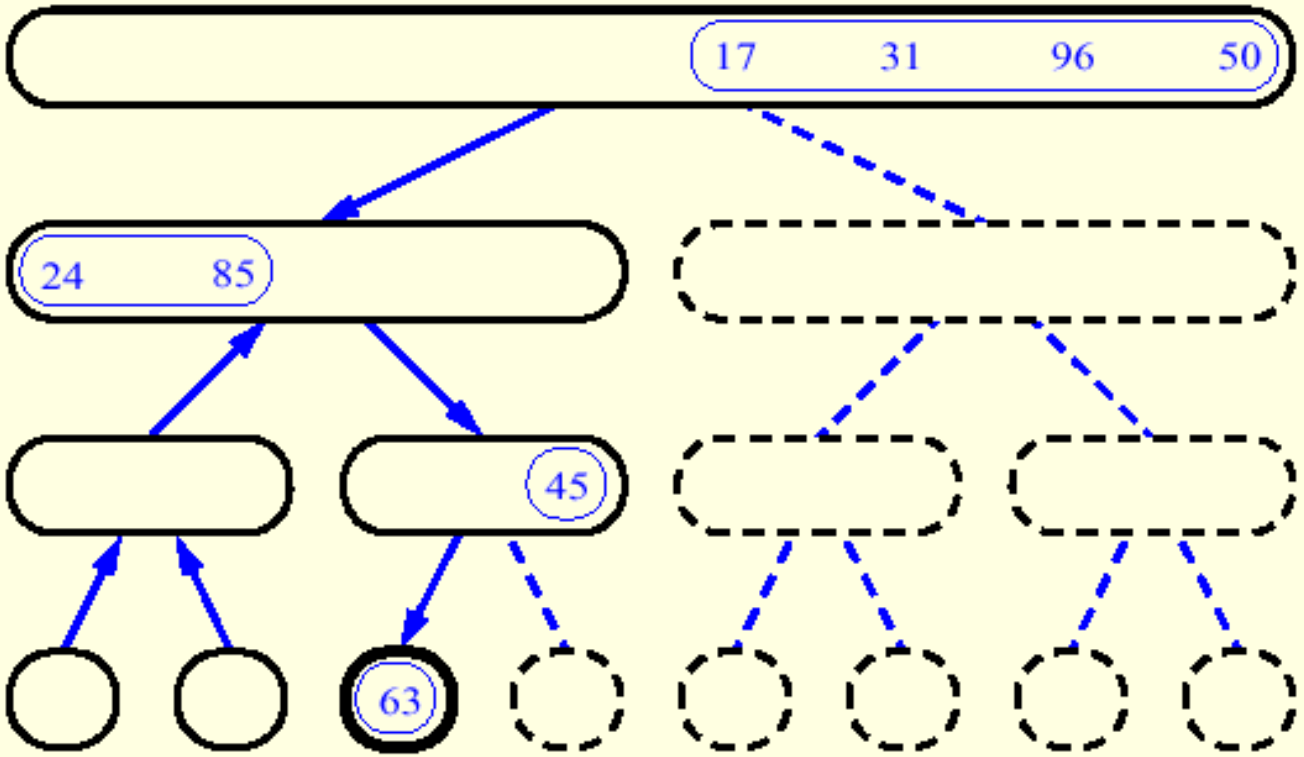
MergeSort (Example) - 9



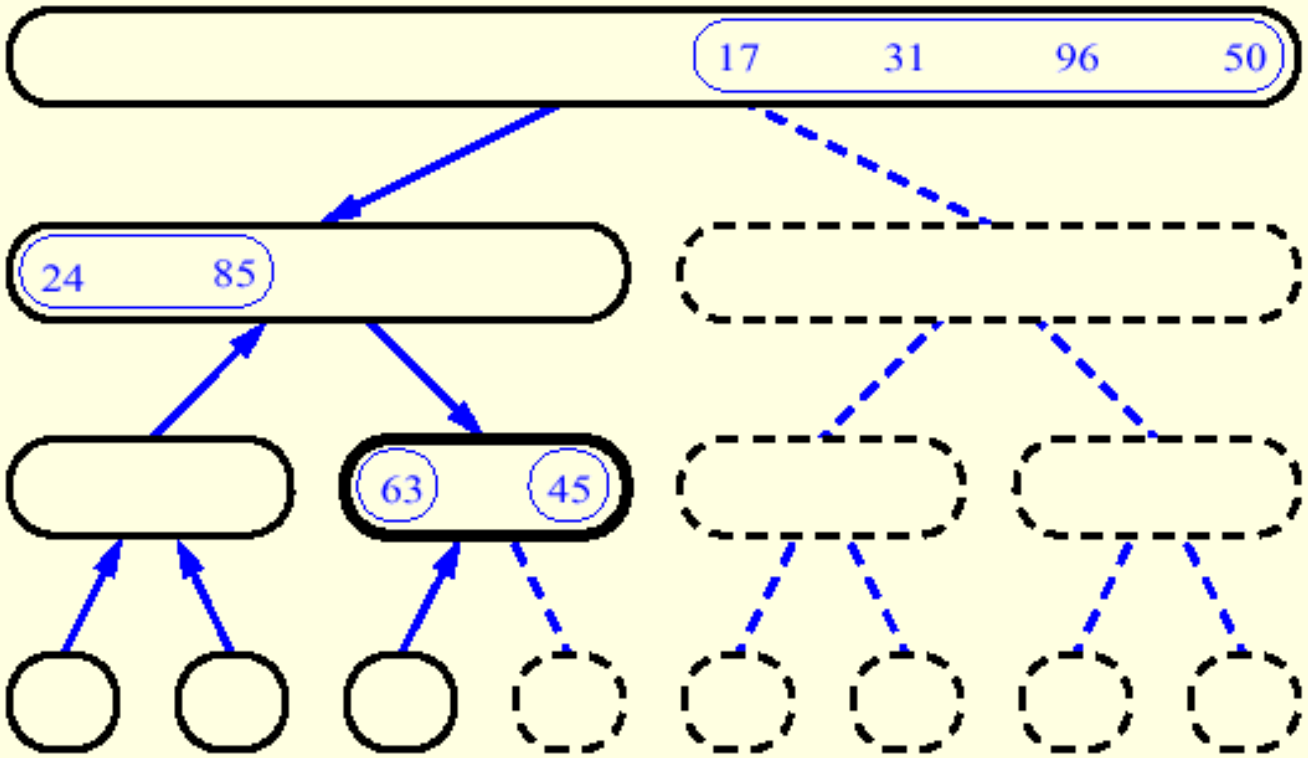
MergeSort (Example) - 10



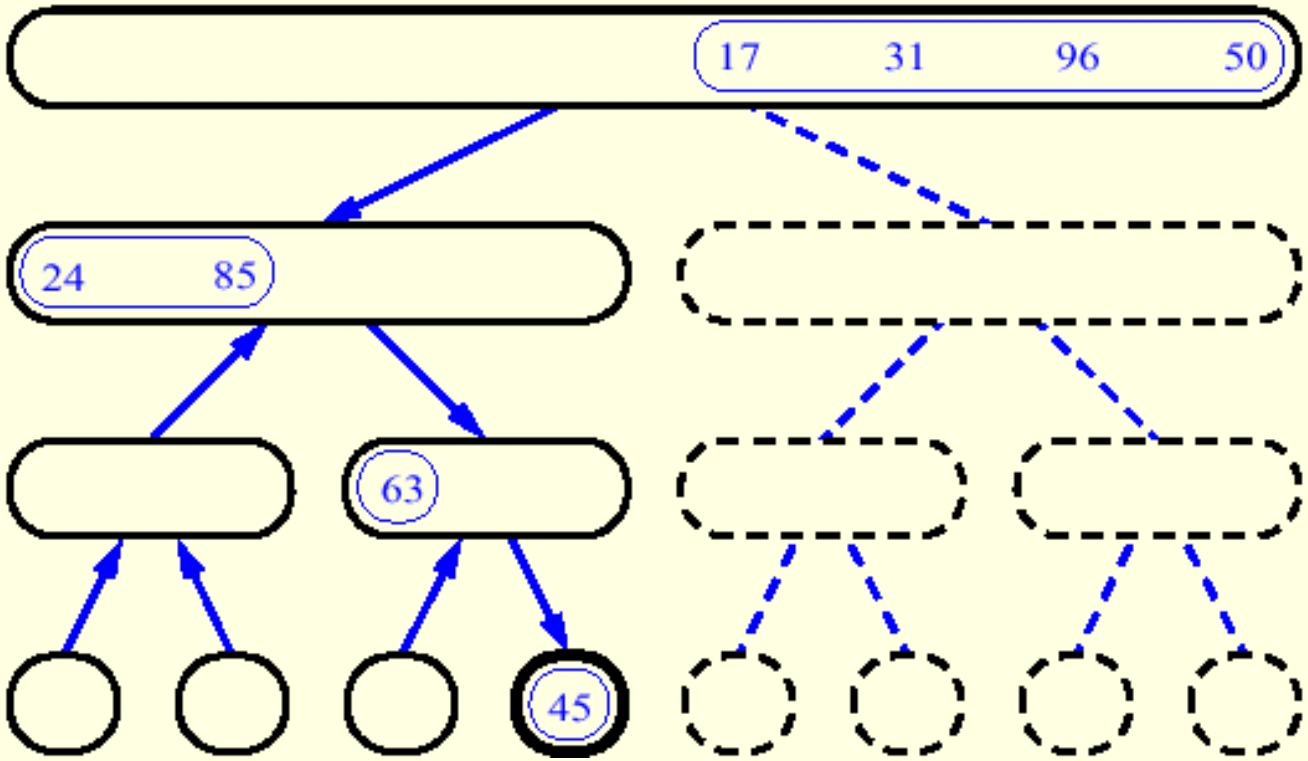
MergeSort (Example) - 11



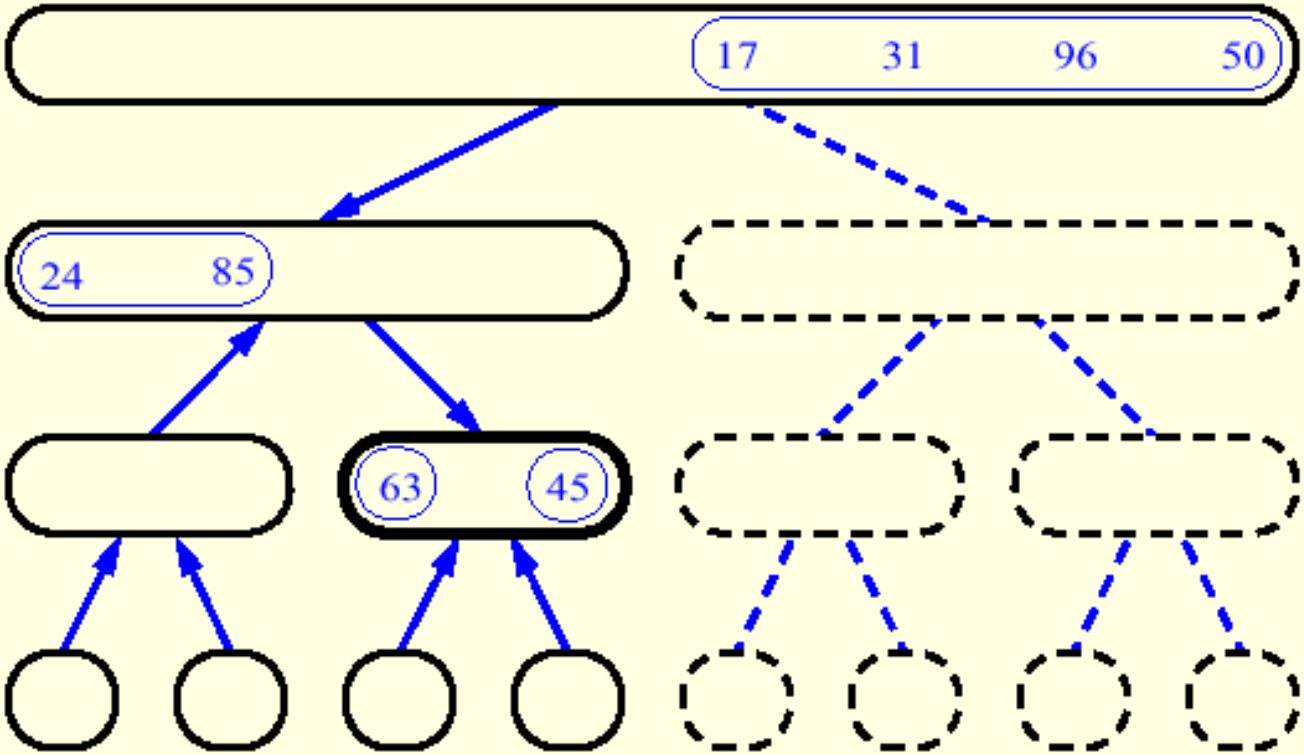
MergeSort (Example) - 12



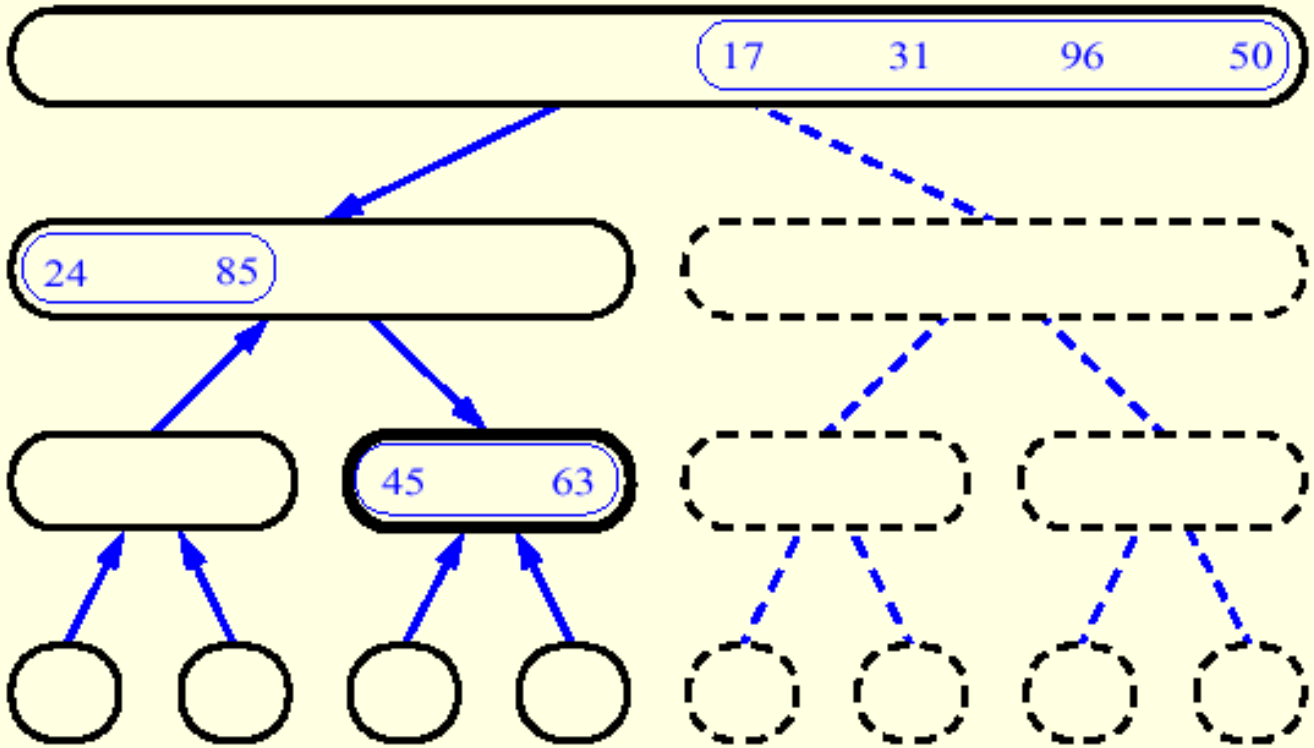
MergeSort (Example) - 13



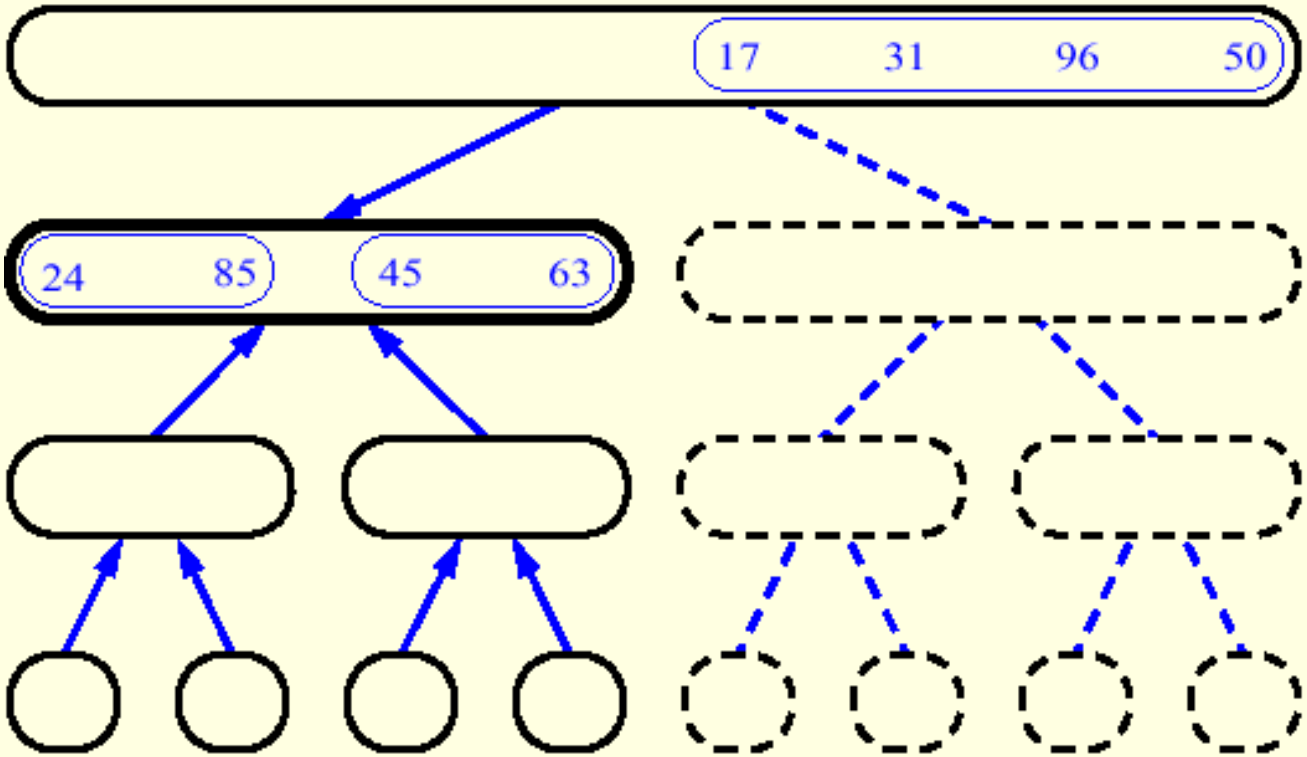
MergeSort (Example) - 14



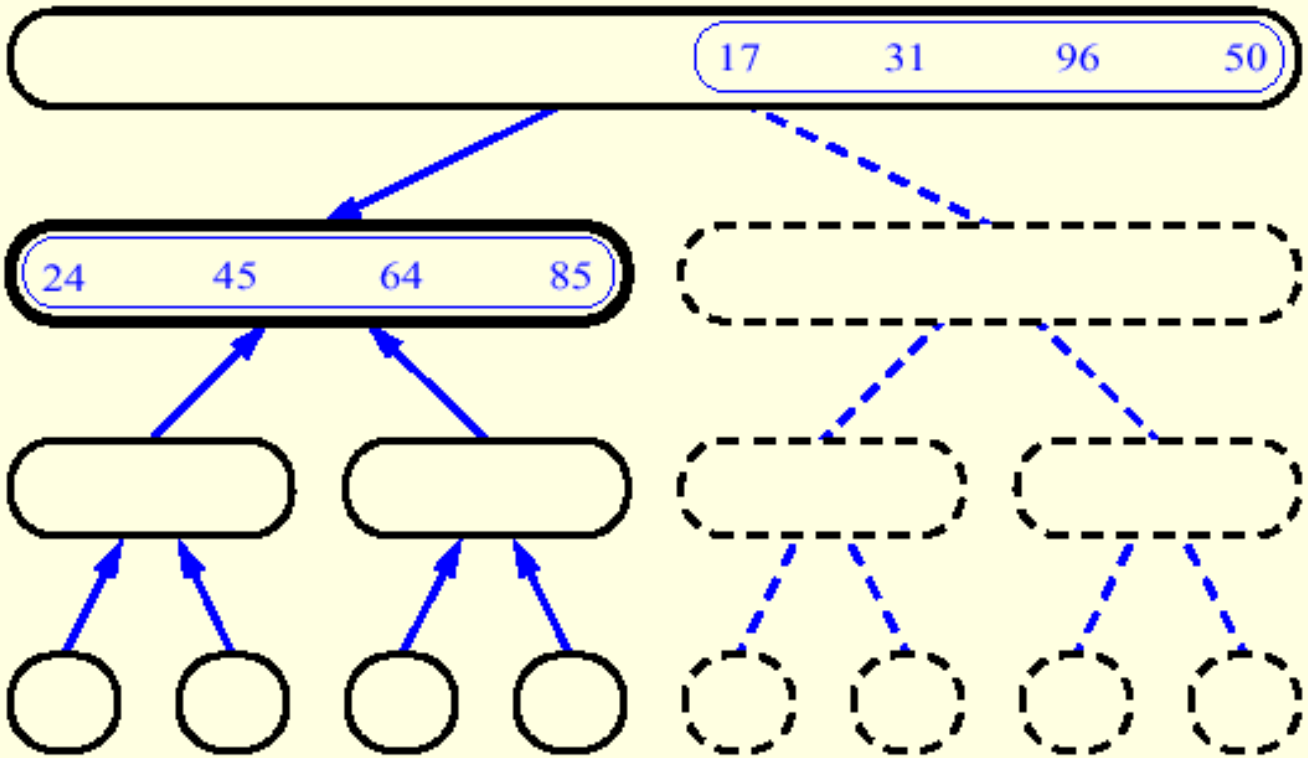
MergeSort (Example) - 15



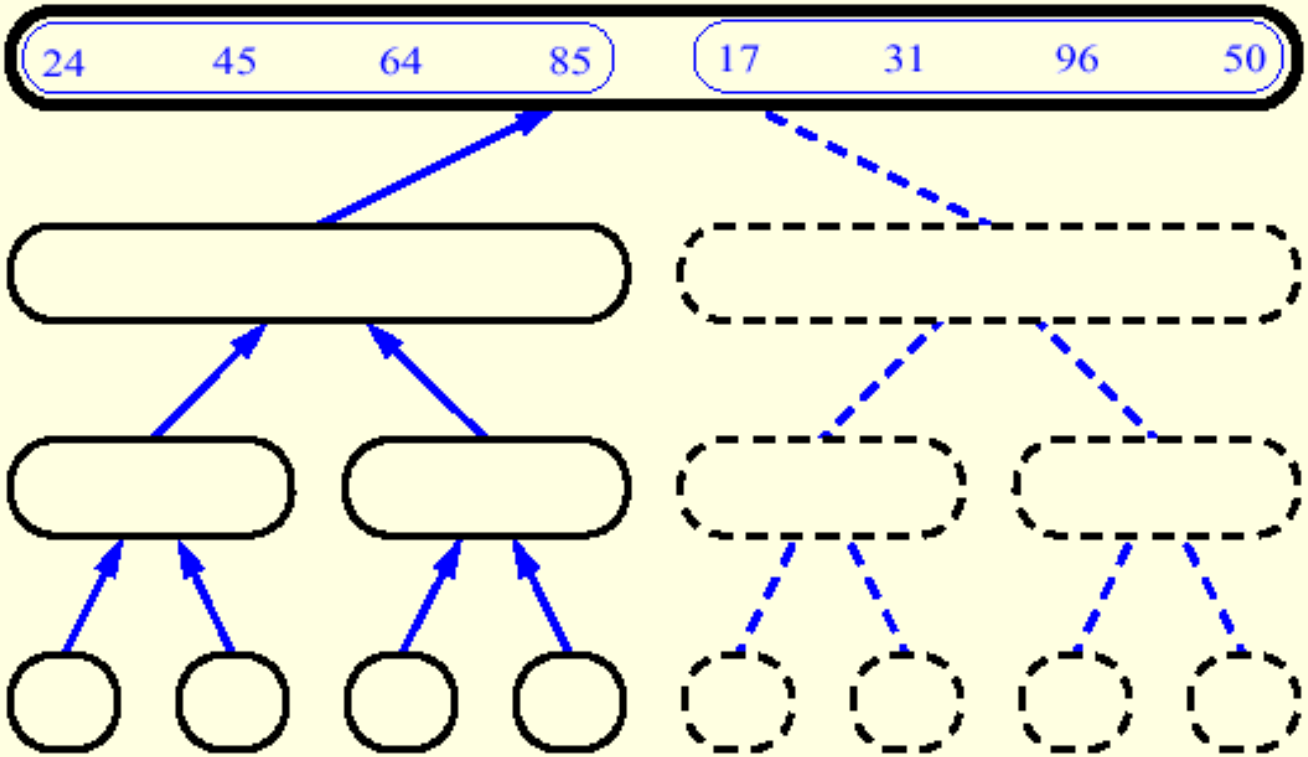
MergeSort (Example) - 16



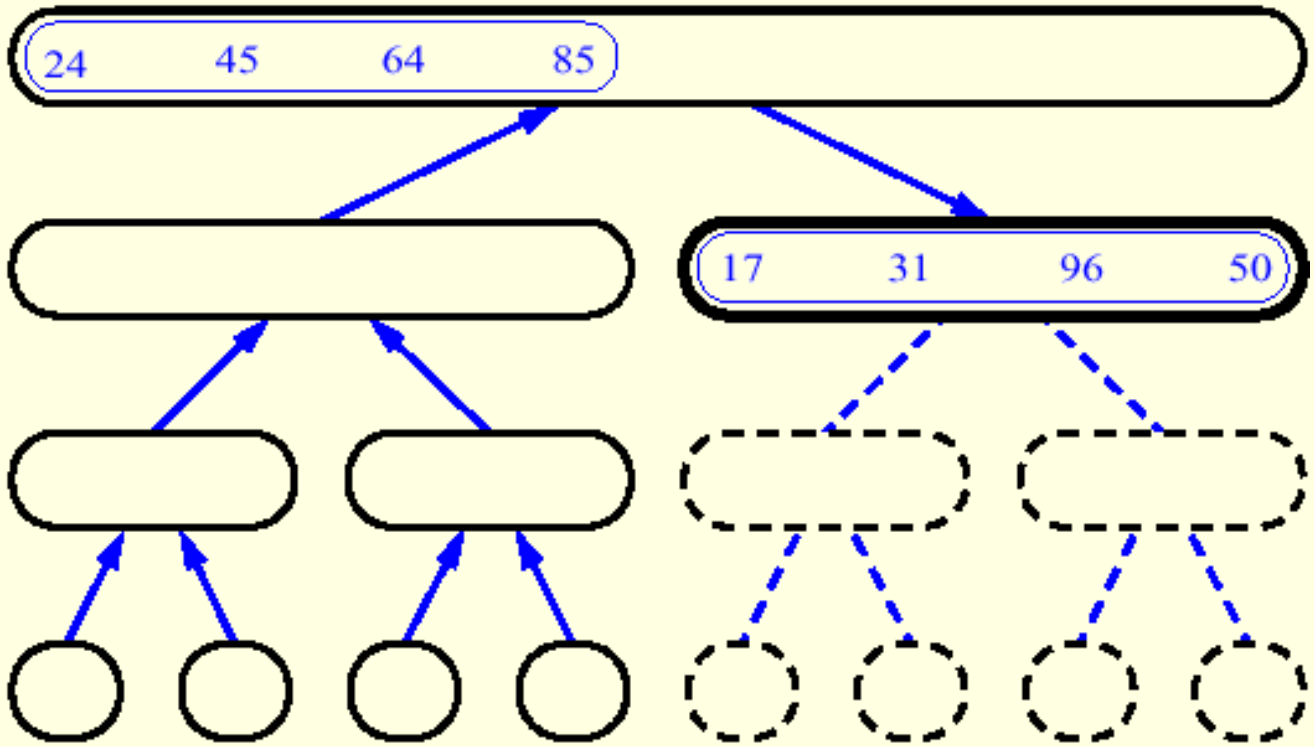
MergeSort (Example) - 17



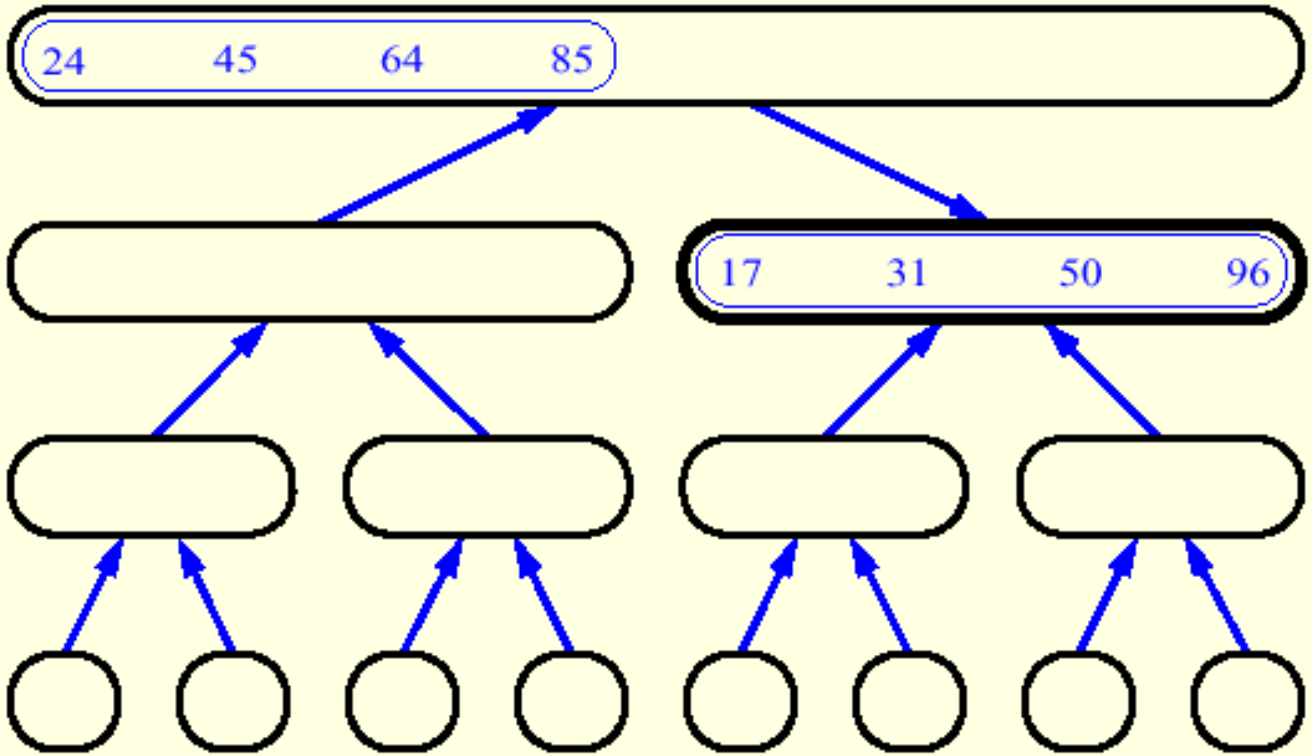
MergeSort (Example) - 18



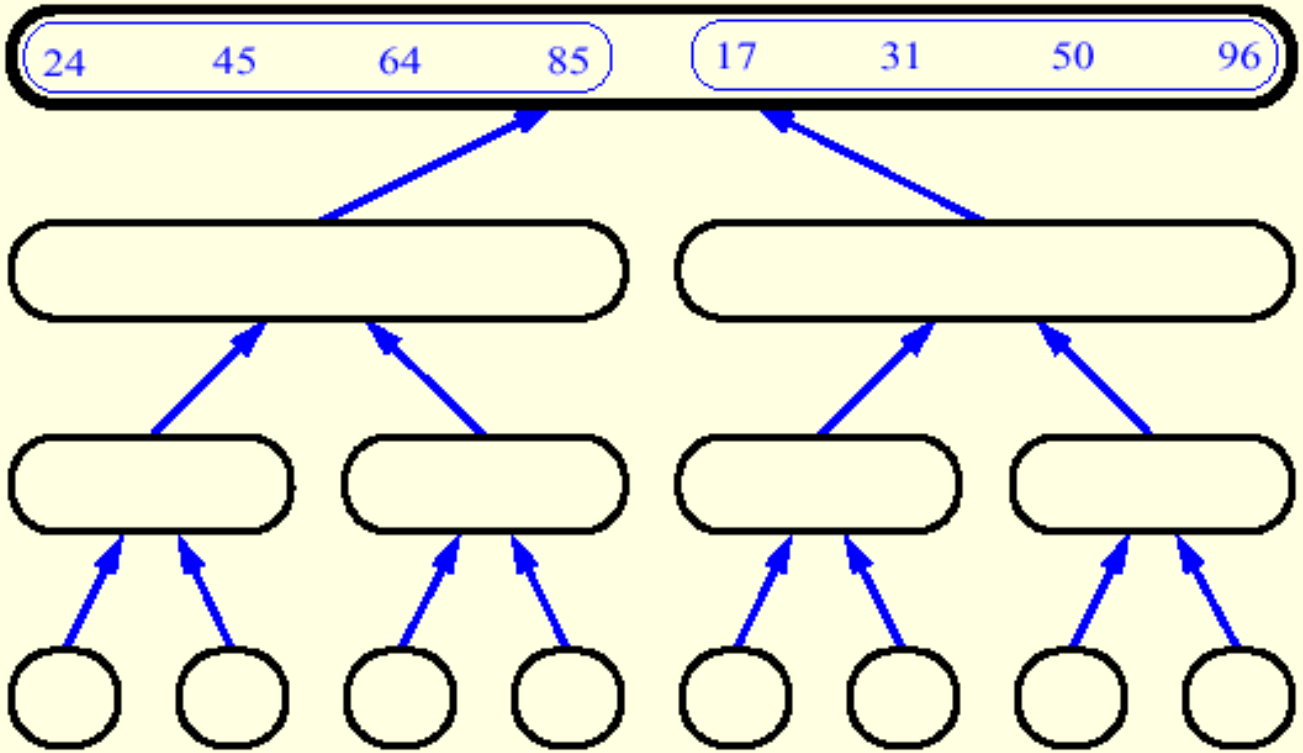
MergeSort (Example) - 19



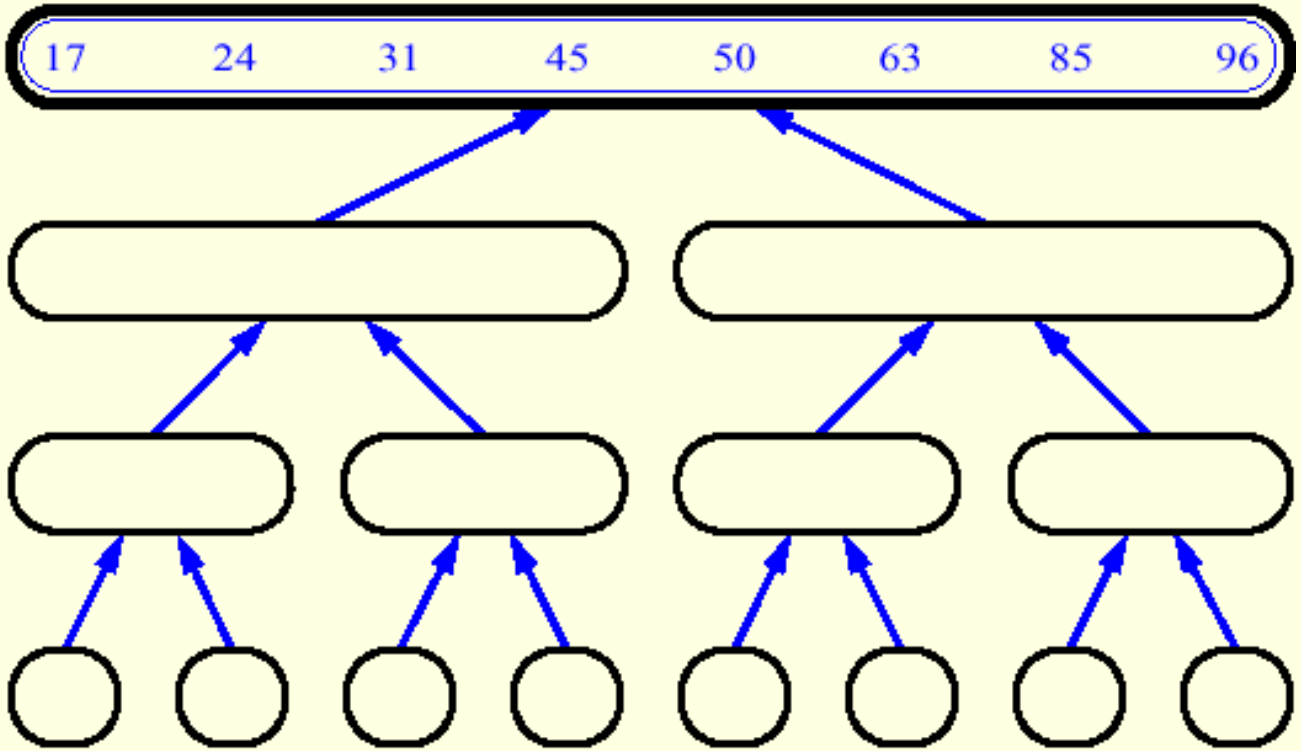
MergeSort (Example) - 20

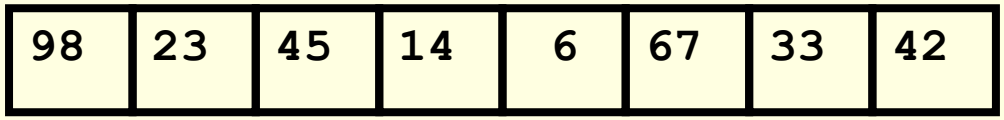


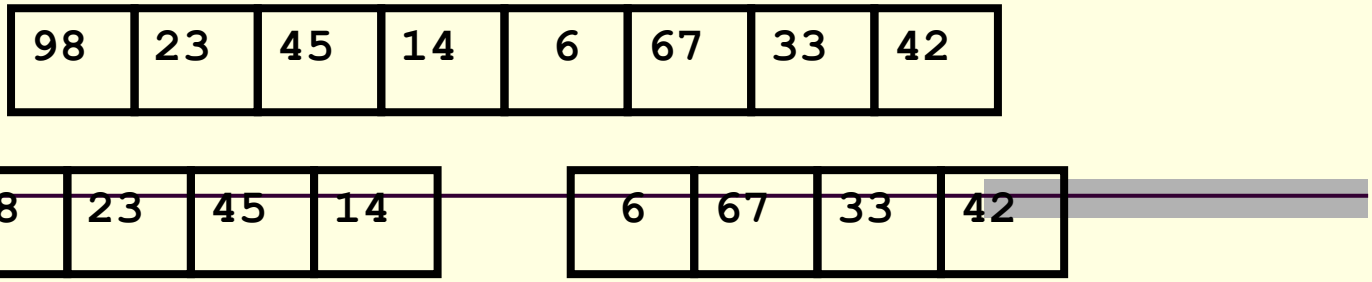
MergeSort (Example) - 21

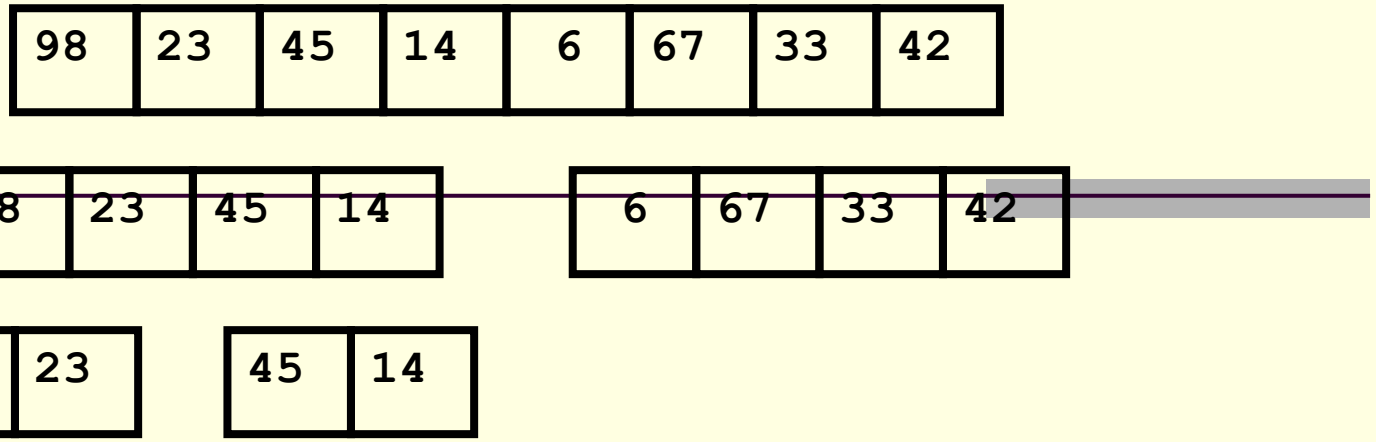


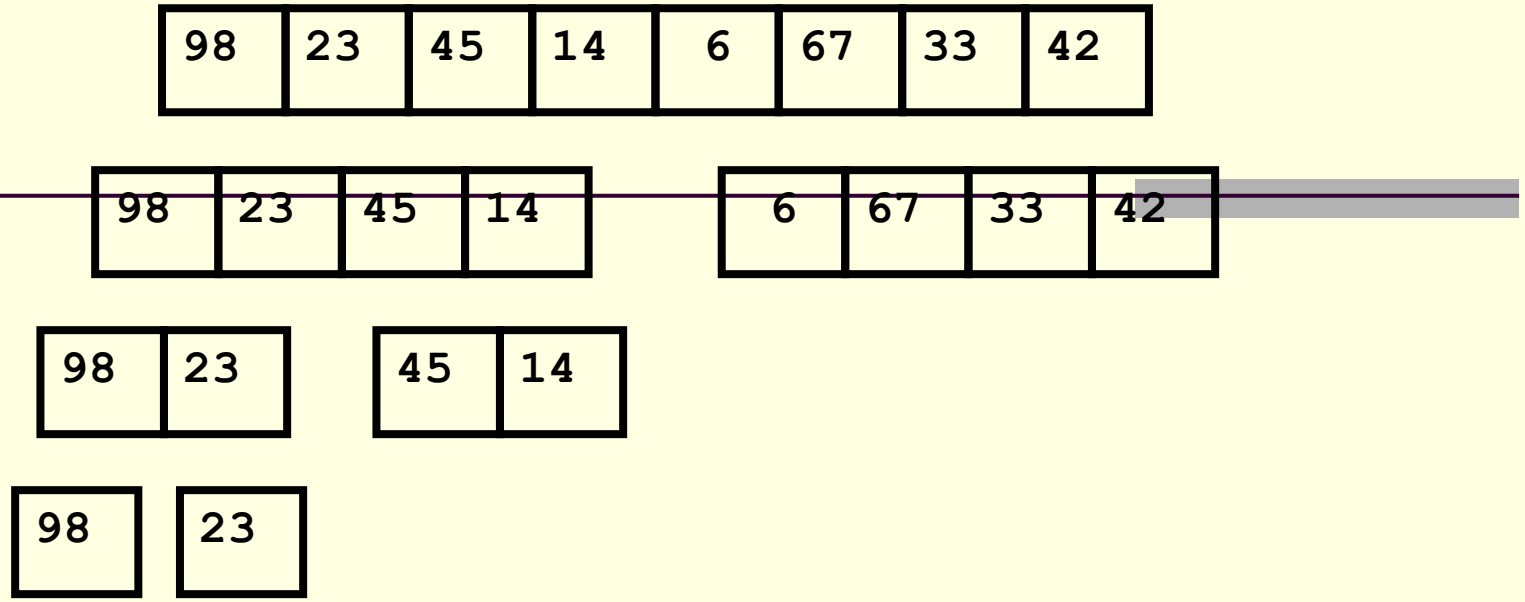
MergeSort (Example) - 22

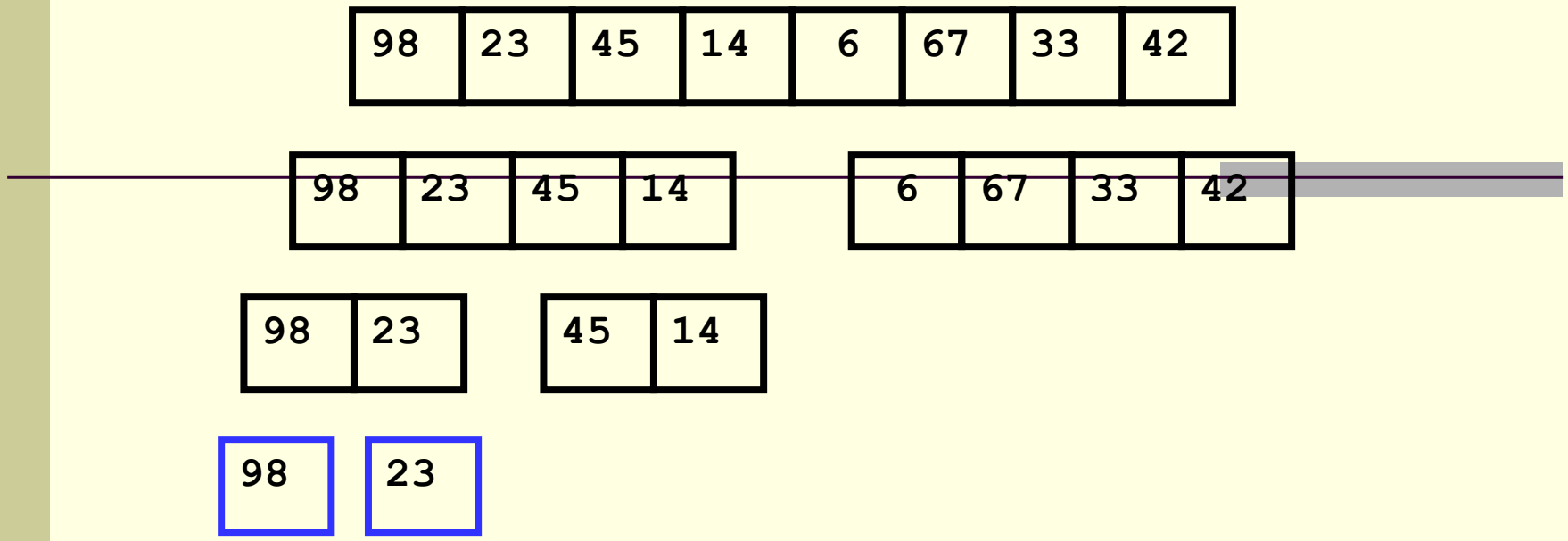


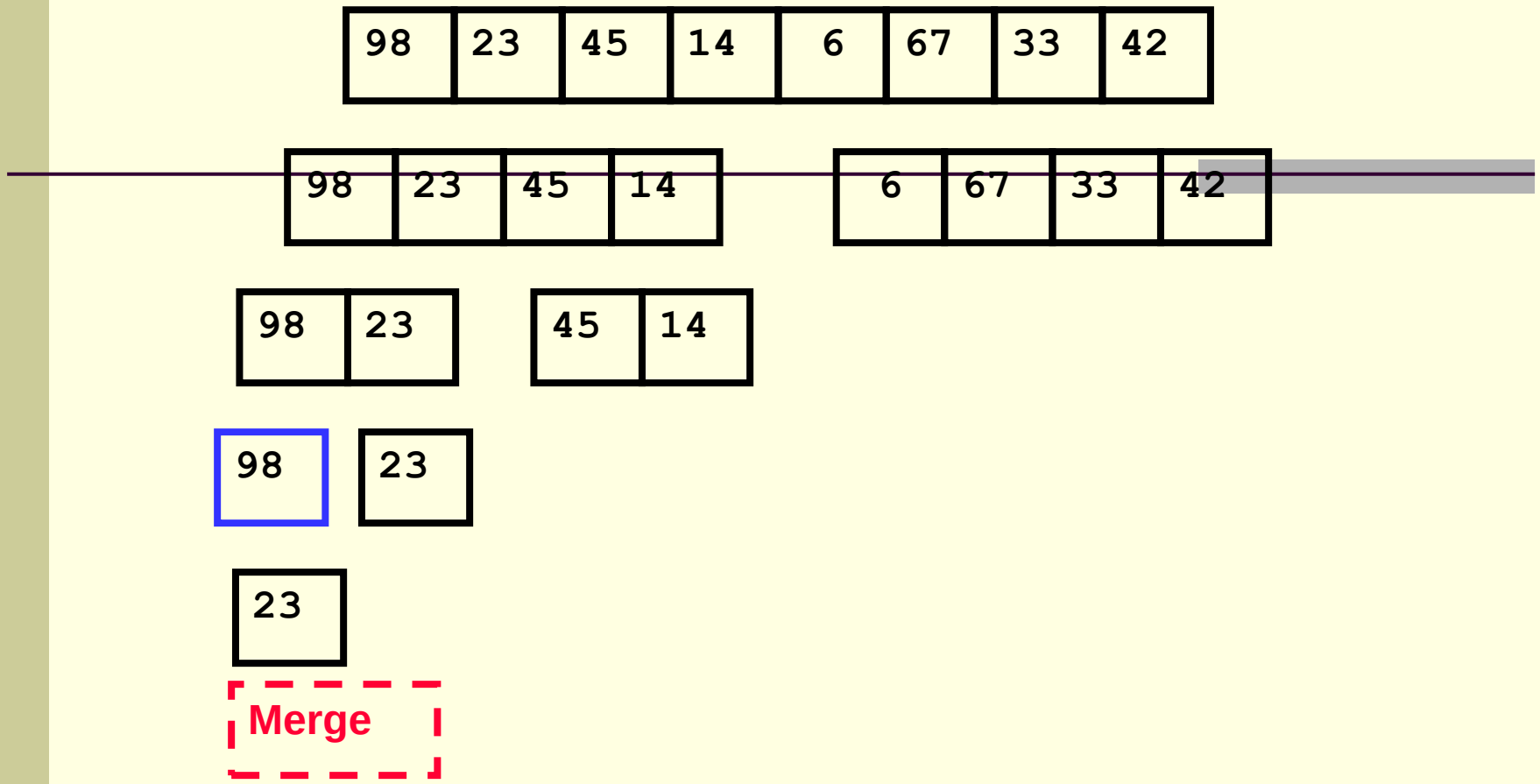

98
23
45
14
6
67
33
42

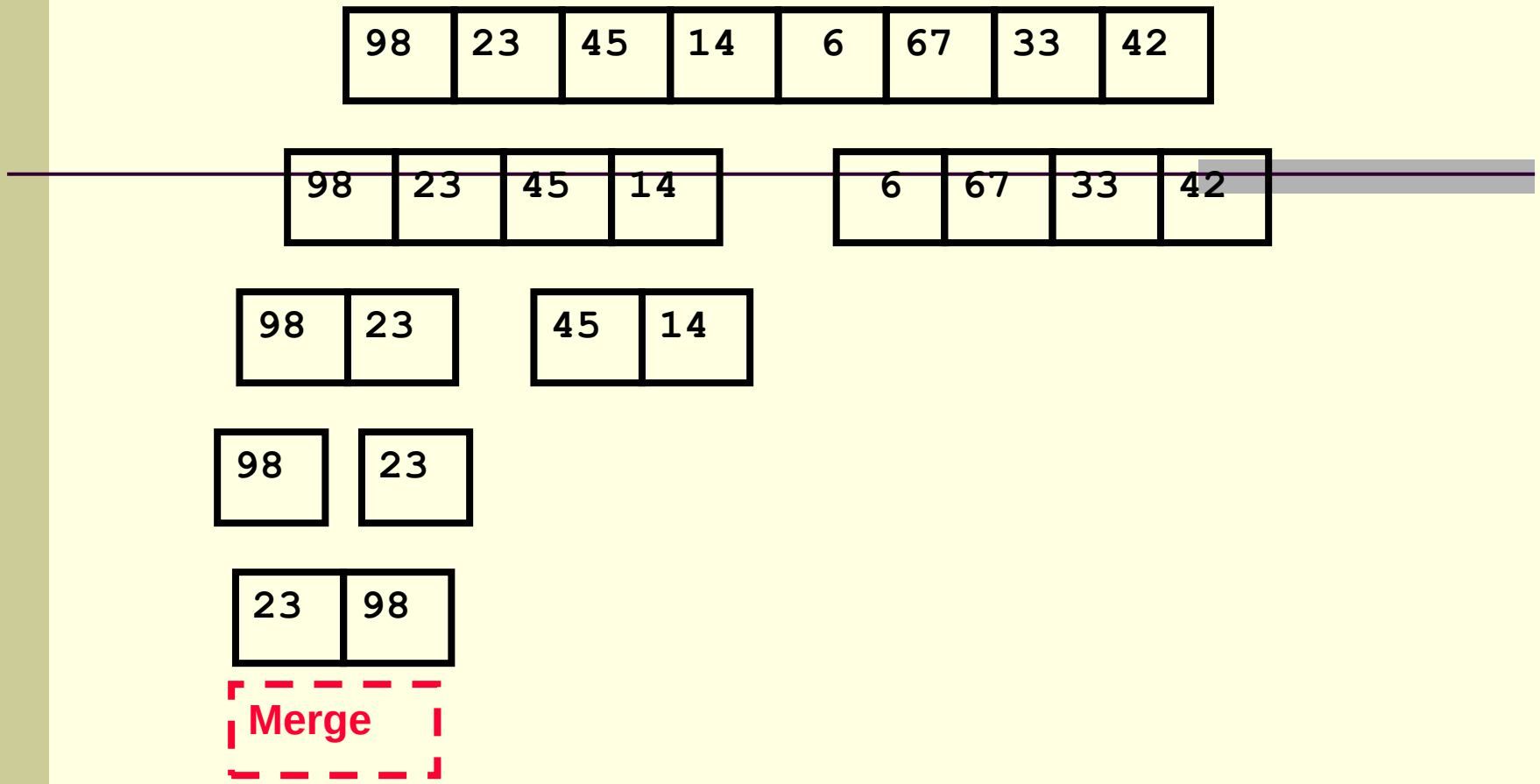

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42

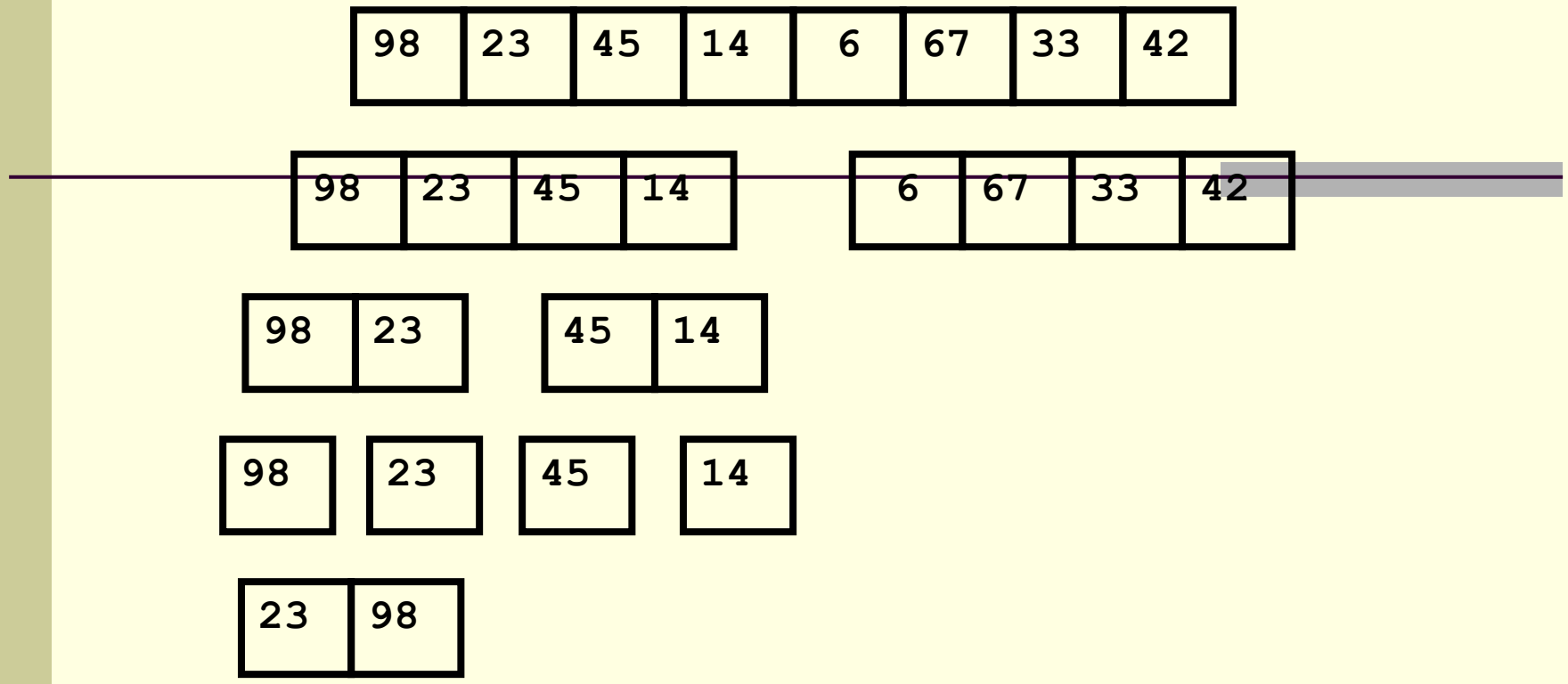

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14

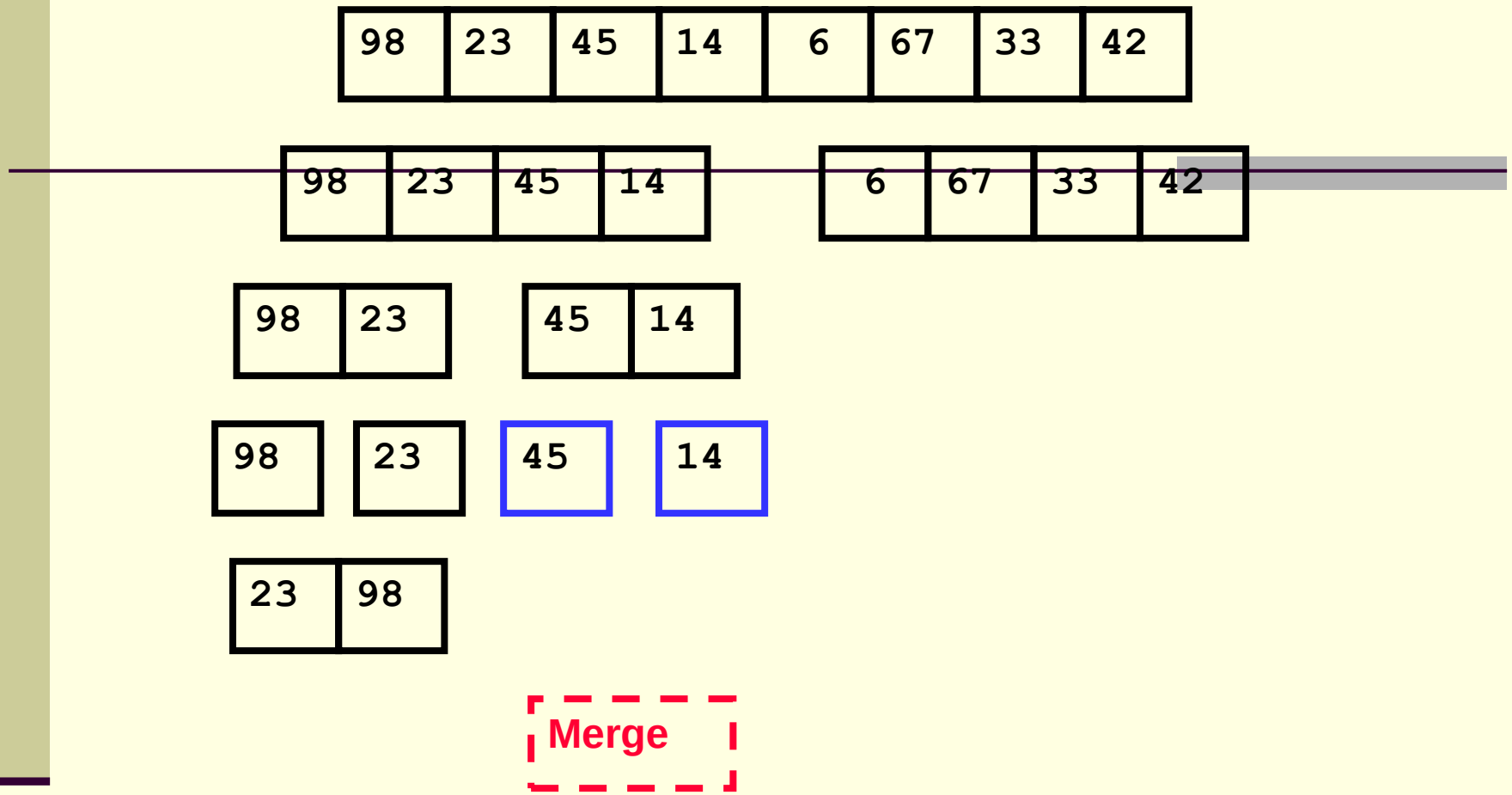

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23

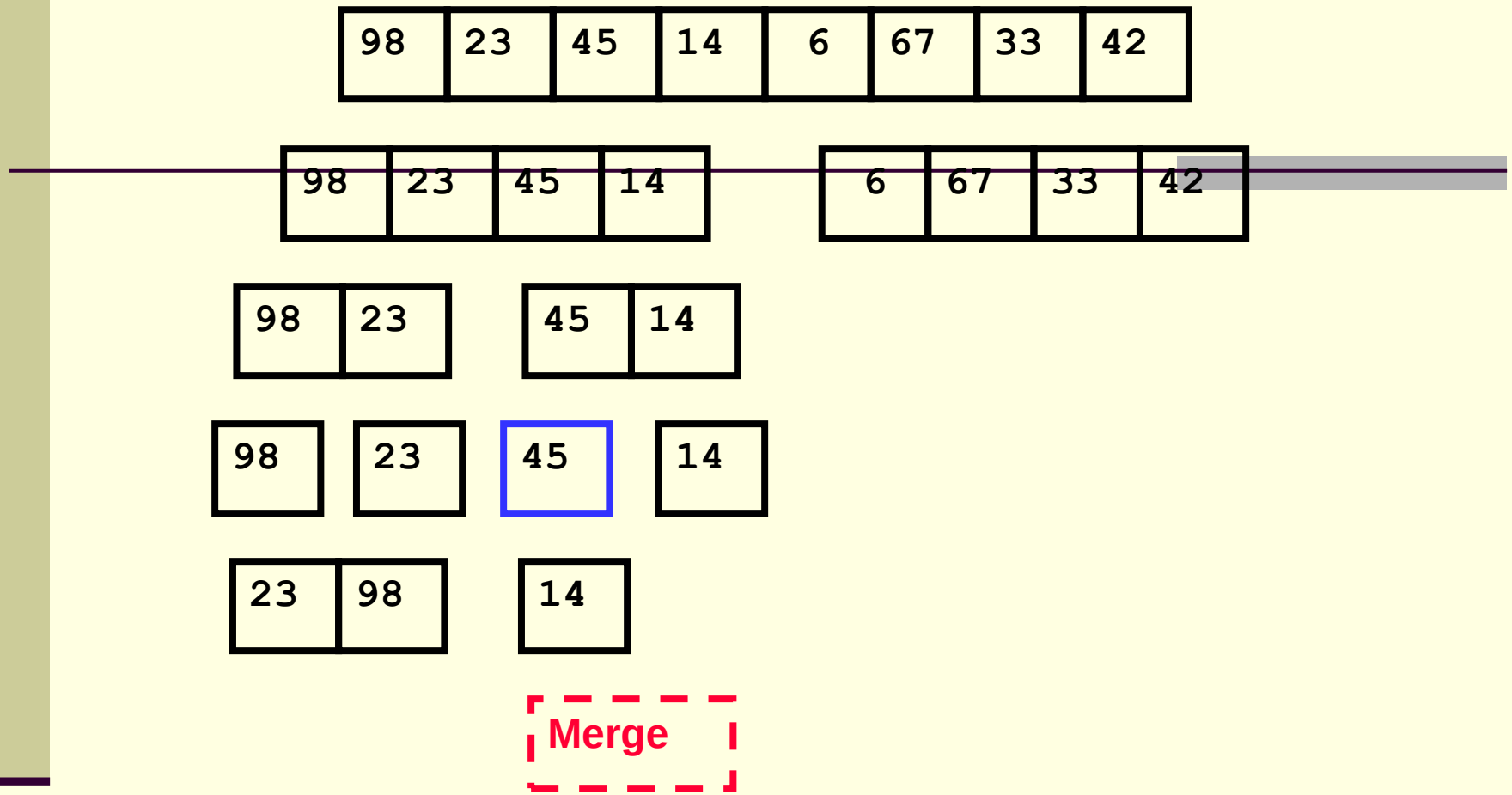

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
Merge

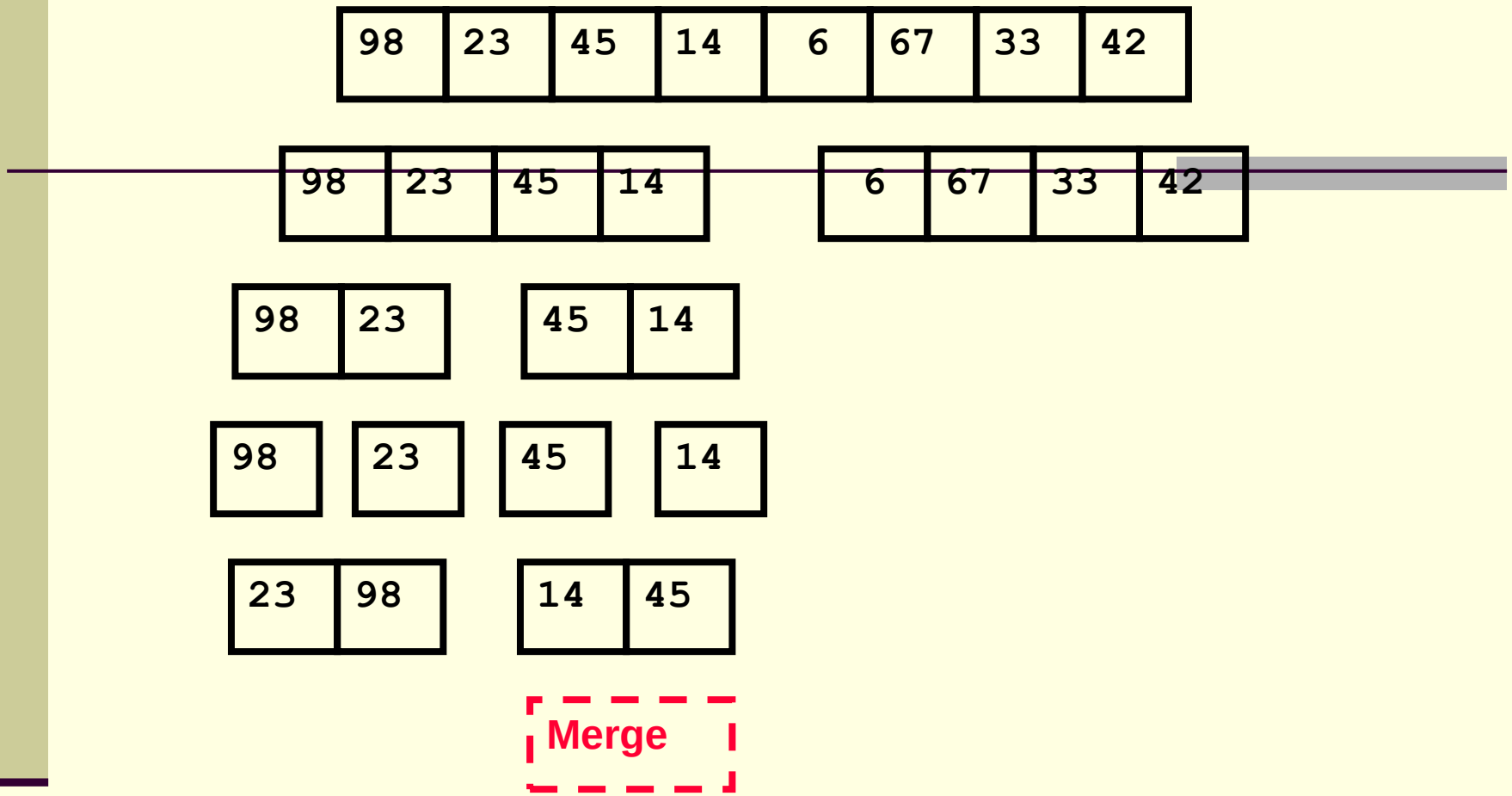

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23
Merge

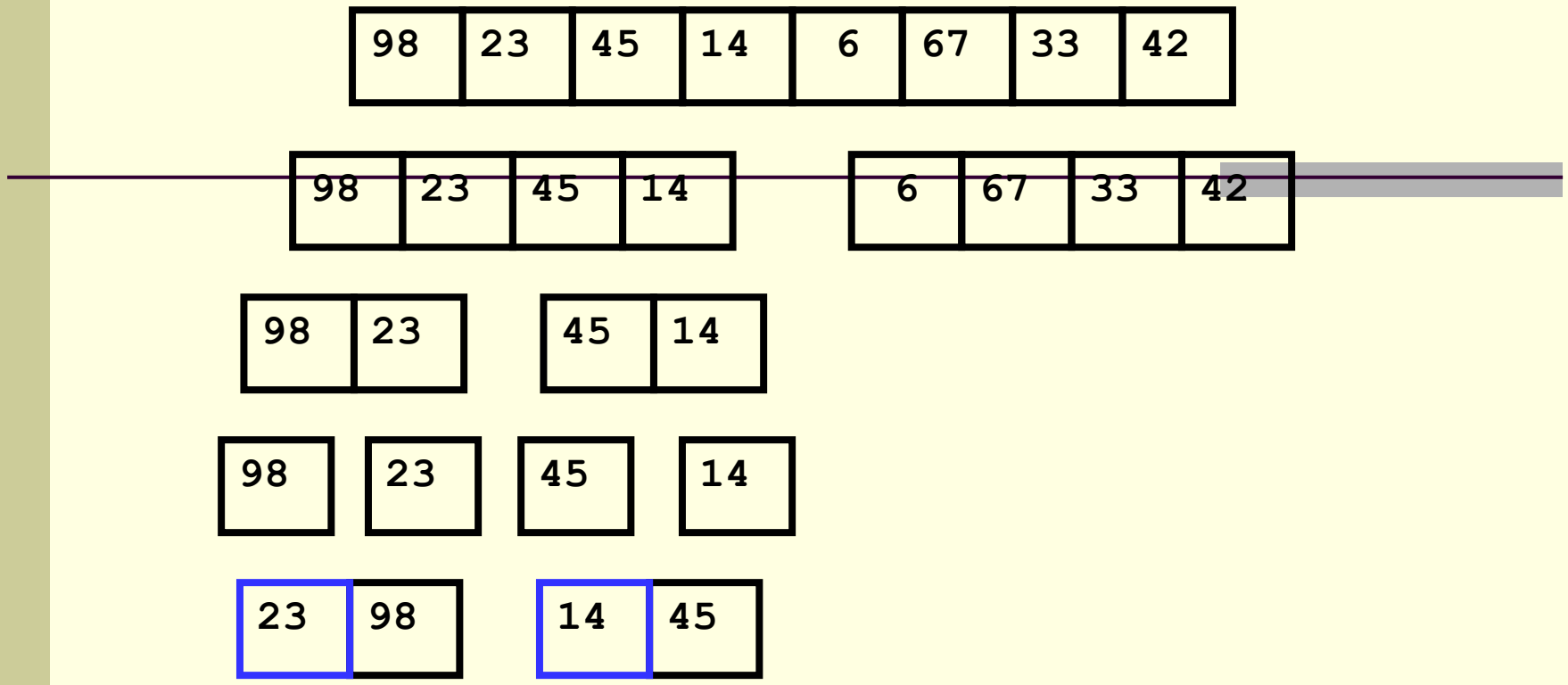

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23 98
Merge

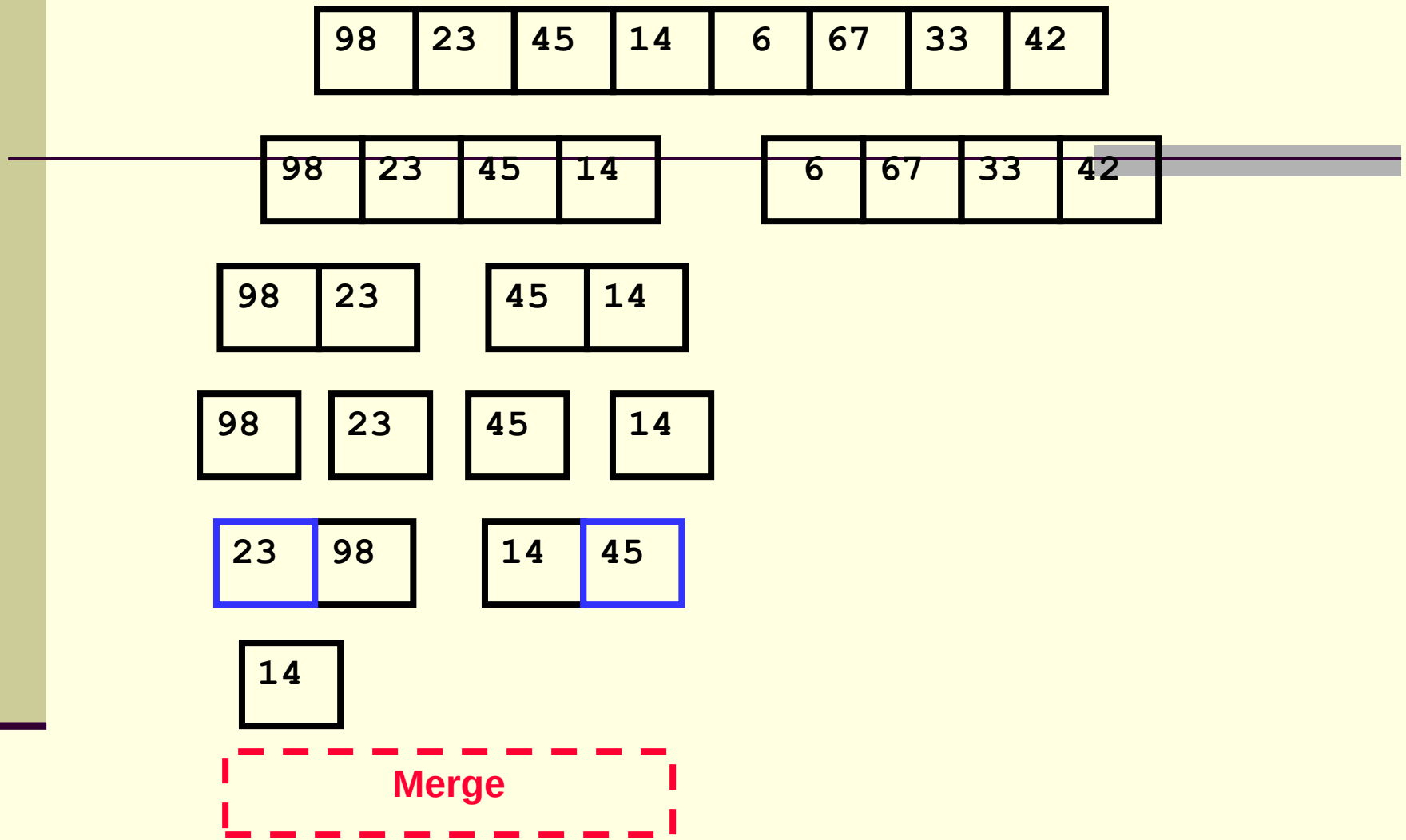

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98

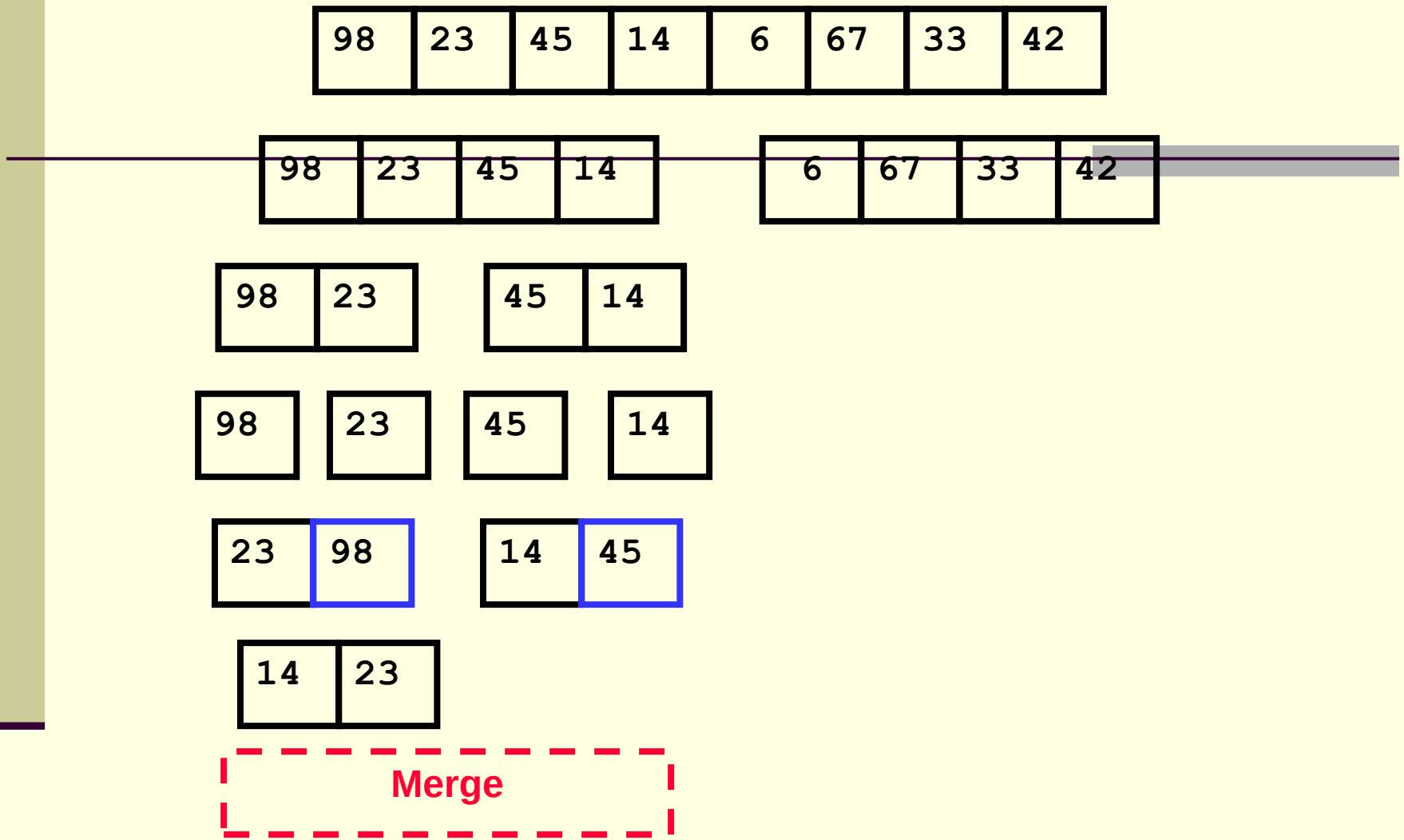

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
Merge

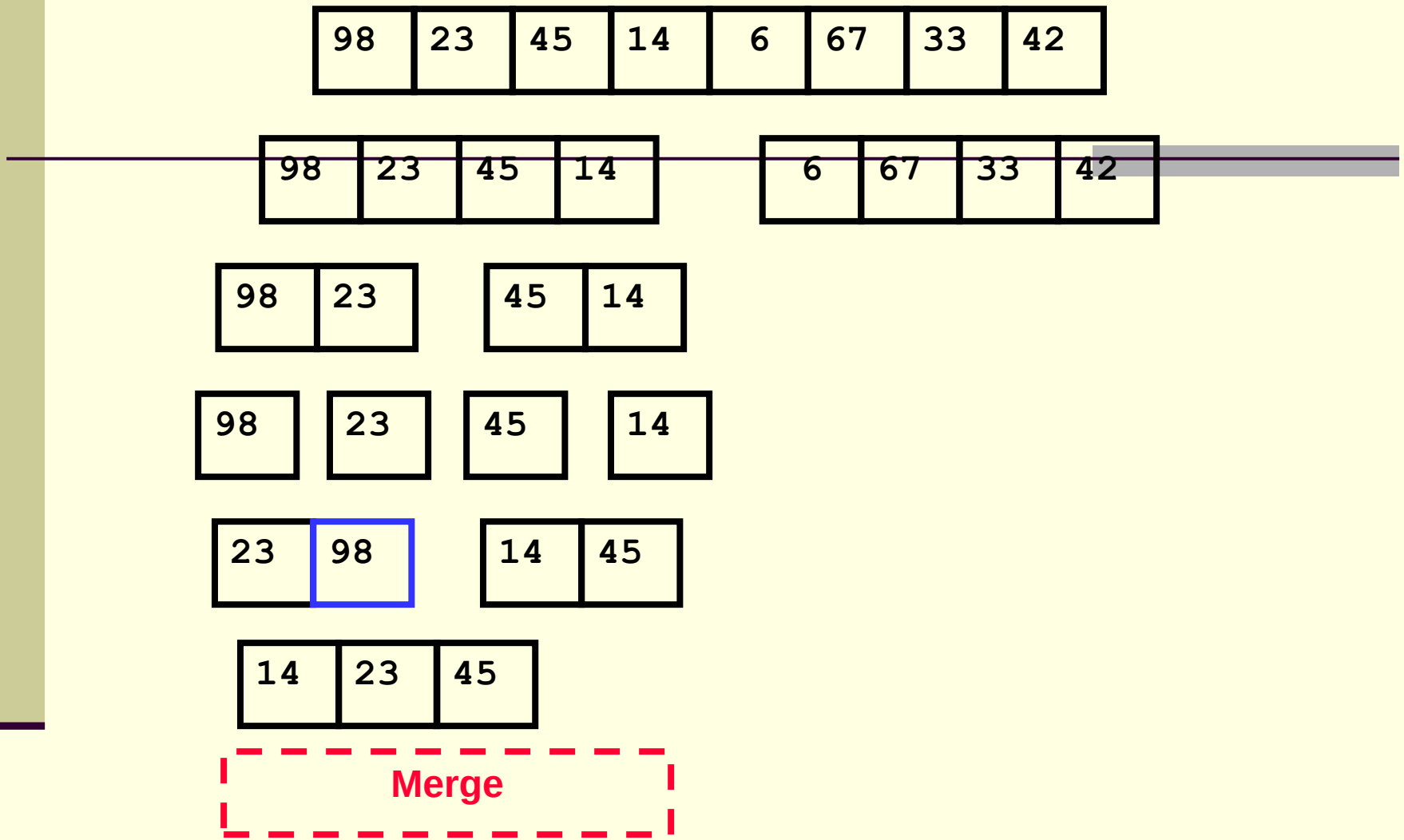

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14
Merge

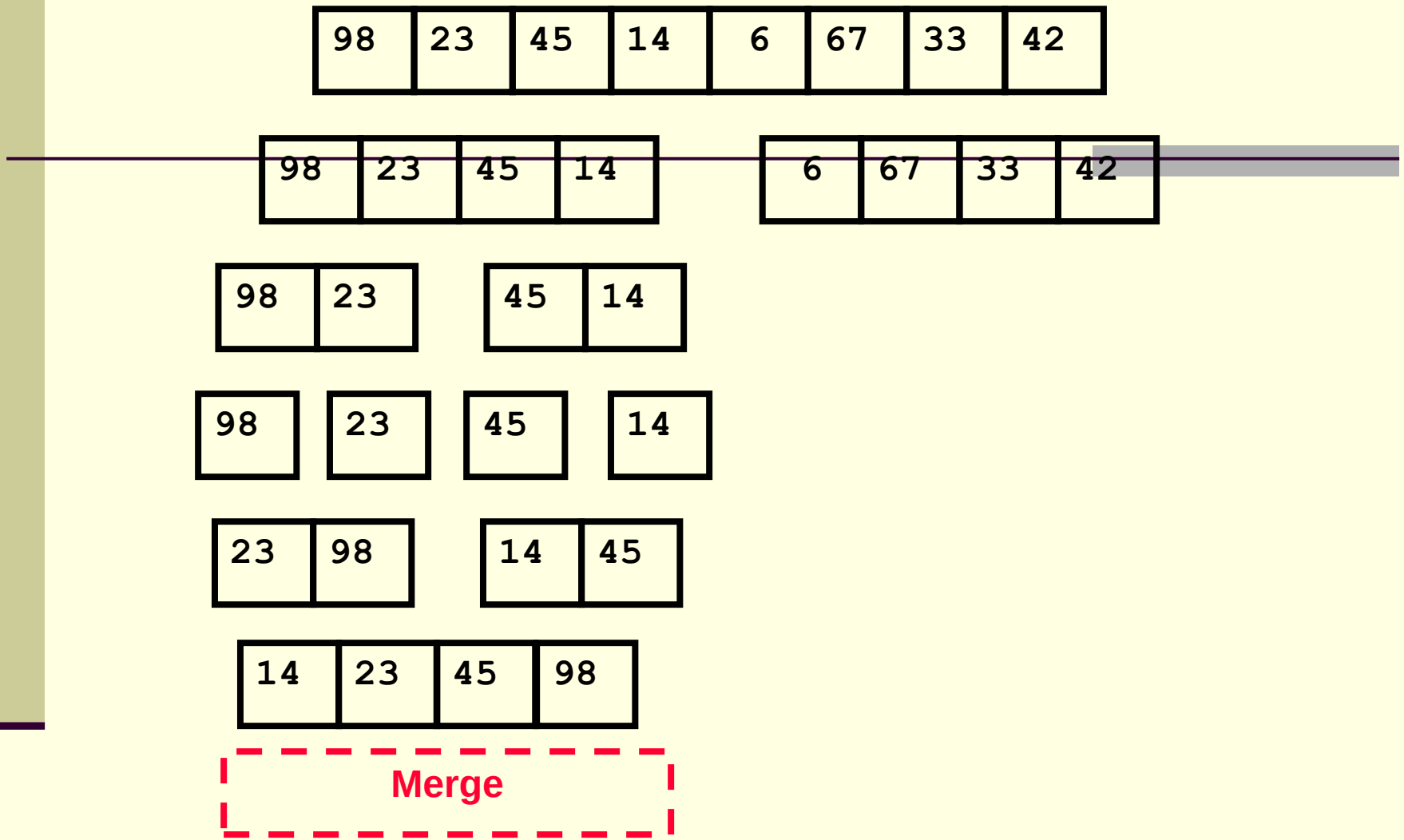

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
Merge

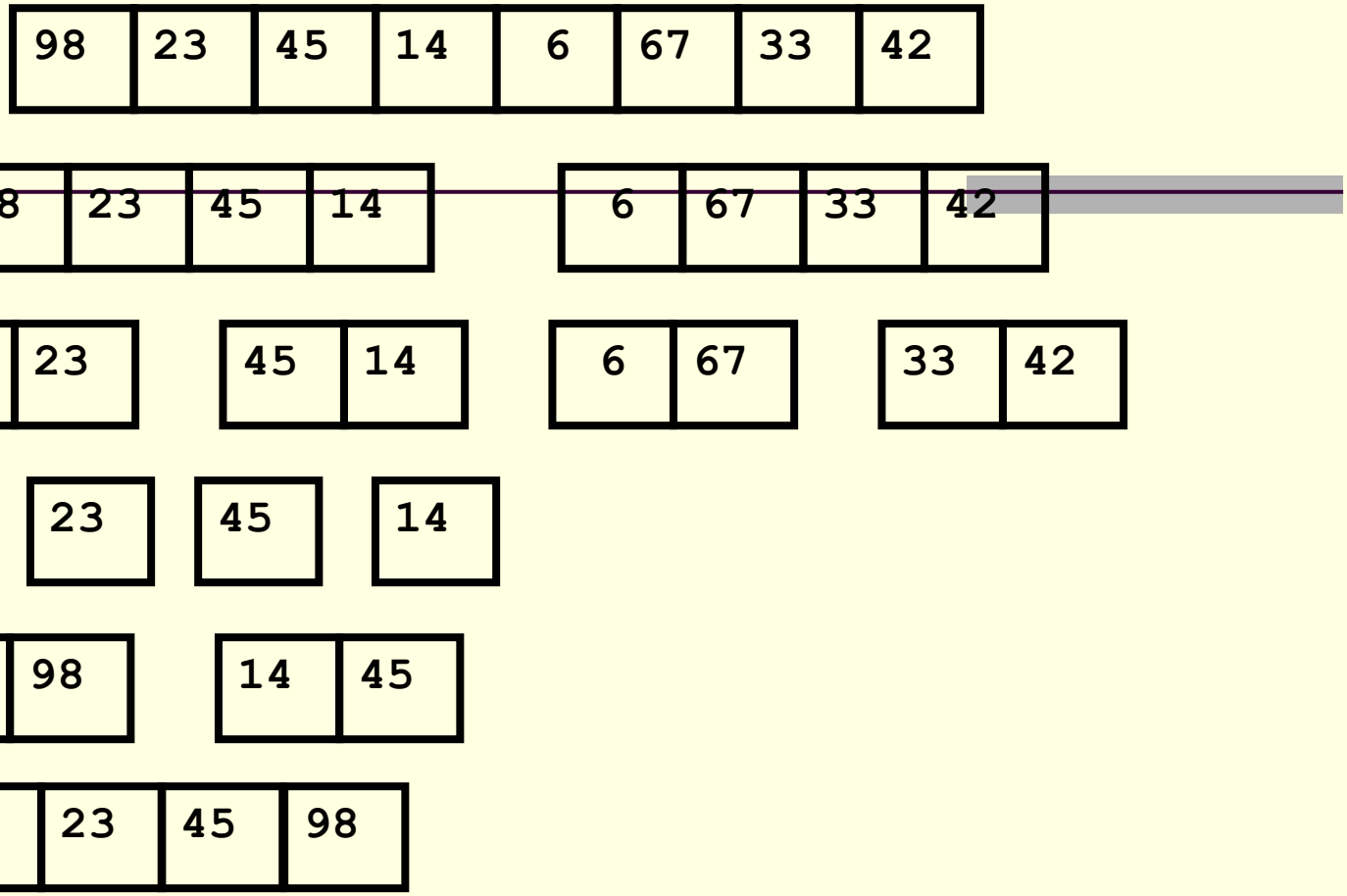

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
Merge

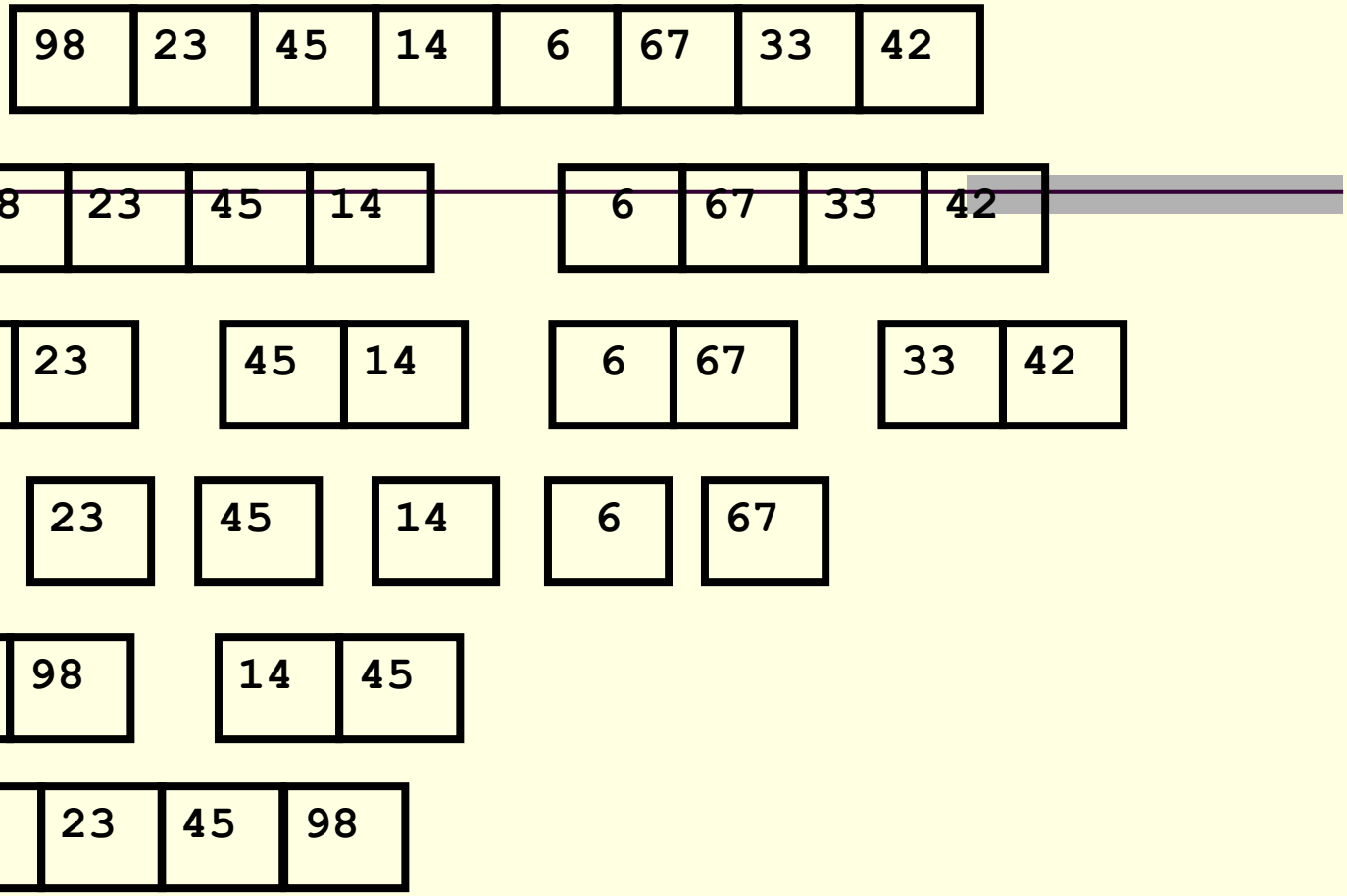

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14
Merge

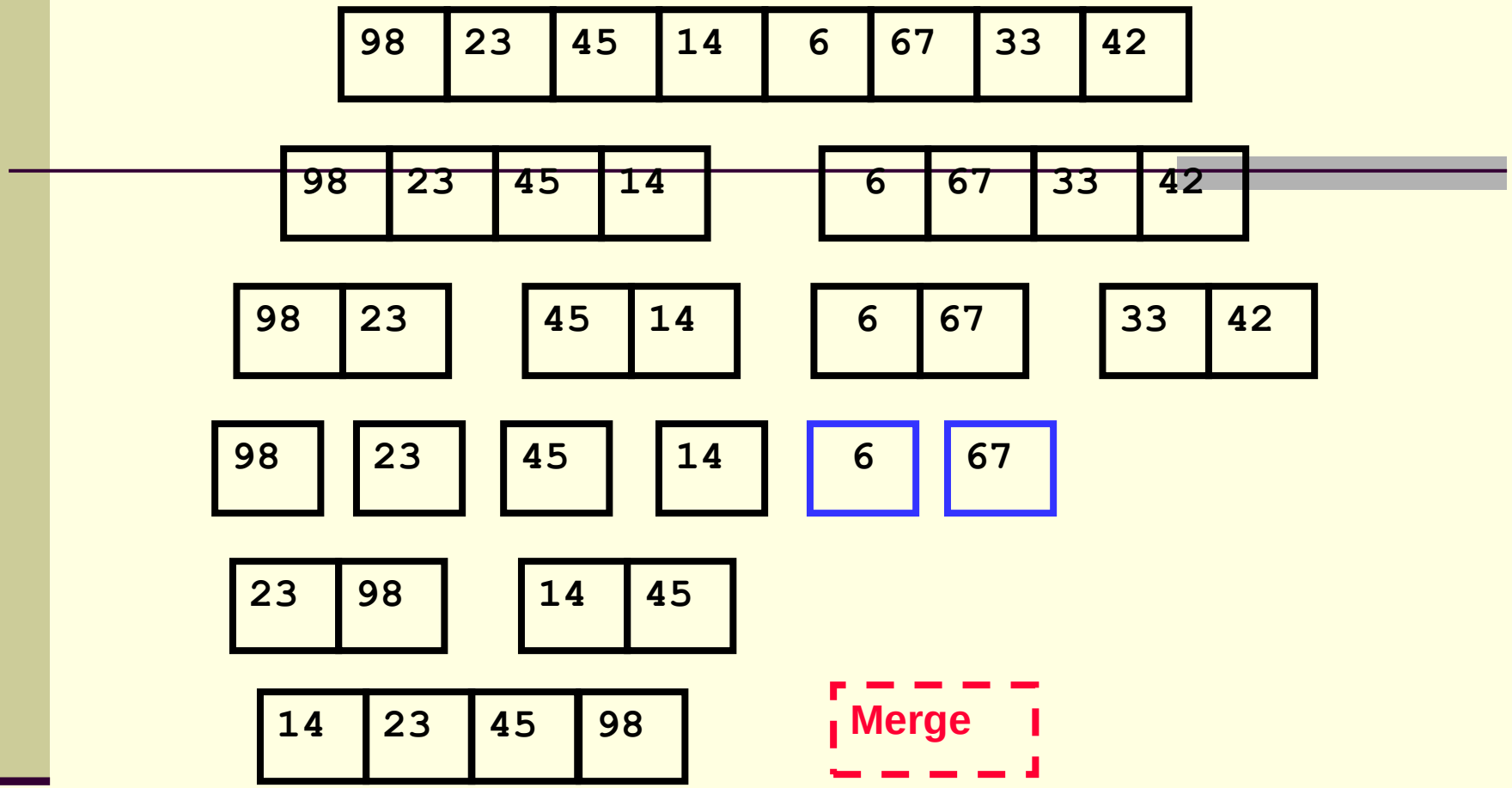

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23
Merge

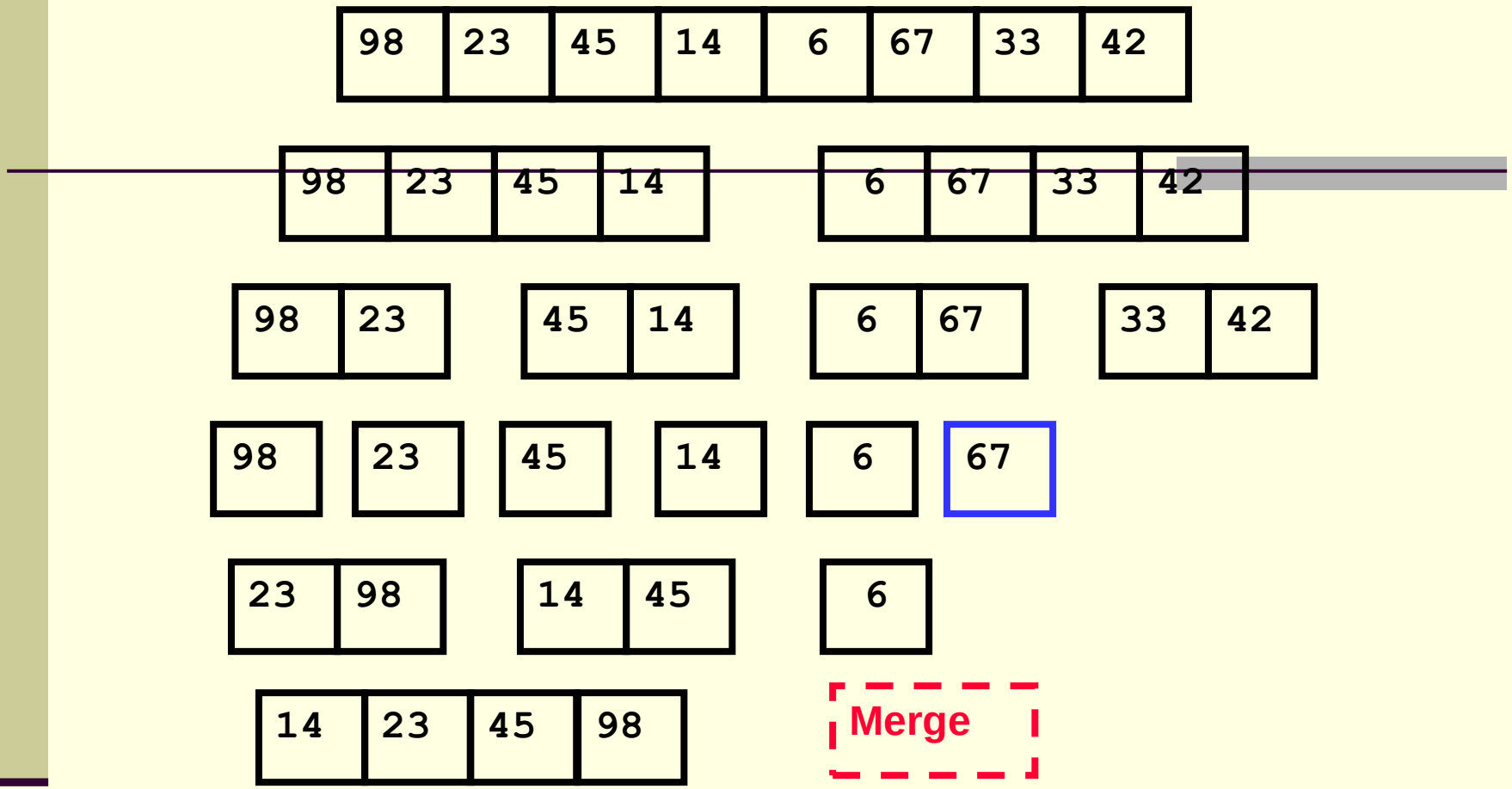

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45
Merge

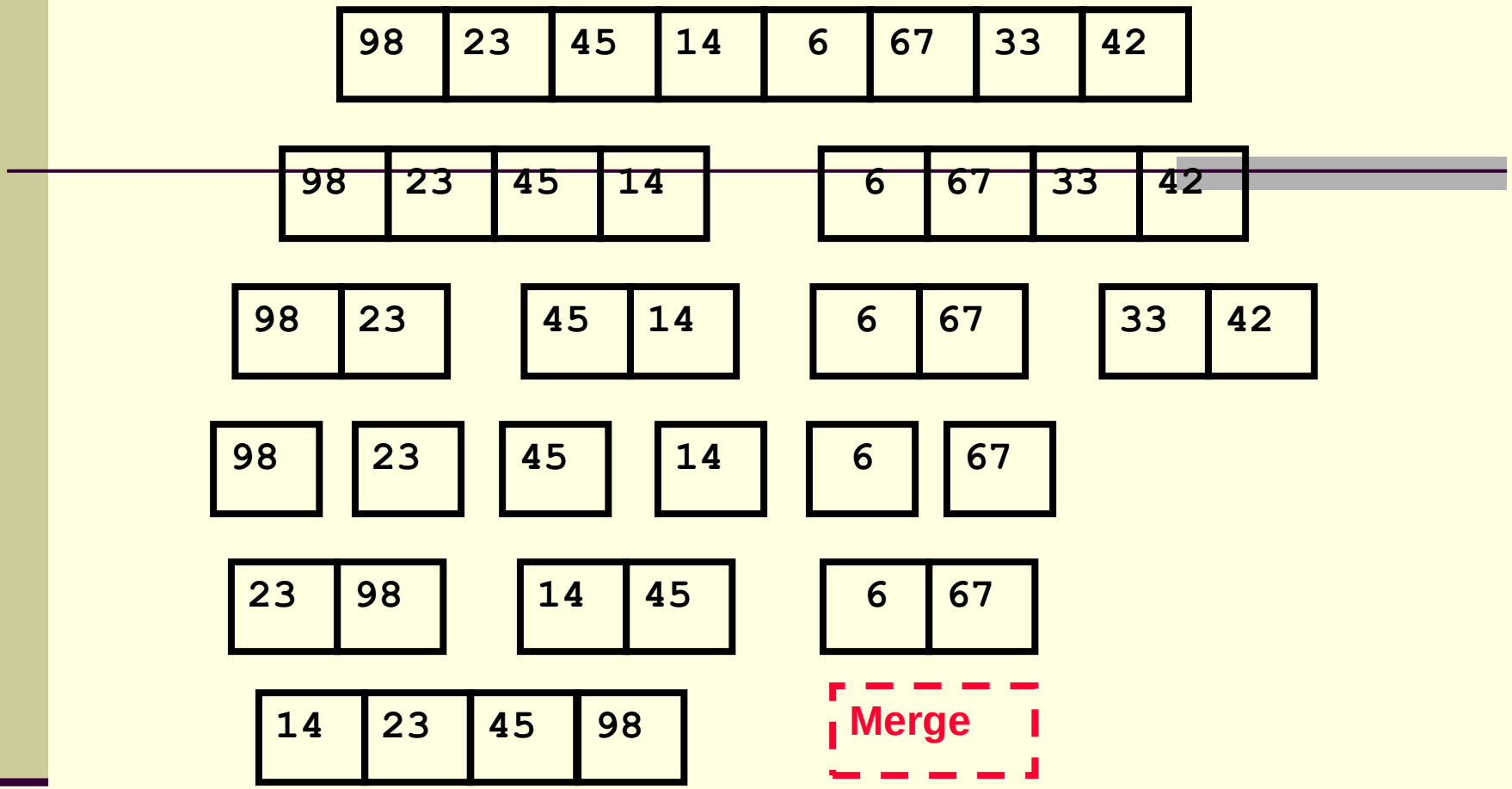

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45 98
Merge

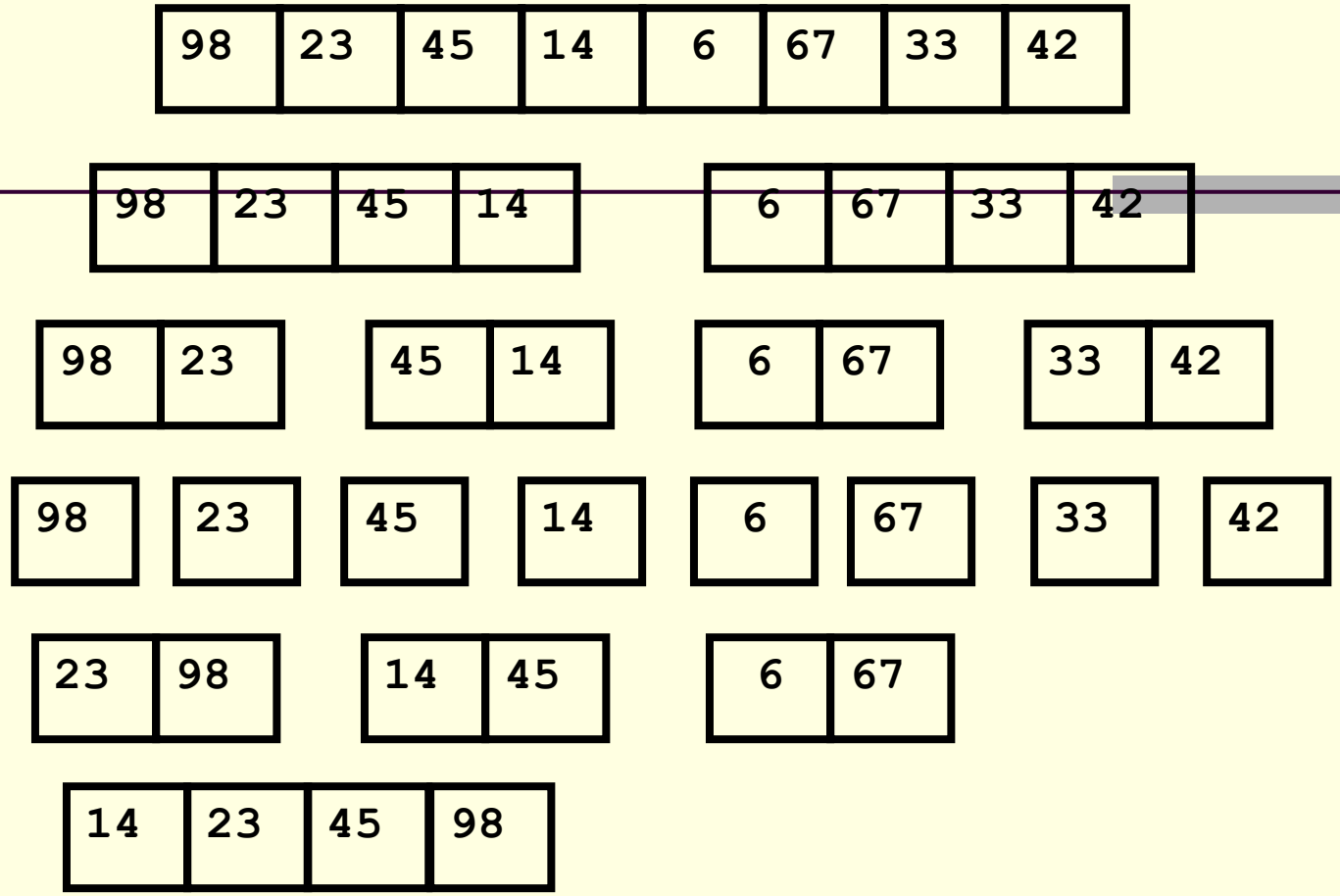

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
23 98
14 45
14 23 45 98

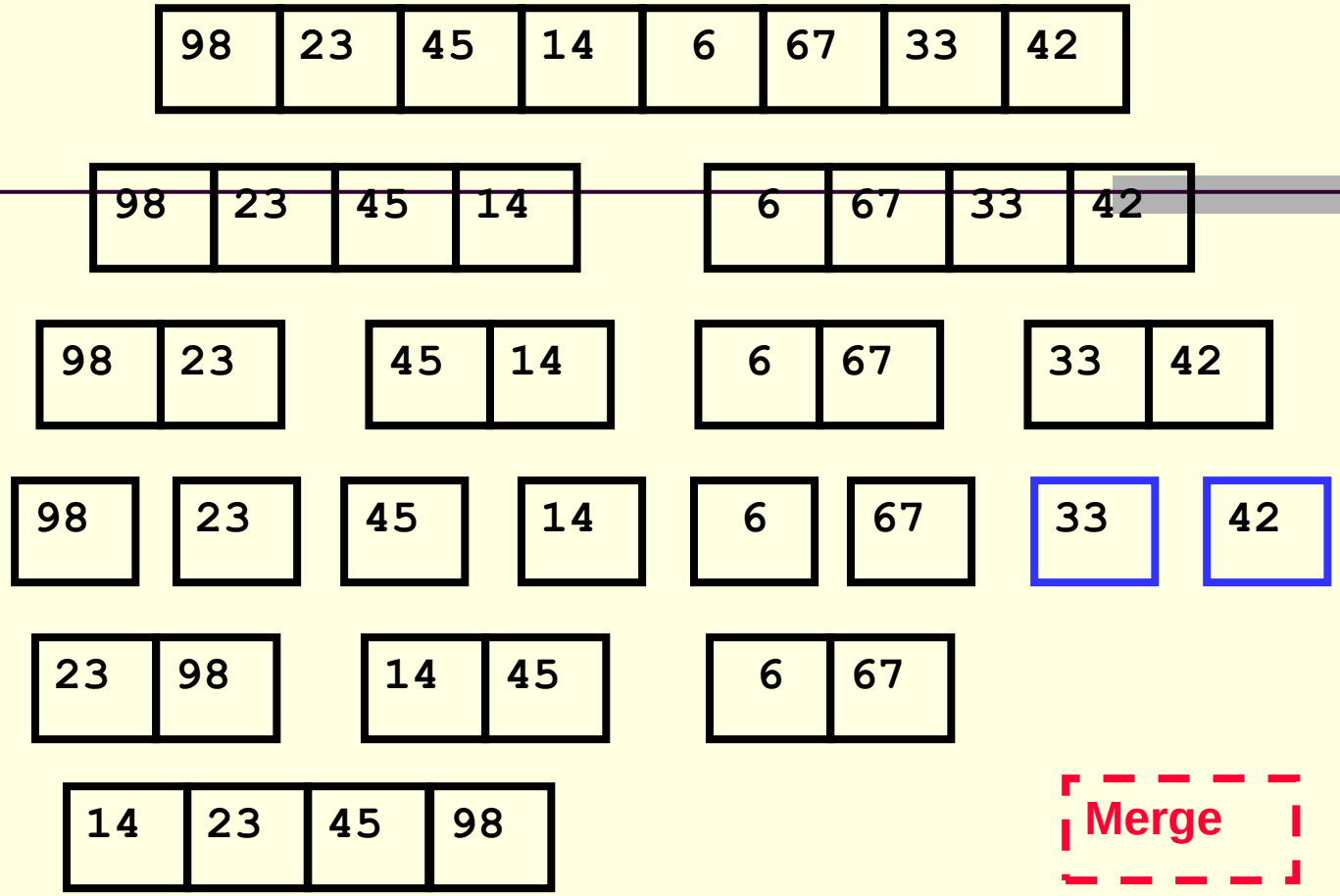

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98

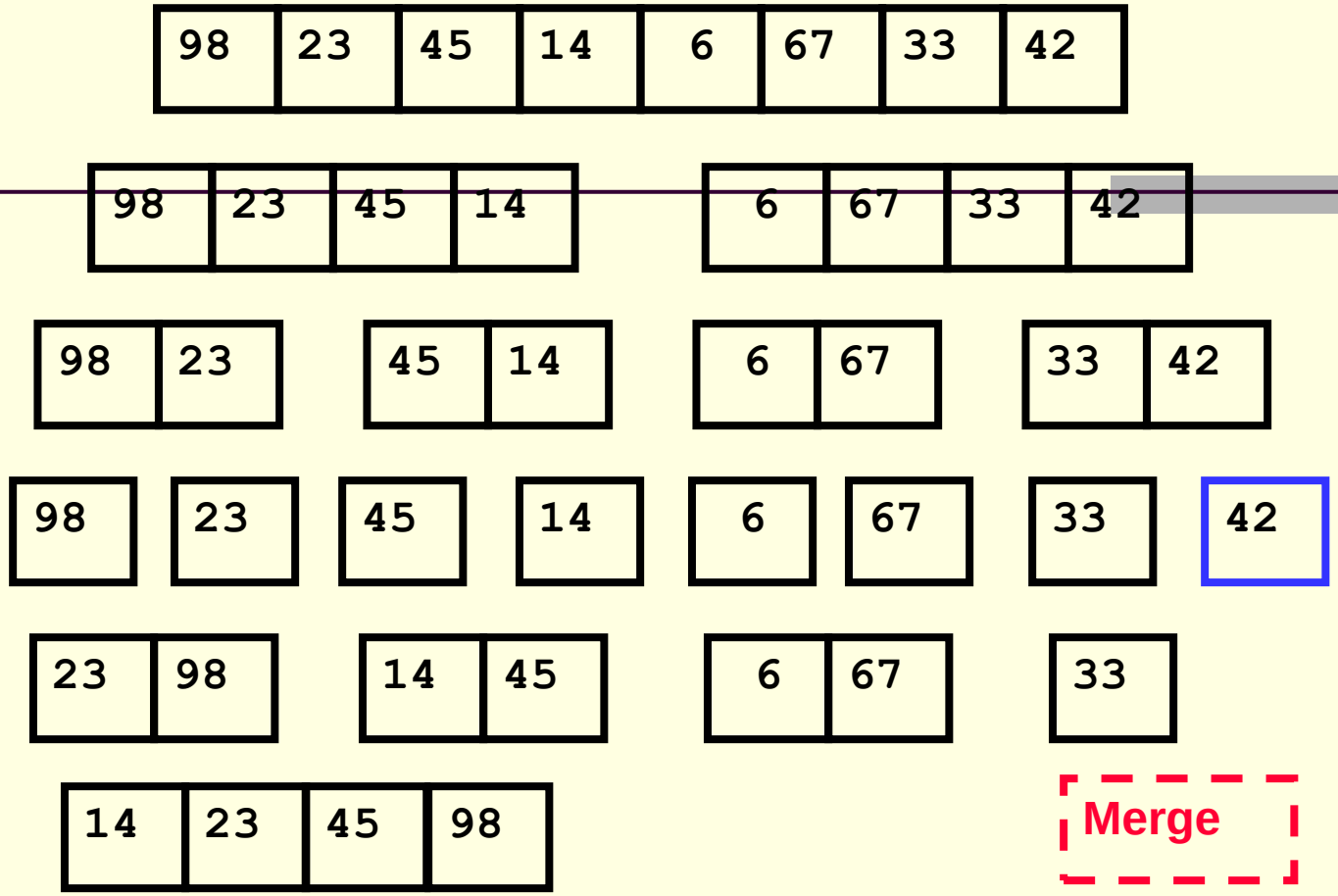

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98
Merge

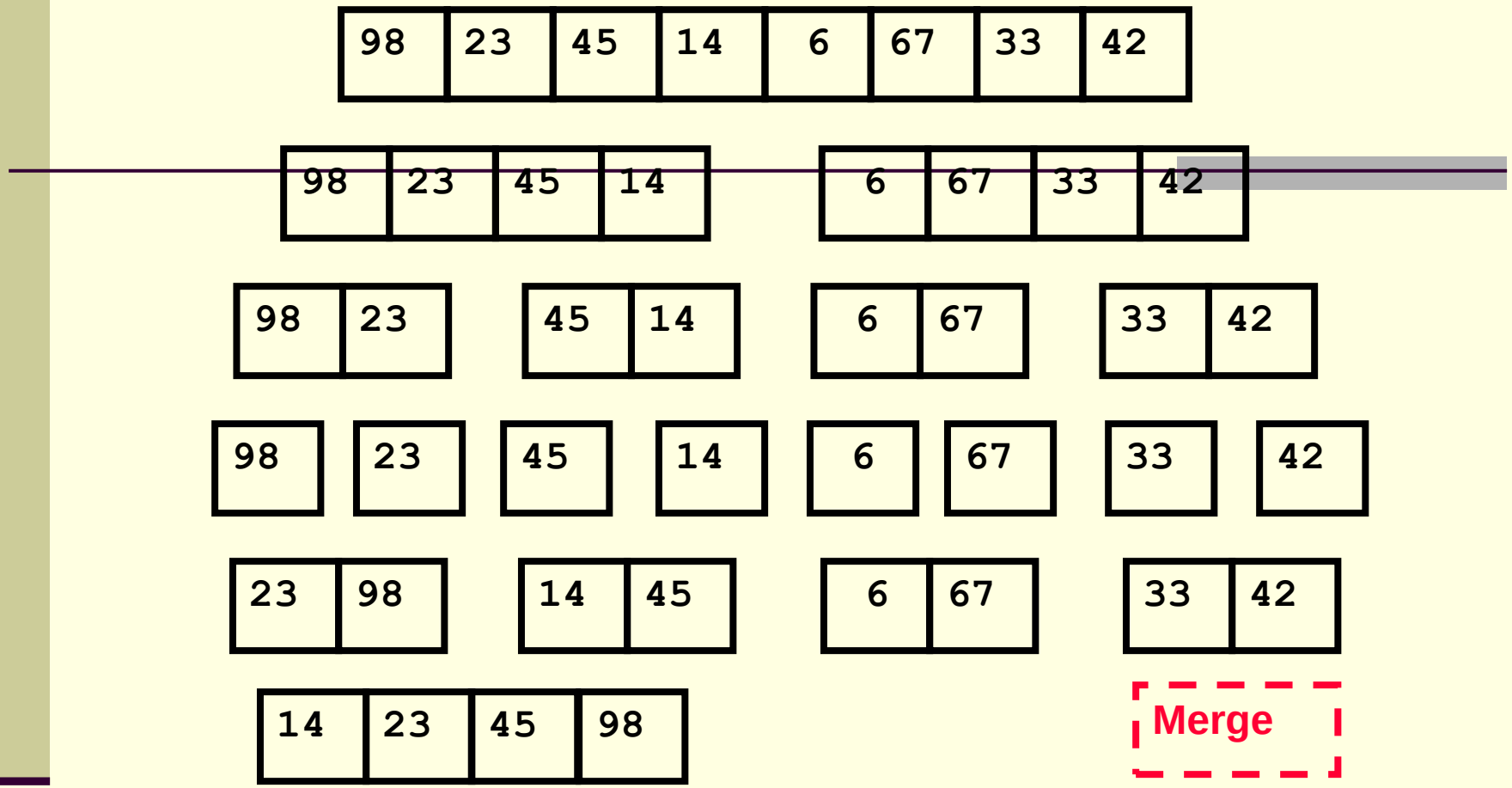

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
6
14 23 45 98
Merge

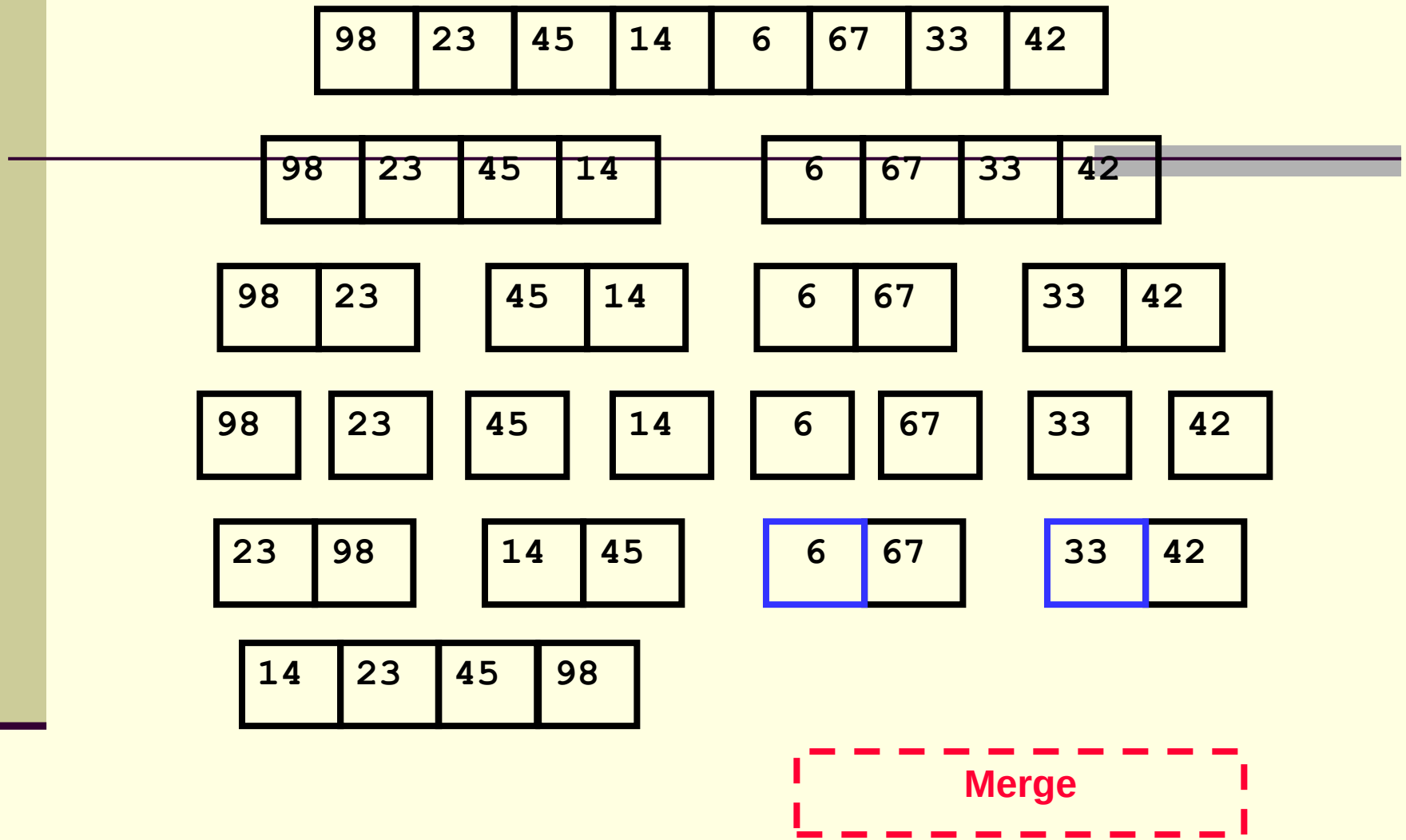

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
6 67
14 23 45 98
Merge

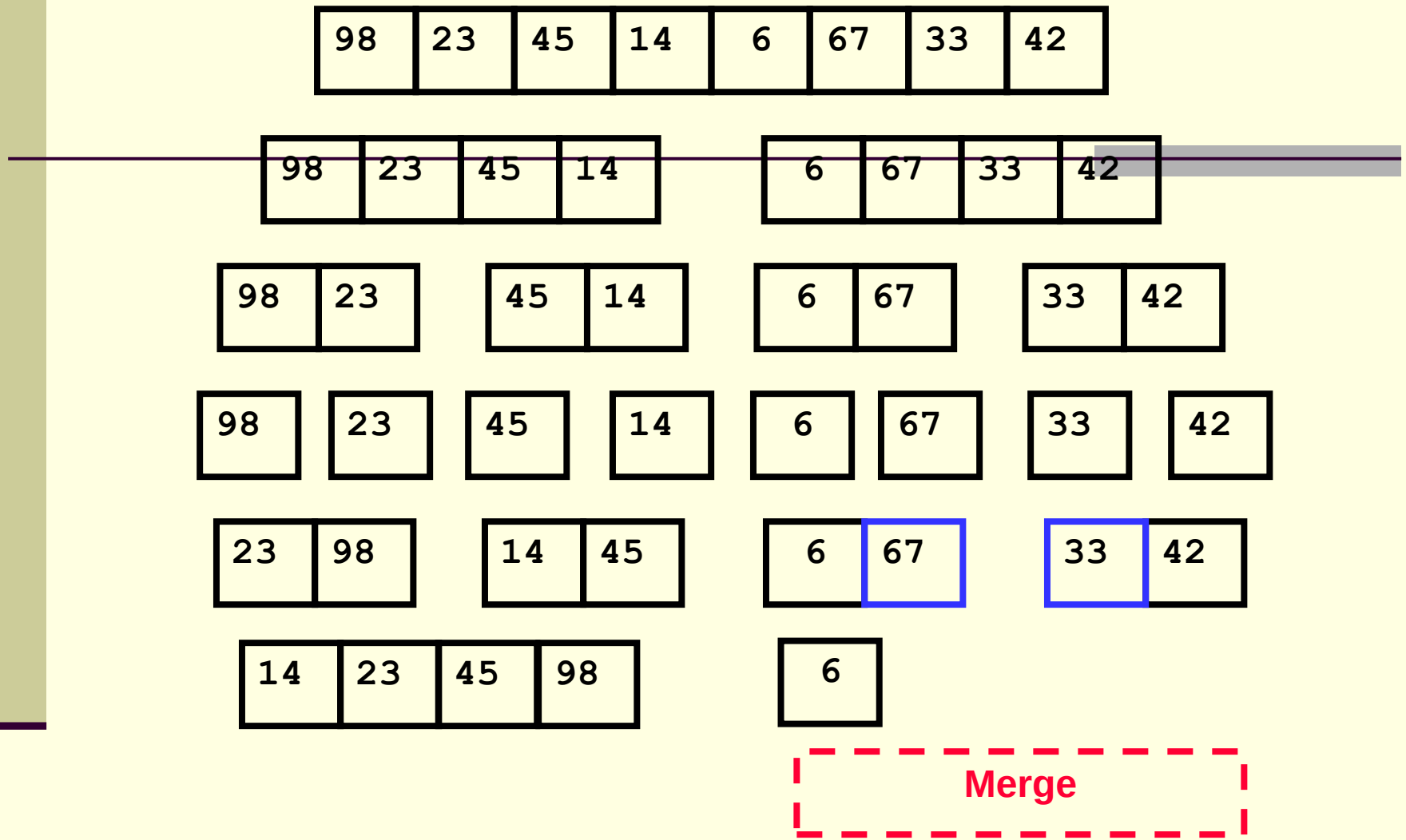

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
14 23 45 98

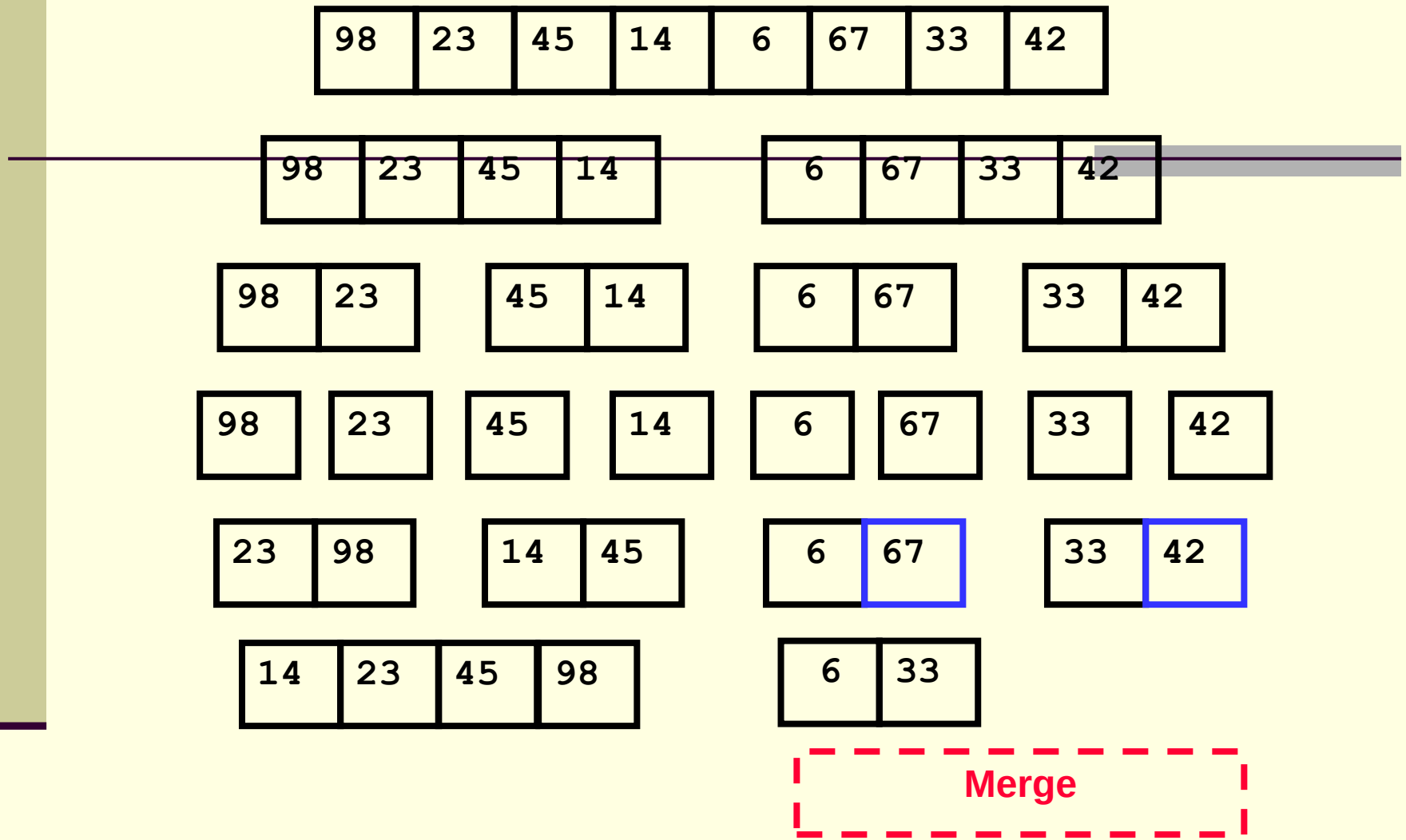

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
14 23 45 98
Merge

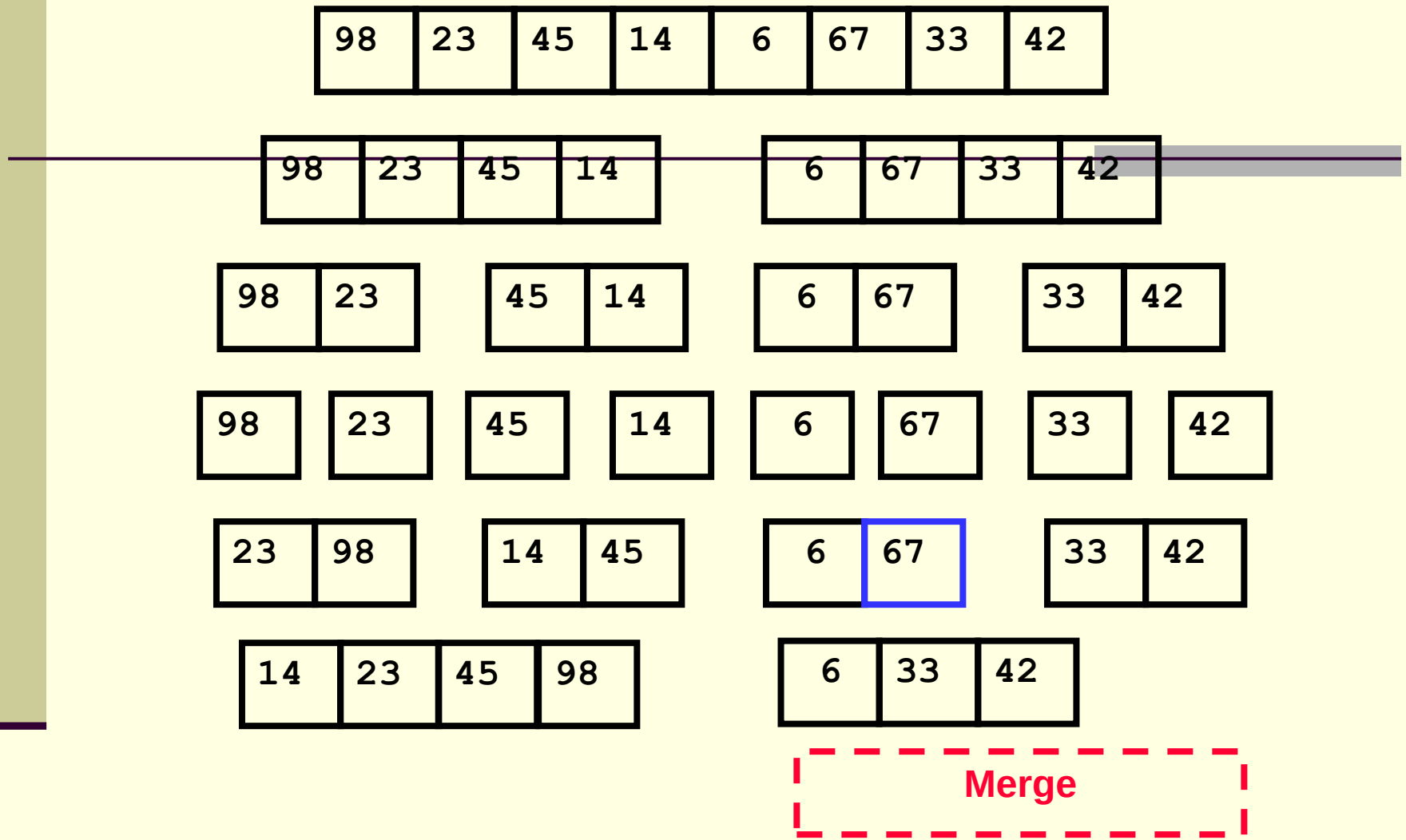

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33
14 23 45 98
Merge

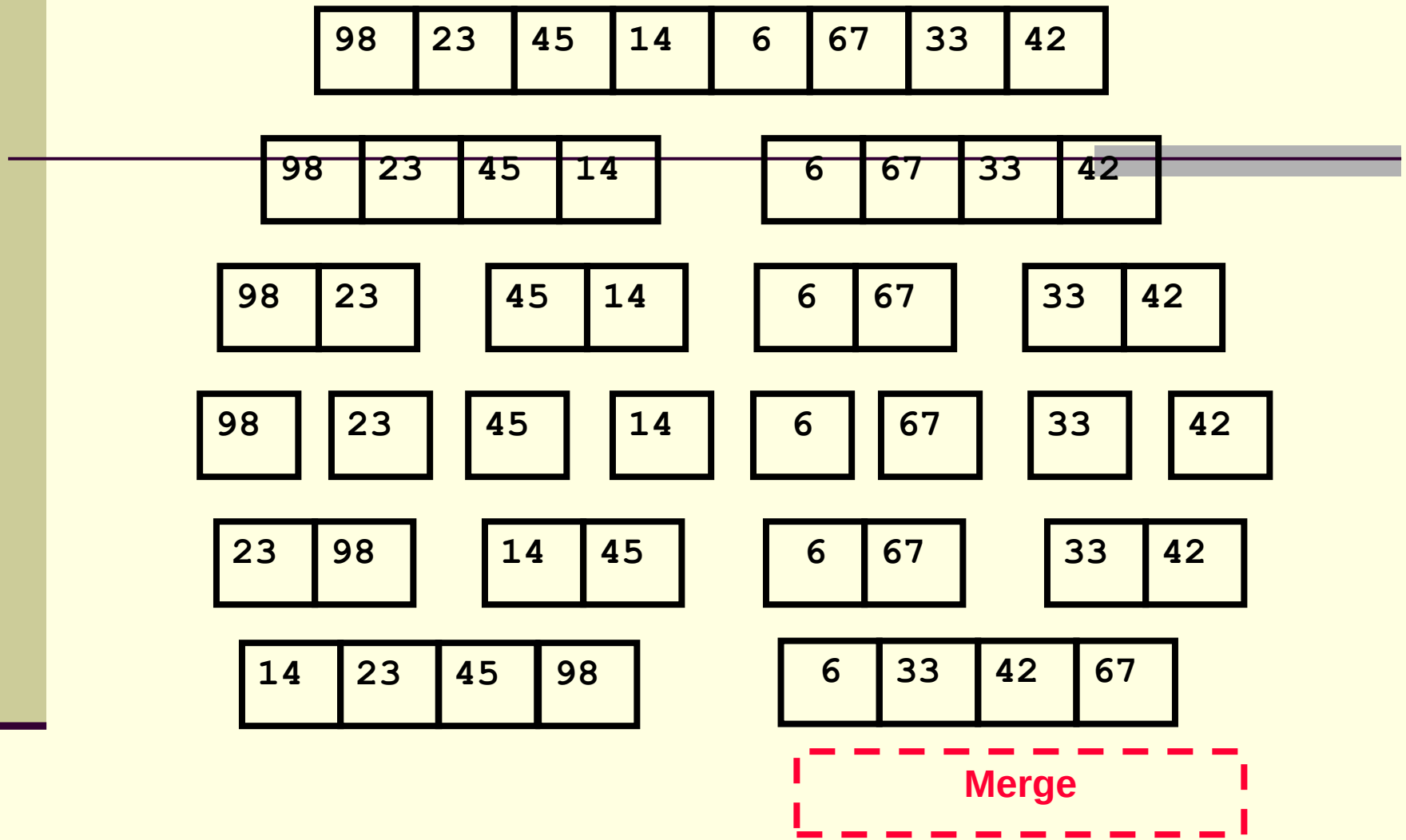

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
Merge

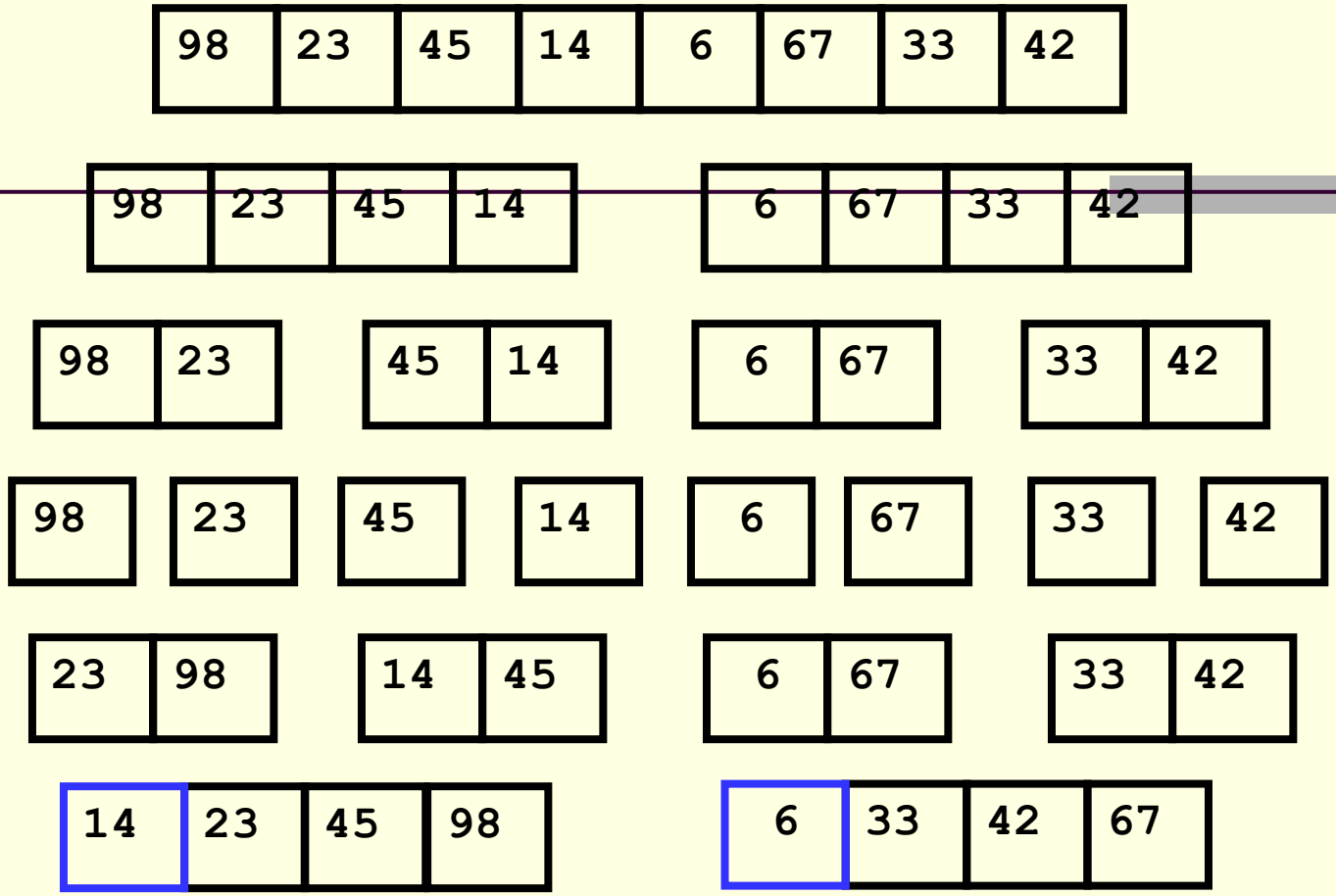

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
Merge

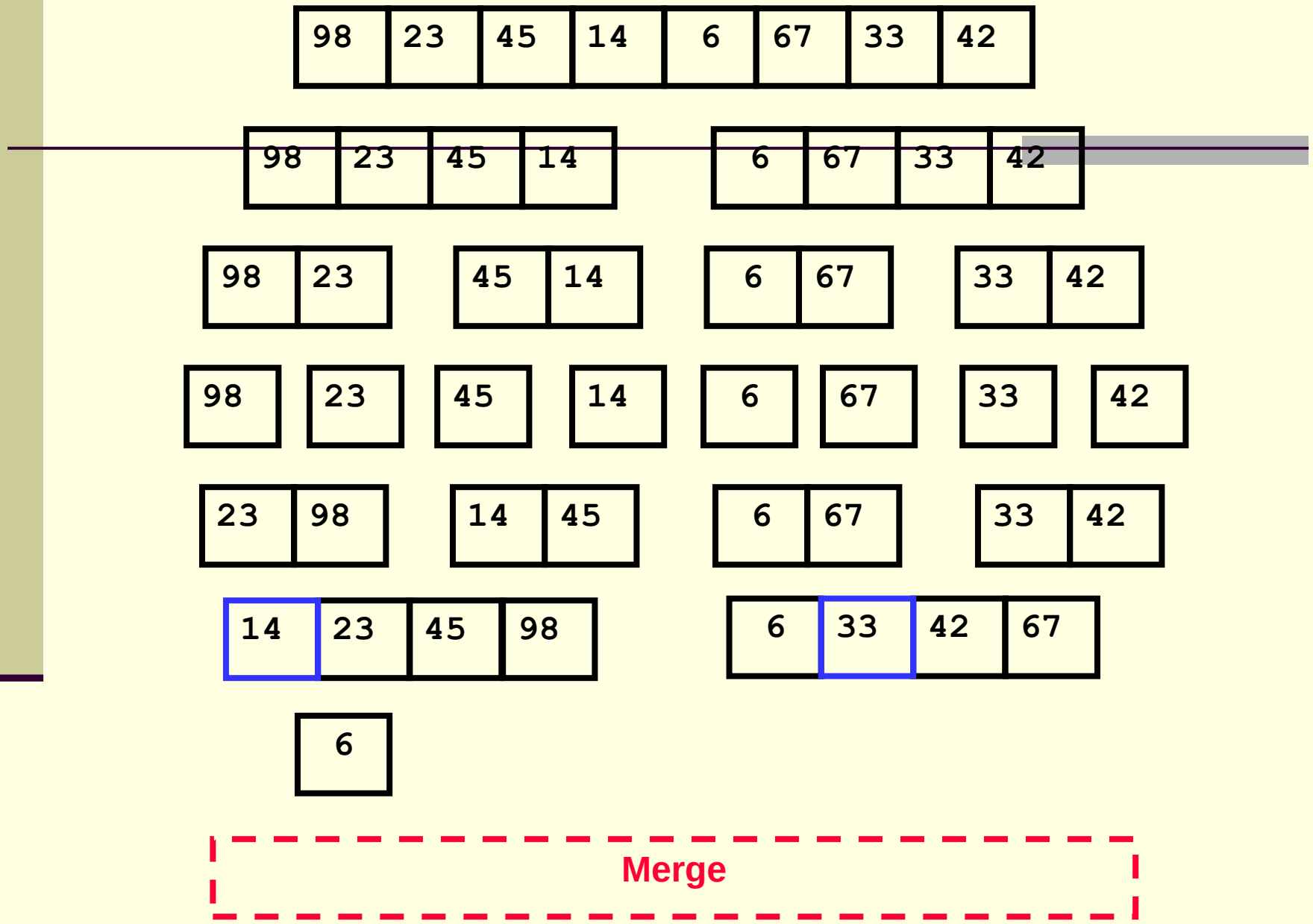

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6
Merge

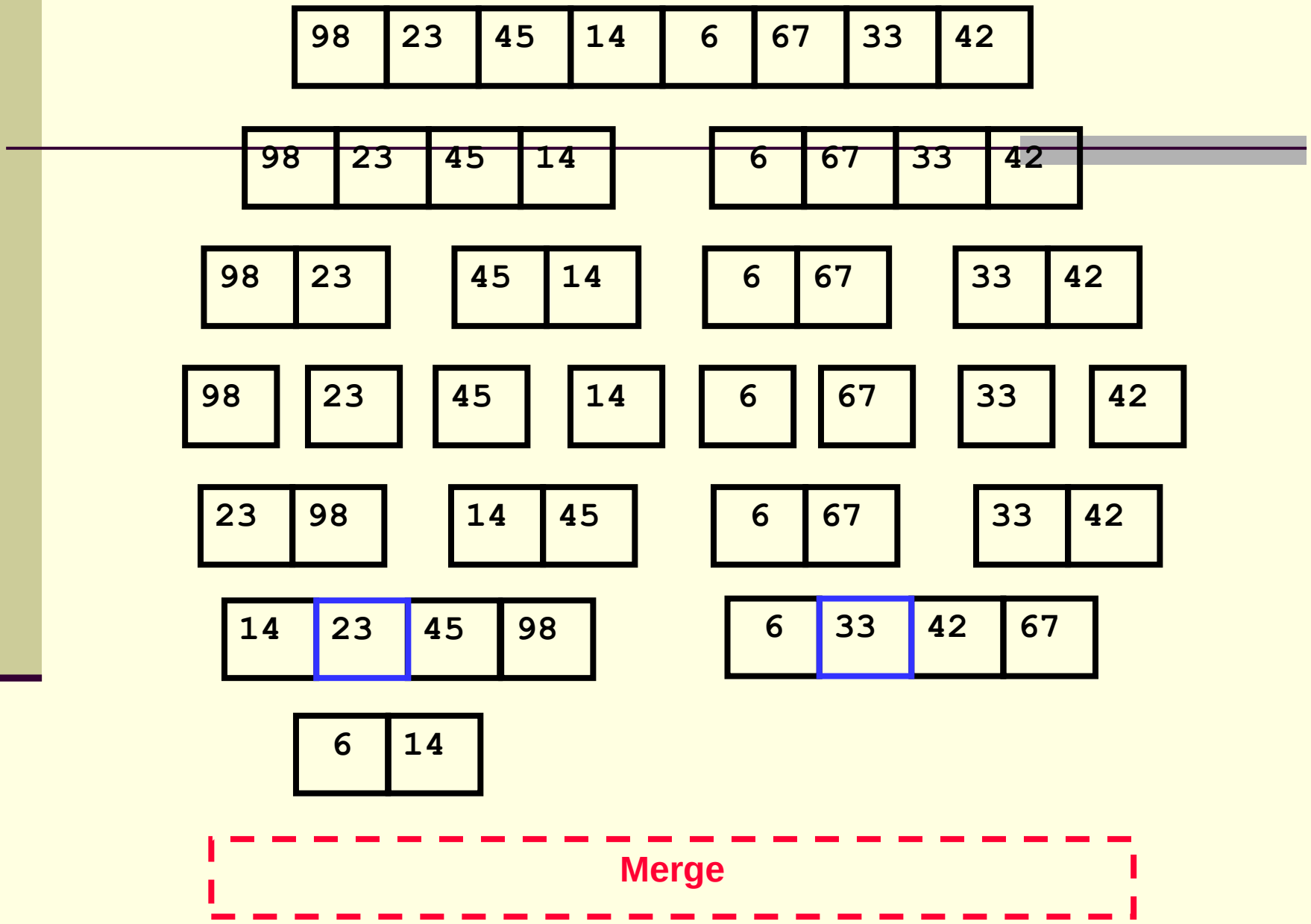

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33
Merge

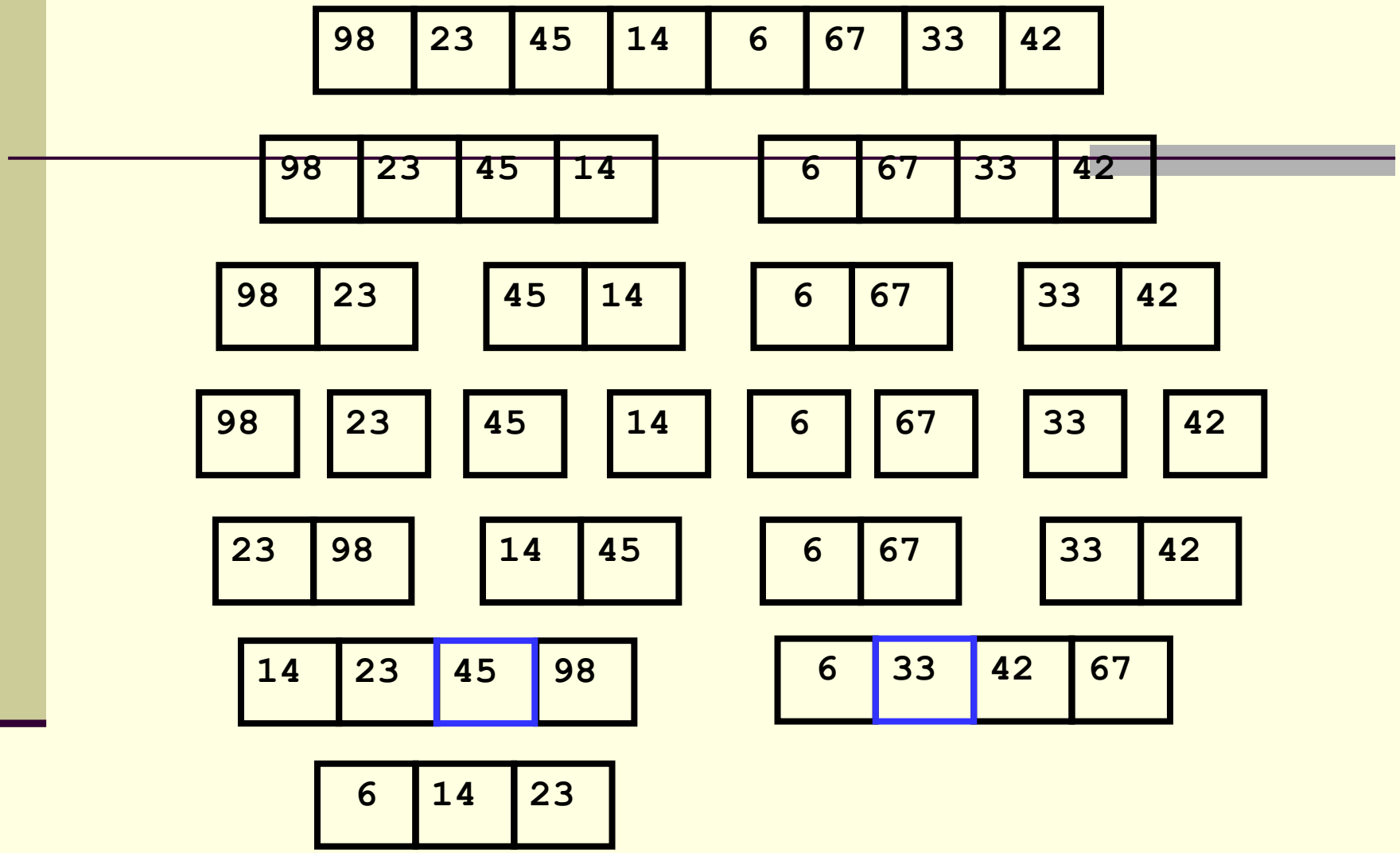

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42
Merge

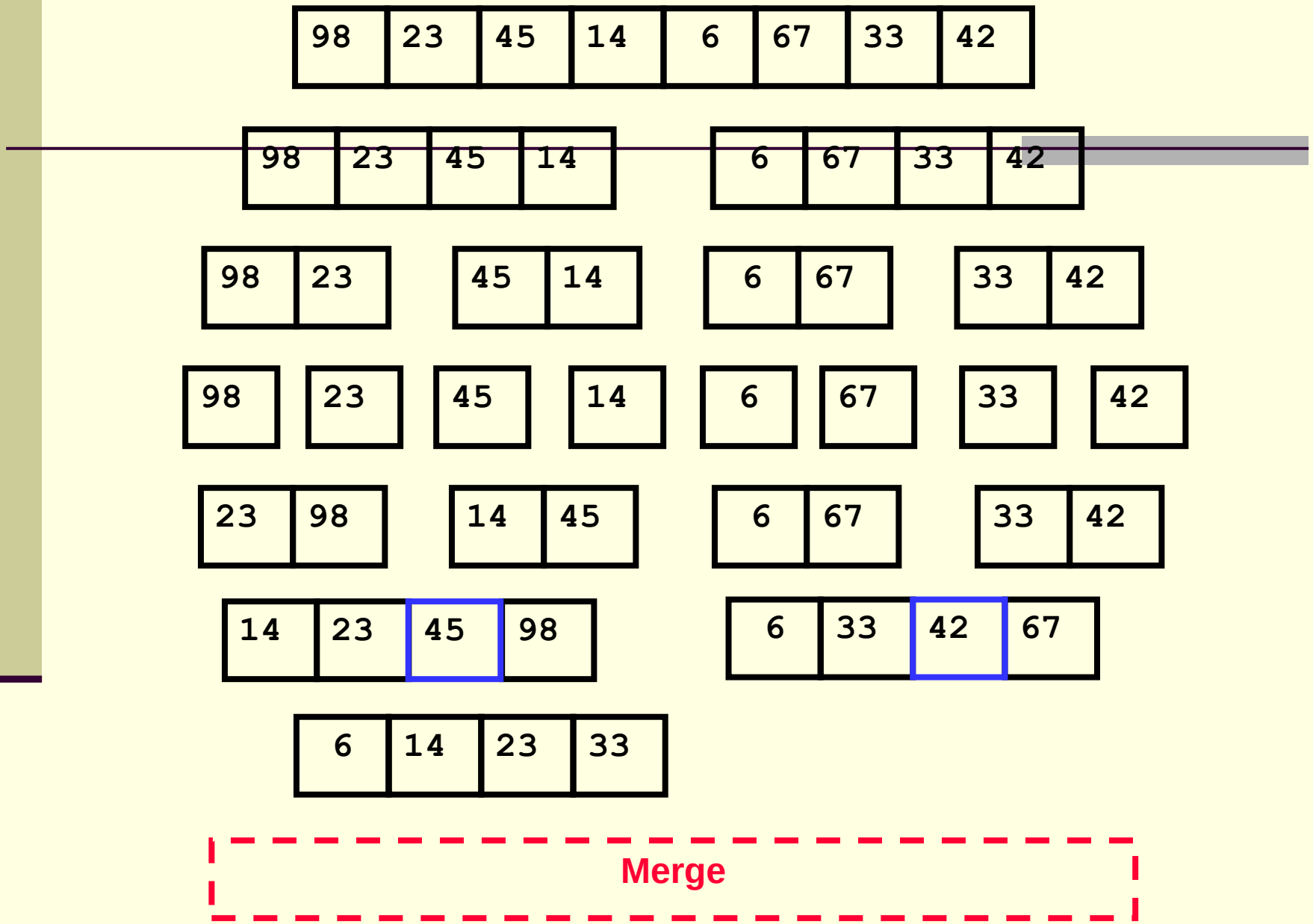

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
Merge

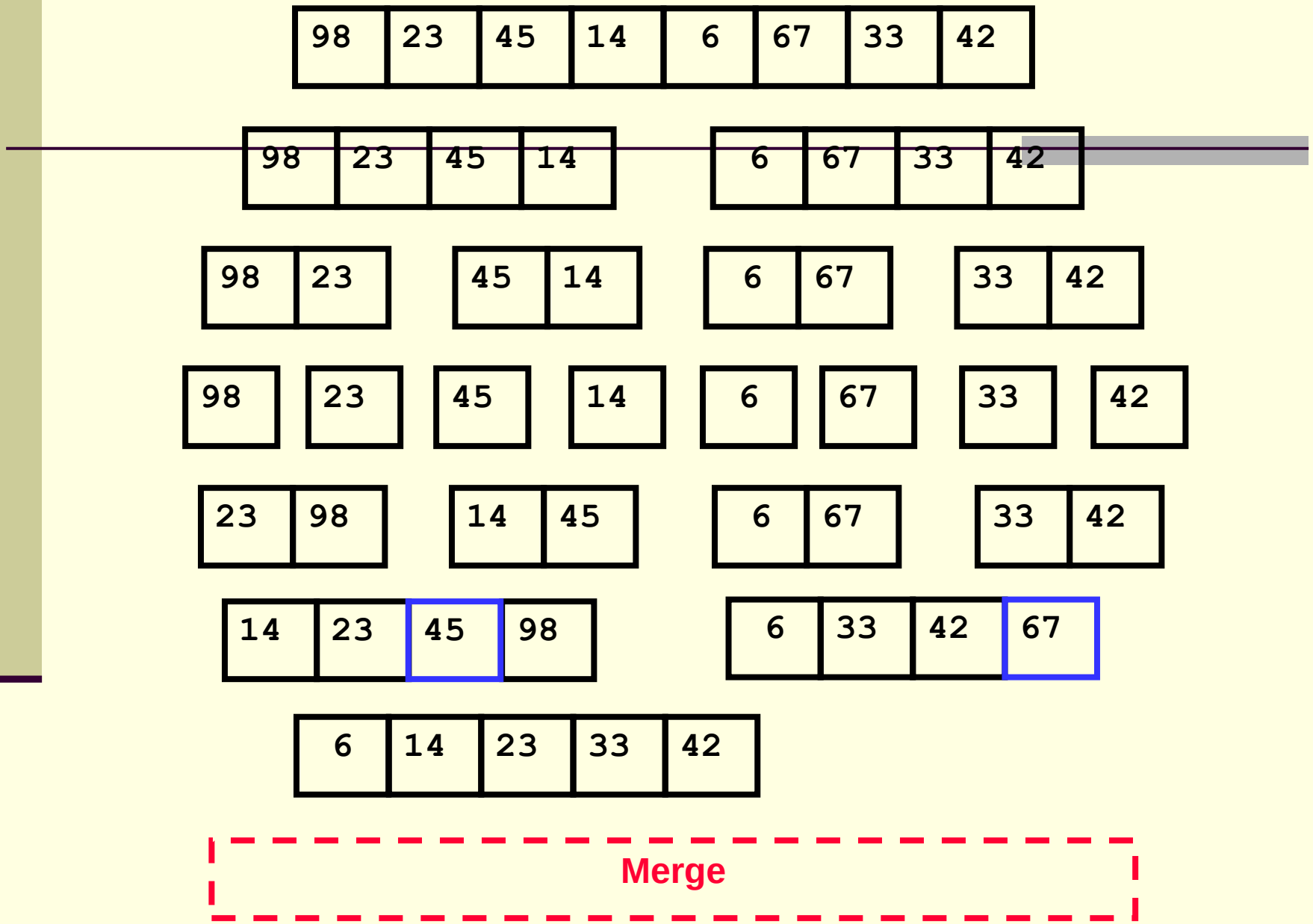

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
Merge

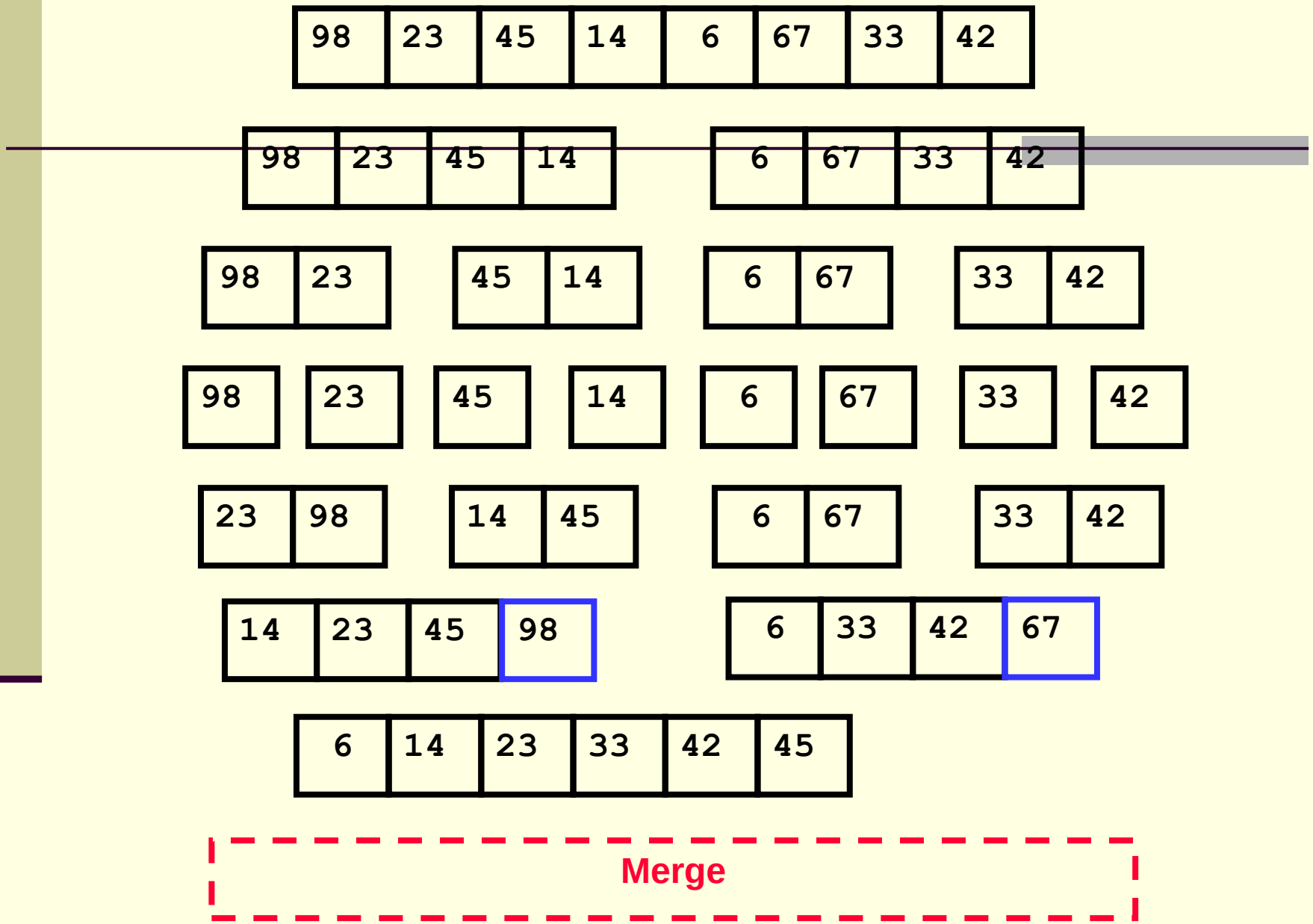

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6
Merge

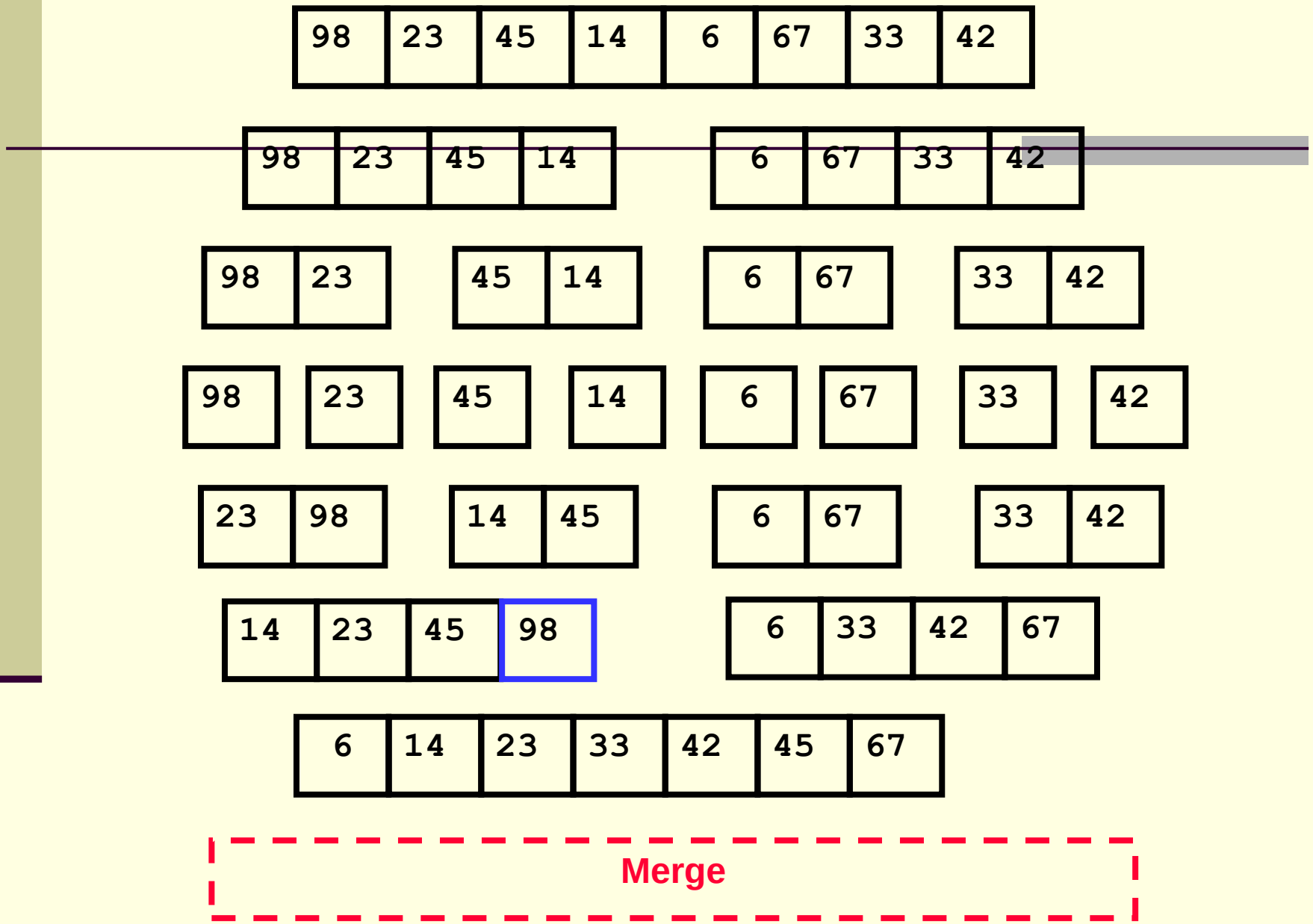

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14
Merge

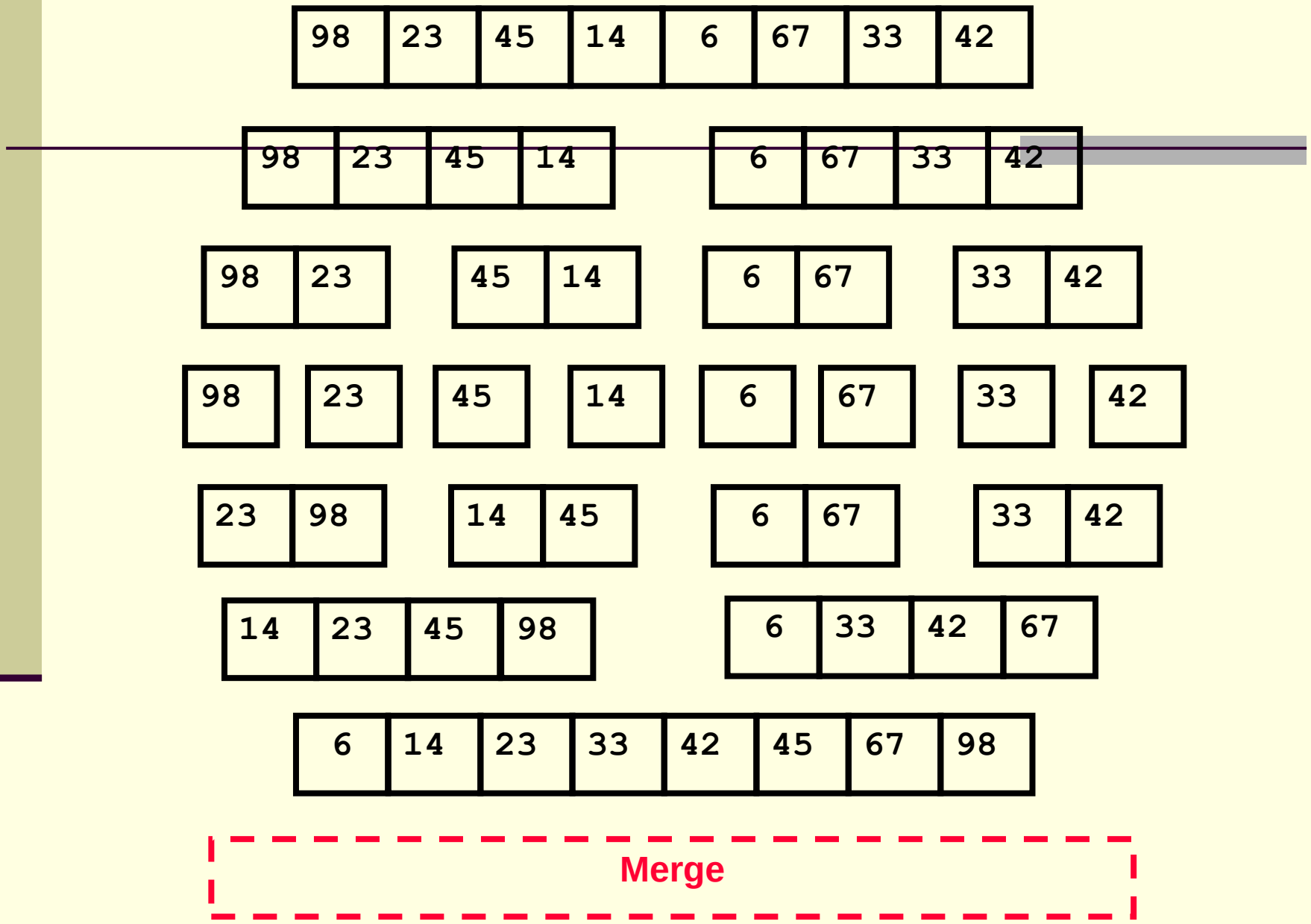

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23
Merge

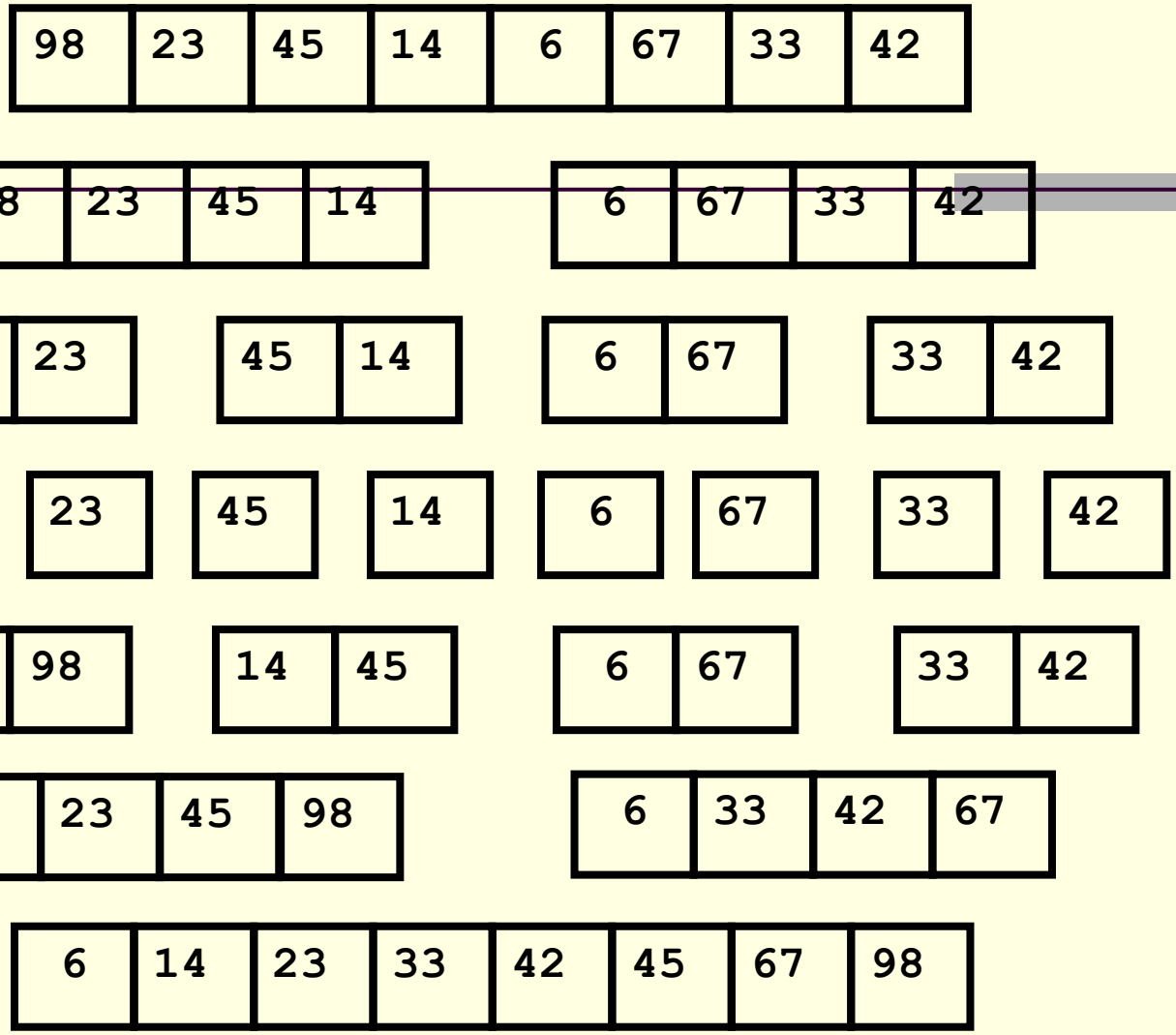

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33
Merge

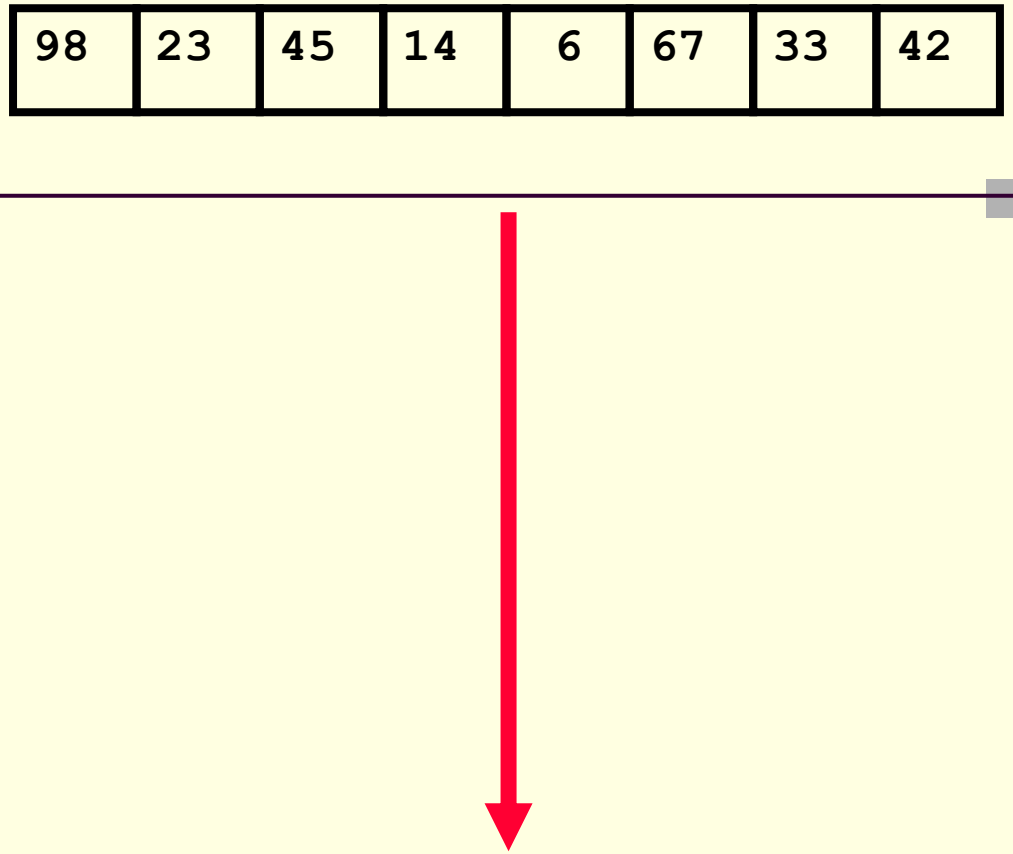

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42
Merge


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

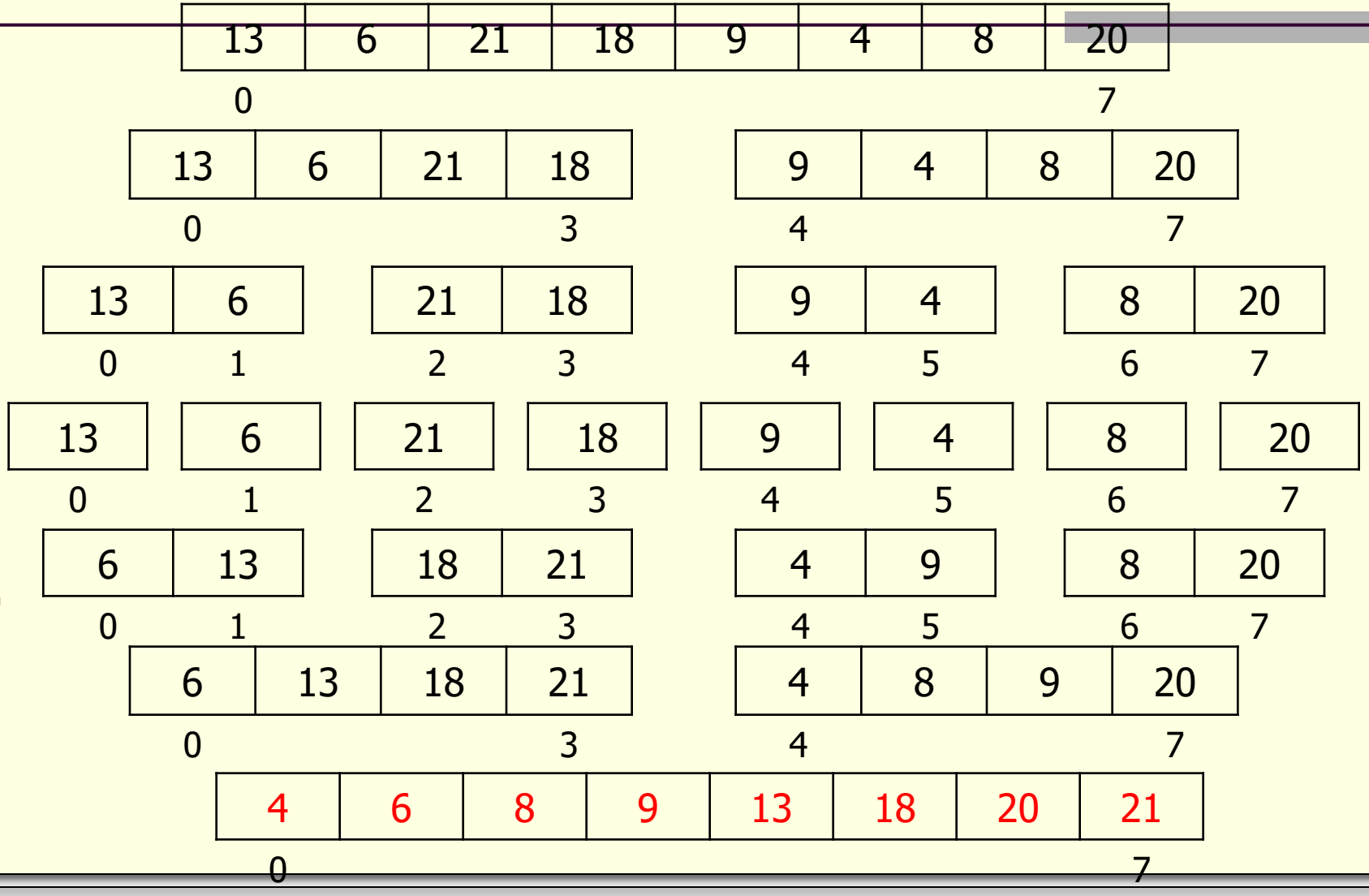

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42 45 67
Merge


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

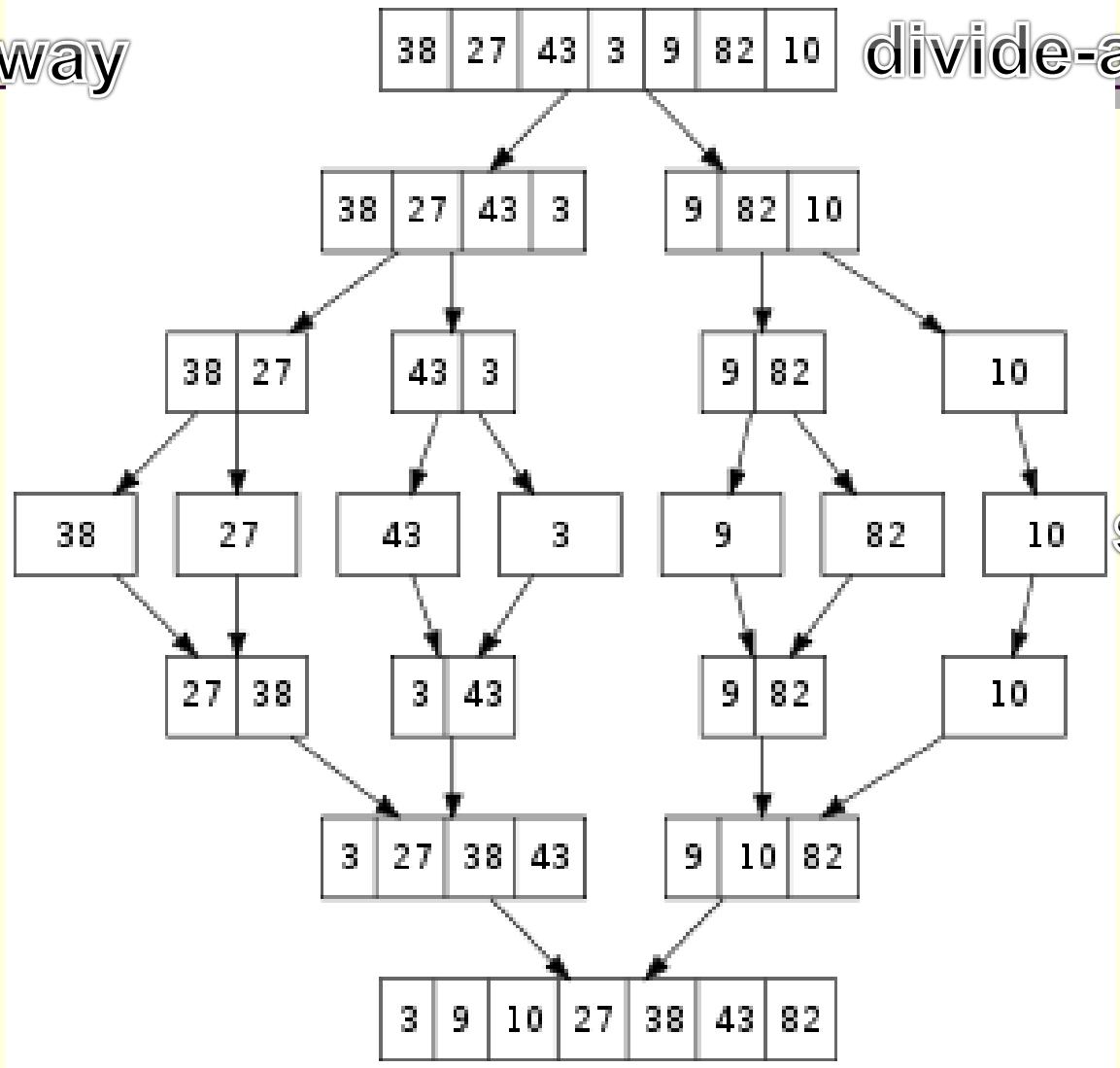

98 23 45 14 6 67 33 42
6 14 23 33 42 45 67 98

Sorting: Merge Sort Example #2



Step-by-step example

two-way



divide-and-conquer

stop dividing

```
void mergesort(int lo, int hi)
 {
        if (lo<hi)
         {
                int m=(lo+hi)/2;
        mergesort(lo, m);
        mergesort(m+1, hi);
        merge(lo, m, hi);
        }
}
```

```
void merge(int lo, int m, int hi)
{
    int i, j, k; // copy both halves of a to auxiliary array b
    for (i=lo; i<=hi; i++)
            b[i]=a[i];

             i=lo;
             j=m+1;
             k=lo; // copy back next-greatest element at each time
    while (i<=m && j<=hi)
            if (b[i]<=b[j])
                        a[k++]=b[i++];
                        else a[k++]=b[j++]; // copy back remaining elements of
                                                first half (if any)
     while (i<=m)
            a[k++]=b[i++];
    while(j<=hi)
            a[k++]=b[j++];
}
```

Sorting: Merge Sort

- Merge Sort **Analysis**
  - Again, here are the steps of Merge Sort:
  1) Merge Sort the first half of the list
  2) Merge Sort the second half of the list
  3) Merge both halves together

  - Let T(n) be the running time of Merge Sort on an input size n
  - Then we have:
    - T(n) = (Time in step 1) + (Time in step 2) + (Time in step 3)

Sorting: Merge Sort

- Merge Sort Analysis
  - T(n): running time of Merge Sort on input size n
  - Therefore, we have:
    - T(n) = (Time in step 1) + (Time in step 2) + (Time in step 3)
  - Notice that Step 1 and Step 2 are sorting problems also
    - **But they are of size n/2…we are halving the input**
  - And the Merge function runs in O(n) time
  - Thus, we get the following equation for T(n)
  - T(n) = T(n/2) + T(n/2) + O(n)
  - T(n) = 2T(n/2) + O(n)

Sorting: Merge Sort

- Merge Sort Analysis
  - T(n) = 2T(n/2) + O(n)
  - For the time being, let's simplify O(n) to just n
  - T(n) = 2T(n/2) + n
  - and we know that T(1) = 1
  - So we now have a Recurrence Relation

Sorting: Merge Sort

- Merge Sort Analysis
  - $T(n) = 2T(n/2) + n$ and $T(1) = 1$
  - So we need to solve this, by removing the T(…)'s from the right hand side
  - Then T(n) will be in its closed form
  - And we can state its Big-O running time
  - We do this in steps
    - We replace n with **n/2** on both sides of the equation
    - We plug the result back in
    - And then we do it again…till a "light goes off" and we see something

Sorting: Merge Sort

- Merge Sort Analysis
  - $T(n) = 2T(n/2) + n$ and $T(1) = 1$
  - Do you know what $T(n/2)$ equals
    - Does it equal 2,125 operations? We don't know!
  - So we need to develop an equation for $T(n/2)$
  - How?
  - Take the original equation shown above
  - **Wherever you see an 'n', substitute with 'n/2'**
  - $T(n/2) = 2T(n/4) + n/2$
  - So now we have an equation for $T(n/2)$

Sorting: Merge Sort

- Merge Sort Analysis
  - $T(n) = 2T(n/2) + n$ and $T(1) = 1$
  - $T(n/2) = 2T(n/4) + n/2$
  - So now we have an equation for $T(n/2)$
    - We can take this equation and substitute it back into the original equation
  - $T(n) = 2T(n/2) + n = 2[2T(n/4) + n/2] + n$
    - now simplify
  - $T(n) = 4T(n/4) + 2n$
    - Same thing here: do you know what $T(n/4)$ equals?
    - No we don't! So we need to develop an eqn for $T(n/4)$

Sorting: Merge Sort

- Merge Sort Analysis
  - $T(n) = 2T(n/2) + n$ and $T(1) = 1$
  - $T(n/2) = 2T(n/4) + n/2$
  - $T(n) = 4T(n/4) + 2n$
    - Same thing here: do you know what T(n/4) equals?
    - No we don't! So we need to develop an eqn for T(n/4)
    - Take the eqn above and again substitute 'n/2' for 'n'
  - $T(n/4) = 2T(n/8) + n/4$
  - So now we have an equation for T(n/4)
    - We can take this equation and substitute it back the equation that we currently have in terms of T(n/4)

Sorting: Merge Sort

- Merge Sort Analysis
  - $T(n) = 2T(n/2) + n$ and $T(1) = 1$
  - $T(n/2) = 2T(n/4) + n/2$
  - $T(n) = 4T(n/4) + 2n$
  - $T(n/4) = 2T(n/8) + n/4$
  - So now we have an equation for $T(n/4)$
    - We can take this equation and substitute it back the equation that we currently have in terms of $T(n/4)$
  - $T(n) = 4T(n/4) + 2n = 4[2T(n/8) + n/4] + 2n$
    - Simplify a bit
  - $T(n) = 8T(n/8) + 3n$

Sorting: Merge Sort

- Merge Sort Analysis
  - So now we have three equations for T(n):
  - $T(n) = 2T(n/2) + n$ ← 1st step of recursion
  - $T(n) = 4T(n/4) + 2n$ ← 2nd step of recursion
  - $T(n) = 8T(n/8) + 3n$ ← 3rd step of recursion

  - So on the kth step/stage of the recursion, we get a generalized recurrence relation:
  - $T(n) = 2^kT(n/2^k) + kn$ ← k^{th} step of recursion

Sorting: Merge Sort

- Merge Sort Analysis
  - So on the kth step/stage of the recursion, we get a generalized recurrence relation:
  - $T(n) = 2^kT(n/2^k) + kn$
  - We need to get rid of the T(…)'s on the right side
  - Remember, we know T(1) = 1
  - So we make a substitution:
    - Let $n = 2^k$
    - and also solve for k
    - $k = \log_2 n$
  - Plug these back in…

Sorting: Merge Sort

- Merge Sort Analysis
  - So on the kth step/stage of the recursion, we get a generalized recurrence relation:
  - $T(n) = 2^kT(n/2^k) + kn$
    - Let $n = 2^k$
    - and also solve for k
    - $k = \log_2 n$
  - Plug these back in…
  - $T(n) = 2^{\log_2 n}T(n/n) + (\log_2 n)n$
  - $T(n) = n*T(1) + n\log n = n + n*\log n$
  - So Merge Sort runs in $O(n*\log n)$ time

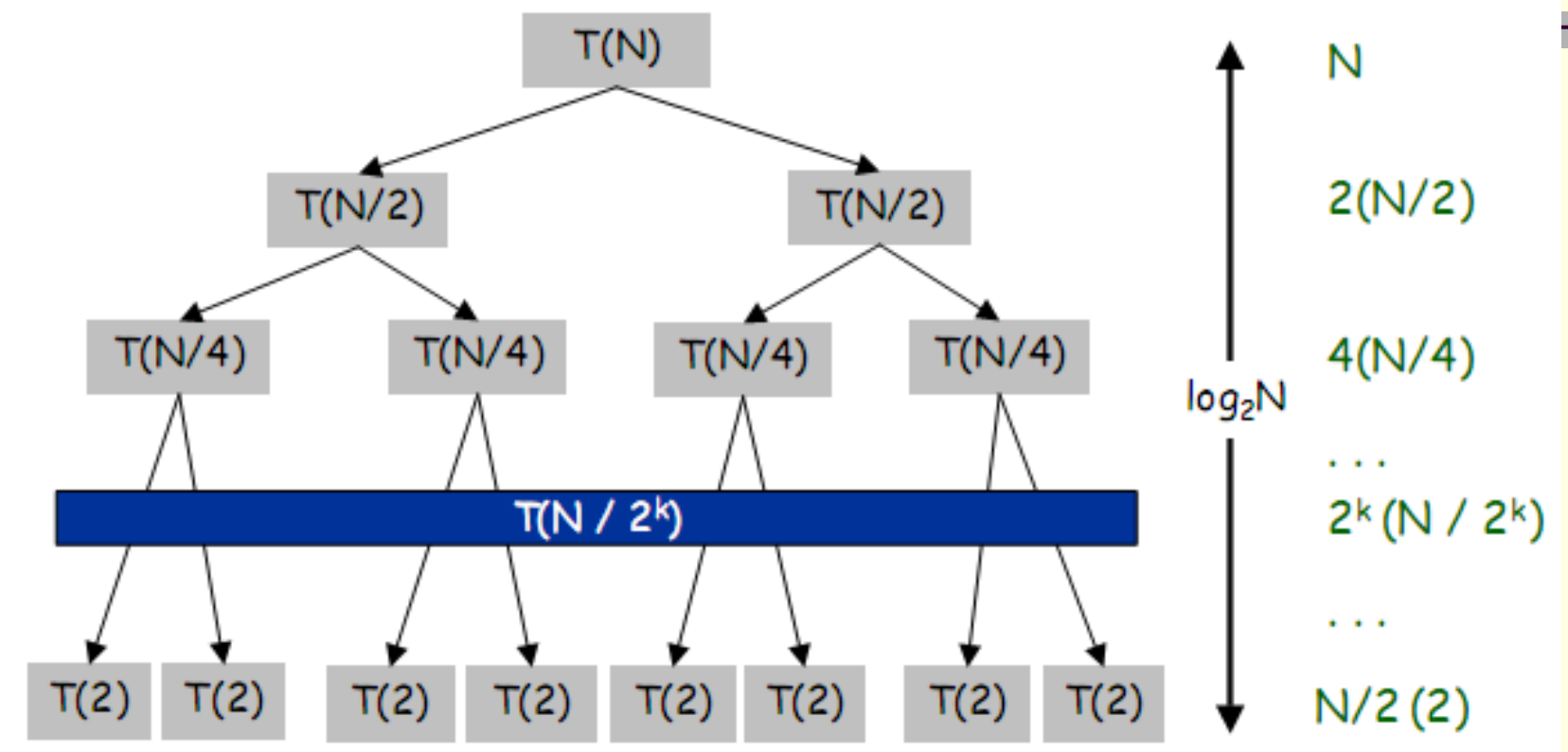
O(nlogn) proof by recursion tree

16

8

4

2



If the number of items N = 16, there are $\log_2 16 = 4$ levels.

NlogN

Sorting: Merge Sort

- Merge Sort Summary
  - Avoids all the unnecessary swaps of n^2 sorts
  - Uses recursion to split up a list until we get to "lists" of 1 or 0 elements
  - Uses a Merge function to merge ("sort") these smaller lists into larger lists
  - Is MUCH faster than n^2 sorts
  - Merge Sort runs in O(nlogn) time

Quicksort

Sorting: Quick Sort

- Quick Sort
  - Most common sort used in practice
  - Why?
    - cuz it is usually the quickest in practice!
  - Quick Sort uses two main ideas to achieve this efficiency:
  1) The idea of making partitions
  2) Recursion

  - Let's look at the partition concept…

Sorting: Quick Sort

- Quick Sort – Partition
  - **A partition works as follows:**
  - Given an array of n elements
    - You must manually select an element in the array to partition by
    - You must then compare ALL the remaining elements against this element
    - If they are greater,
      - Put them to the “right” of the partition element
    - If they are less,
      - Put them to the “left” of the partition element

Sorting: Quick Sort

- Quick Sort – Partition
  - **A partition works as follows:**
    - Once the partition is complete, what can we say about the position of the partition element?
    - We can say (we KNOW) that **the partition element is in its CORRECTLY sorted location**
    - In fact, after you partition the array, you are left with:
      - all the elements to the left of the partition element, in the array, that still need to be sorted
      - all the elements to the right of the partition element, in the array, that still need to be sorted
    - And if you sort those two sides, the entire array will be sorted!

Sorting: Quick Sort

- Quick Sort
  - Partition:
    - Essentially breaks down the sorting problem into two smaller sorting problems
      - …what does that sound like?
  - Code for Quick Sort (at a real general level):
    1) Partition the array with respect to a random element
    2) Sort the left part of the array using Quick Sort
    3) Sort the right part of the array using Quick Sort
  - Notice there is no “merge” step like in Merge Sort
    - at the end, all elements are already in their proper order

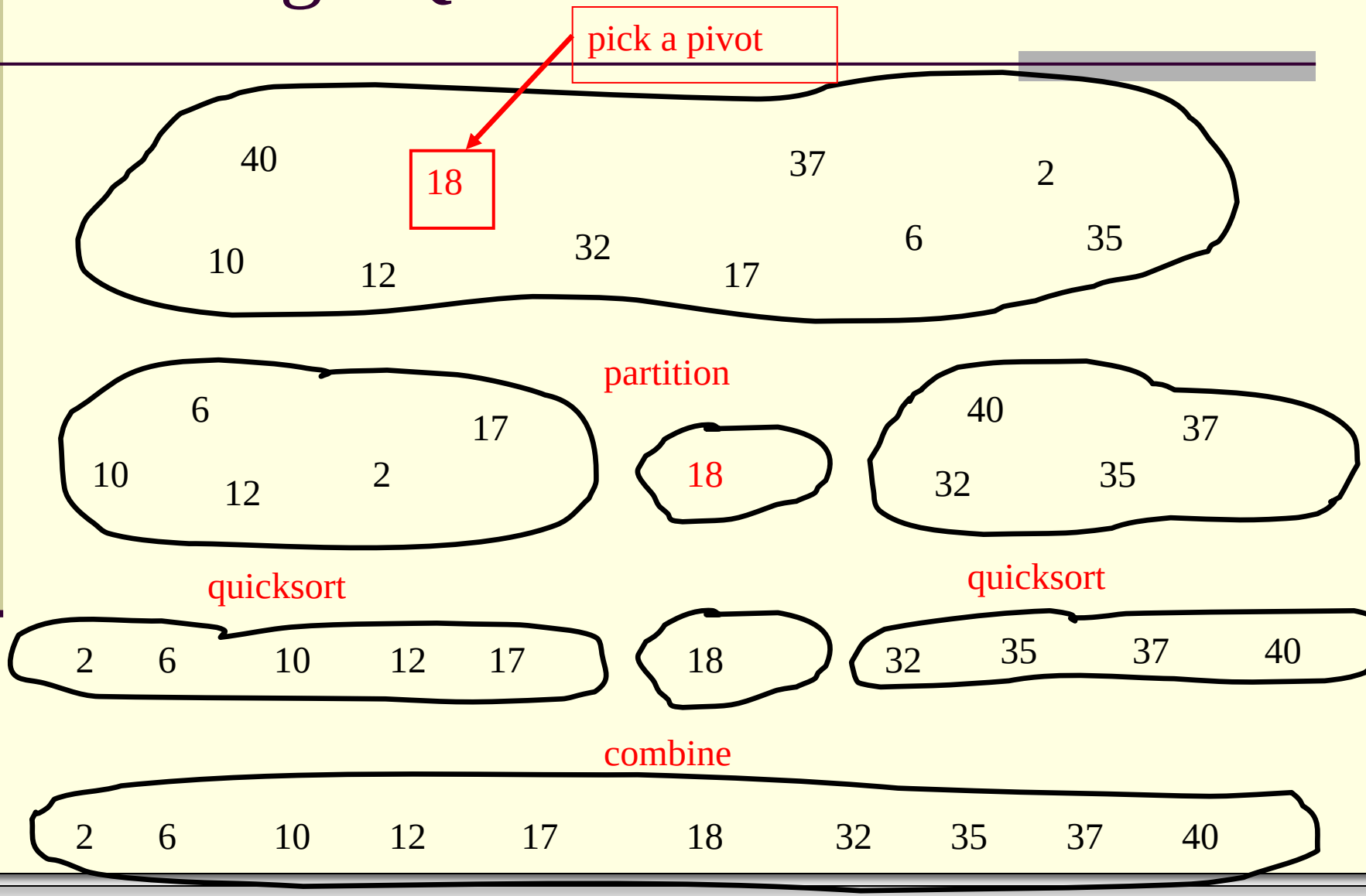
Sorting: Quick Sort

- Quick Sort
  - Code for Quick Sort (at a real general level):
    1) Partition the array with respect to a random element
    2) Sort the left part of the array using Quick Sort
    3) Sort the right part of the array using Quick Sort
  - Quick Sort is a recursive algorithm:
    - We need a base case
      - A case that does NOT make recursive calls
    - Our base case, or terminating condition, will be when we sort an array with only one element
      - We know the array is already sorted!

Sorting: Quick Sort

- Quick Sort
  - Let S be the input set.
  1. If $|S| = 0$ or $|S| = 1$, then return.
  2. Pick an element v in S. Call v the partition element.
  3. Partition $S - \{v\}$ into two disjoint groups:
     - $S_1 = \{x \in S - \{v\} \mid x \leq v\}$
     - $S_2 = \{x \in S - \{v\} \mid x \geq v\}$
  4. Return { quicksort(S_1), v, quicksort(S_2) }

Sorting: Quick Sort



- Quicksort is a divide and conquer algorithm.
- Quicksort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quicksort can then recursively sort the sub-lists.
- The steps are:
  - Pick an element, called a *pivot*, from the list.
  - Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
  - Recursively sort the sub-list of lesser elements and the sub-list of greater elements.
- The base case of the recursion are lists of size zero or one, which never need to be sorted.

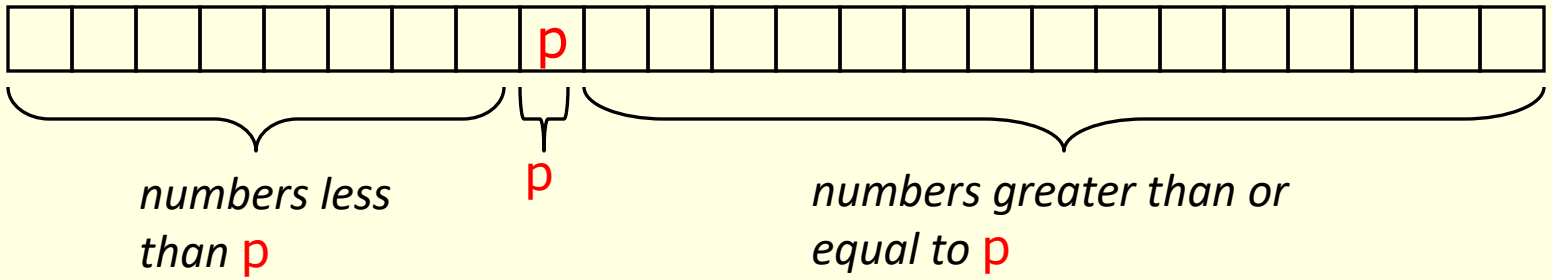
Quicksort I

- To sort a[left...right]:

1. if left < right:
   1.1. Partition a[left...right] such that:
        all a[left...p-1] are less than a[p], and
        all a[p+1...right] are >= a[p]
   1.2. Quicksort a[left...p-1]
   1.3. Quicksort a[p+1...right]
2. Terminate

Partitioning (Quicksort II)

- A key step in the Quicksort algorithm is partitioning the array
  - We choose some (any) number p in the array to use as a pivot
  - We partition the array into three parts:



Partitioning II

- Choose an array value (say, the first) to use as the pivot
- Starting from the left end, find the first element that is greater than or equal to the pivot
- Searching backward from the right end, find the first element that is less than the pivot
- Interchange (swap) these two elements
- Repeat, searching from where we left off, until done

Sorting: Quick Sort

- The idea of “in place”
  - In Computer Science, an “in-place” algorithm is one where the output usually overwrites the input
    - There is more detail, but for our purposes, we stop with that
  - Example:
    - Say we wanted to reverse an array of n items
      - Here is a simple way to do that:

```
function reverse(a[0..n]) {
        allocate b[0..n]
        for i from 0 to n
                b[n - i] = a[i]
        return b
}
```

Sorting: Quick Sort

- The idea of “in place”
  - Example:
    - Say we wanted to reverse an array of n items
      - Here is a simple way to do that:

```
function reverse(a[0..n]) {
        allocate b[0..n]
        for i from 0 to n
                b[n - i] = a[i]
        return b
}
```

    - Unfortunately, this method requires O(n) extra space to create the array b
      - And allocation can be a slow operation

Sorting: Quick Sort

- The idea of “in place”
  - Example:
    - Say we wanted to reverse an array of n items
    - If we no longer need the original array a
    - We can overwrite it using the following in-place algorithm

```
function reverse-in-place(a[0..n])
        for i from 0 to floor(n/2)
                swap(a[i], a[n-i])
```

    - Many Sorting algorithms are in-place algorithms
    - Quick sort is NOT an in-place algorithm
    - BUT, **the Partition algorithm can be in-place**

Sorting: Quick Sort

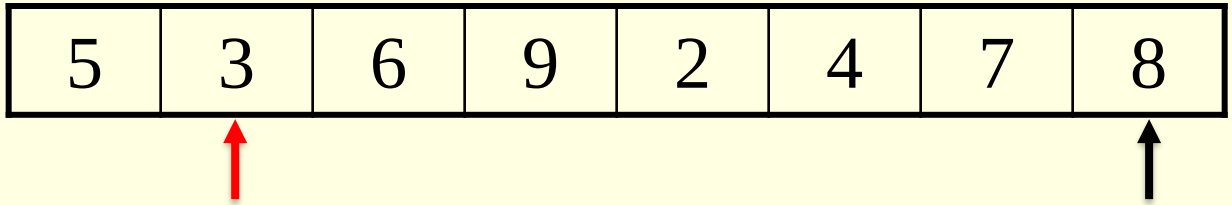
- How to Partition “in-place”
  - Consider the following list of values that we want to partition

5	3	6	9	2	4	7	8

  - Let us assume for the time being that we will partition based on the first element in the array
  - The algorithm will partition these elements “in-place”

Sorting: Quick Sort

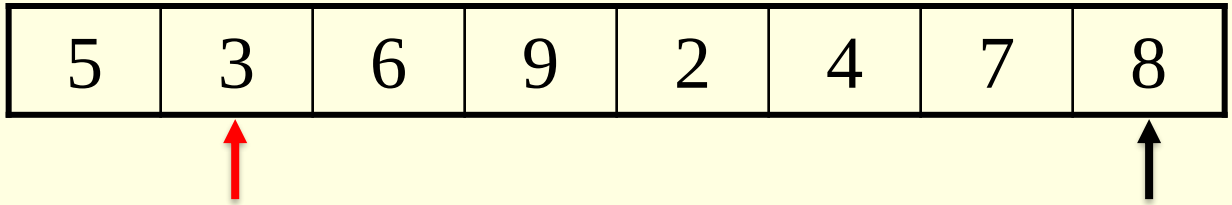
- How to Partition “in-place”



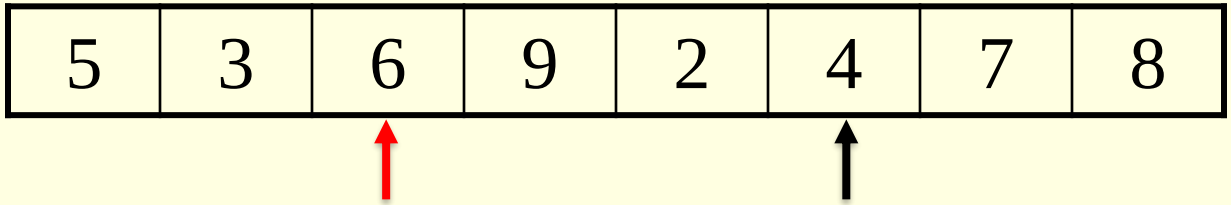
  - Here’s how the partition will work:
    - Start two counters, one at index one and one at index 7
      - The last element in the array
    - Advance the left counter forward until an element greater than the partition element is encountered
    - Advance the right counter backwards until a value less than the pivot is encountered

Sorting: Quick Sort

- How to Partition “in-place”

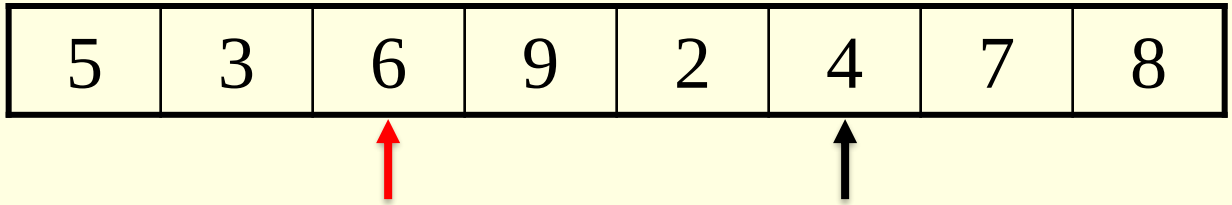


  - After these two steps are performed, we have:

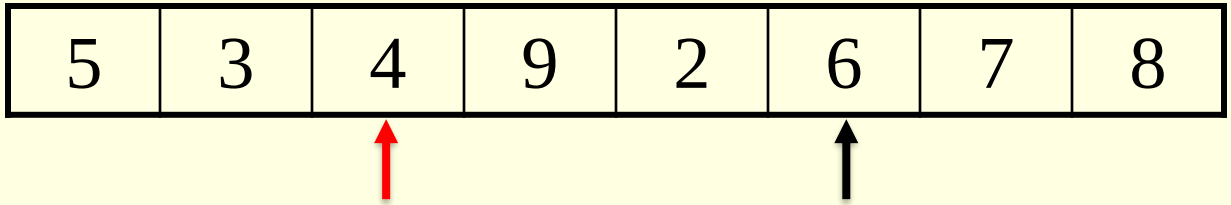


Sorting: Quick Sort

- How to Partition “in-place”

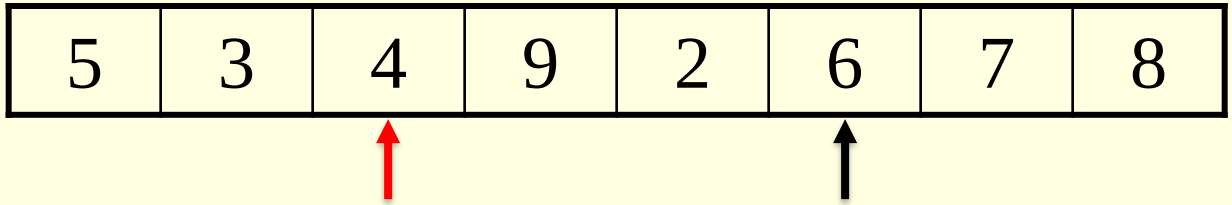


  - We know that these two elements are on the “wrong” side of the array …so SWAP them!

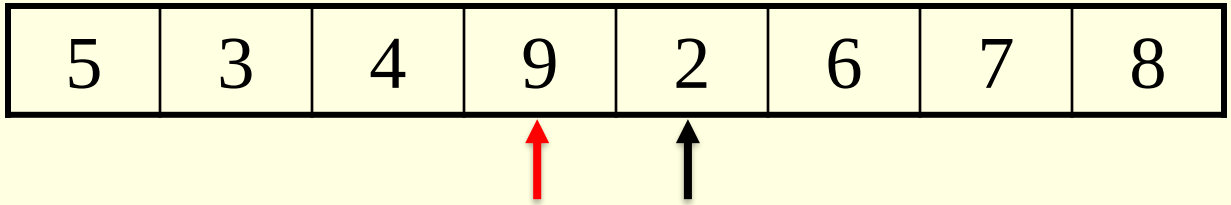


Sorting: Quick Sort

- How to Partition “in-place”

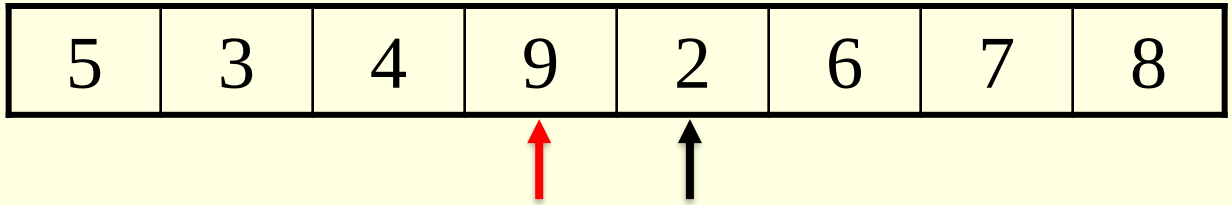


  - Now continue to advance the pointers as before

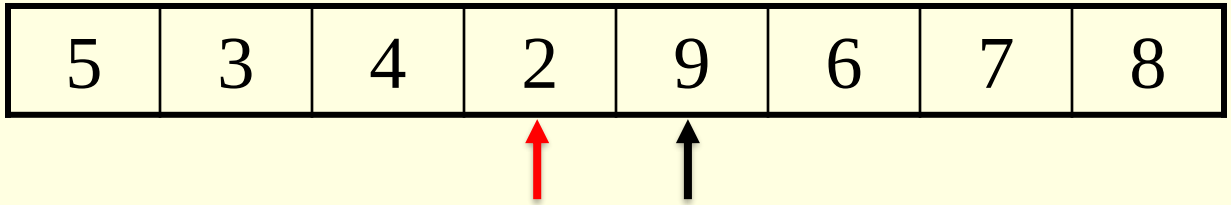


Sorting: Quick Sort

- How to Partition “in-place”



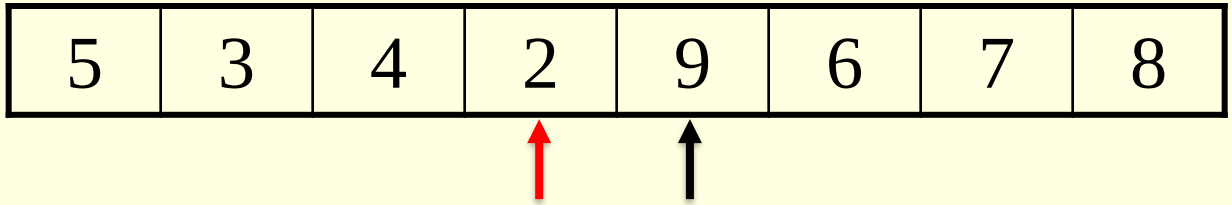
  - Then SWAP as before:



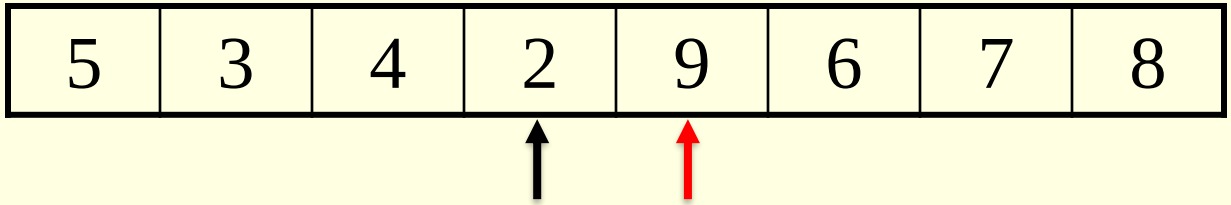
    - At some point, the counters will cross over each other

Sorting: Quick Sort

- How to Partition “in-place”



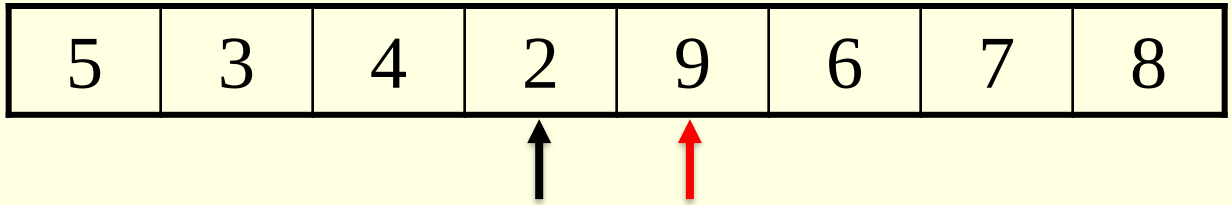
  - Again, advance the pointers as before



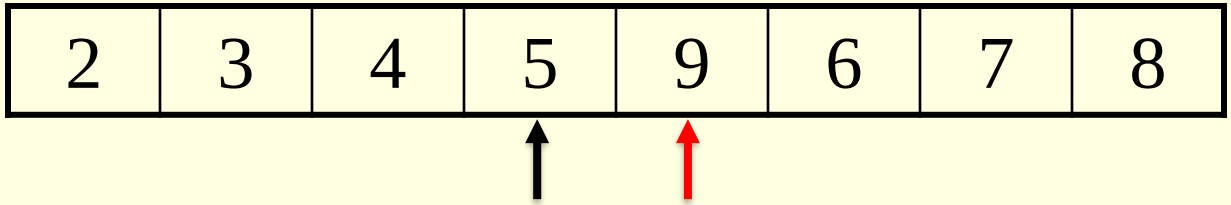
    - So we see that the counters crossed over each other

Sorting: Quick Sort

- How to Partition “in-place”



  - Now, SWAP the value stored in the original right counter (black arrow) with the partition element



    - Finally, RETURN the index the five is stored in (the right pointer) to indicate where the partition element ended up

Partitioning

- To partition a[left...right]:

1. Set pivot = a[left], l = left + 1, r = right;
2. while l < r, do
   2.1. while l < right & a[l] < pivot , set l = l + 1
   2.2. while r > left & a[r] >= pivot , set r = r - 1
   2.3. if l < r, swap a[l] and a[r]
3. Set a[left] = a[r], a[r] = pivot
4. Terminate

Example of partitioning

- choose pivot: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6
- search: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6
- swap: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6
- search: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6
- swap: 4 3 3 1 2 4 3 1 2 9 8 9 6 5 6
- search: 4 3 3 1 2 4 3 1 2 9 8 9 6 5 6
- swap: 4 3 3 1 2 2 3 1 4 9 8 9 6 5 6
- search: 4 3 3 1 2 2 3 1 4 9 8 9 6 5 6 (left >right)
- swap with pivot: 1 3 3 1 2 2 3 4 4 9 8 9 6 5 6

The partition method

```
int partition(int[] a, int left, int right)
{
        int p = a[left], l = left + 1, r = right;
        while (l < r)
             {
           while (l < right && a[l] < p) l++;
           while (r > left && a[r] >= p) r--;
           if (l < r)
                        {
              int temp = a[l]; a[l] = a[r]; a[r] = temp;
            }
             }
        a[left] = a[r];
        a[r] = p;
        return r;
}
```

The quicksort method

```
void quicksort(int[] array, int left, int right)
{
    if (left < right)
{
        int p = partition(array, left, right);
        quicksort(array, left, p - 1);
        quicksort(array, p + 1, right);
    }
}
```

Sorting: Quick Sort

- Quick Sort Analysis
  - This is more difficult to do than Merge Sort
    - Why?
    - With Merge Sort, we knew that our recursive calls always had equal sized inputs
      - Remember: we would split the array of size n into two arrays of size n/2 (so the smaller arrays were always the same size)
  - How is Quick Sort different? (more difficult?)
    - Each recursive call of Quick Sort could have a different sized set of numbers to sort
      - Because the size of the sets is based on our partition element
      - If partition element is in the middle, each set has about half
      - Otherwise, one set is large and one is small

Sorting: Quick Sort

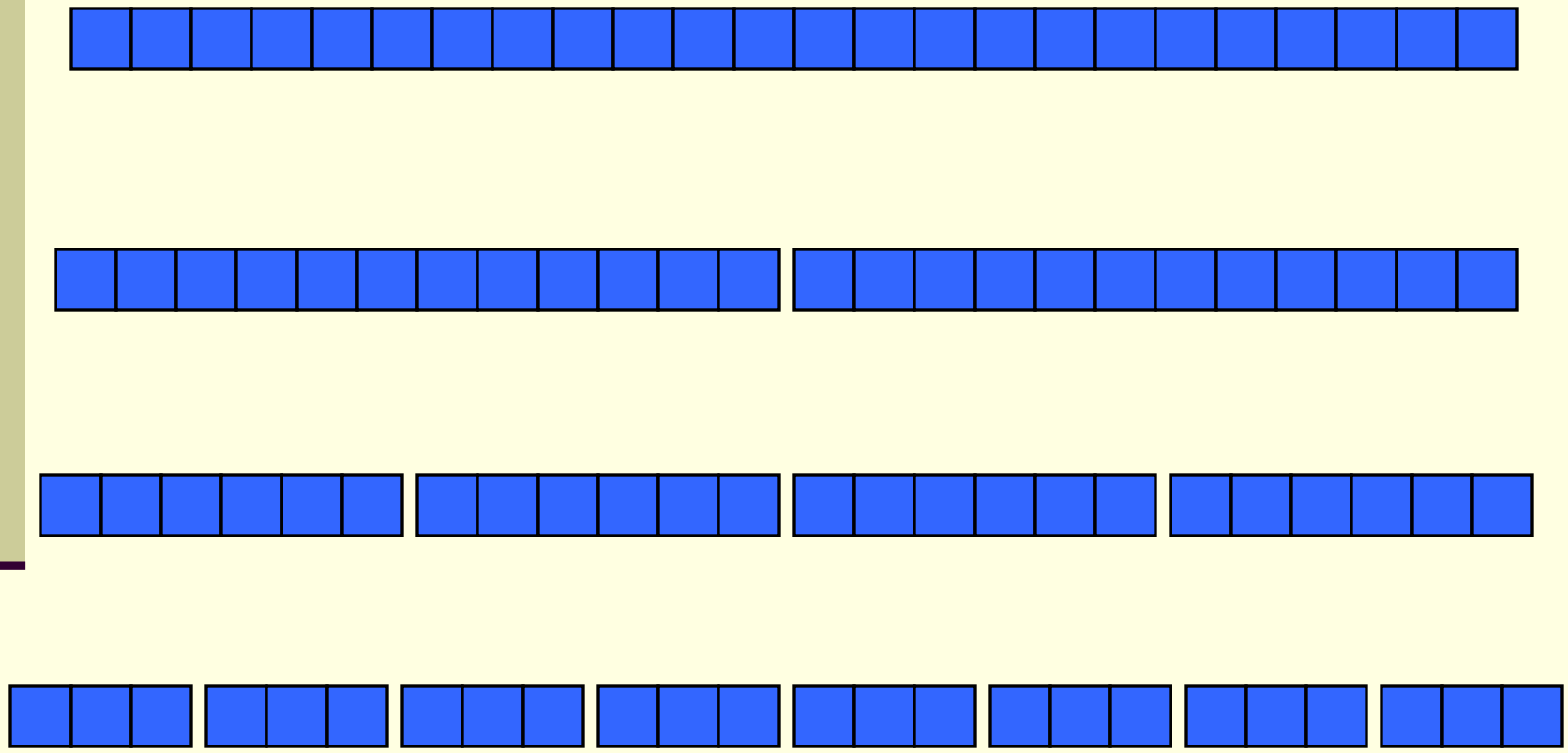
- Quick Sort Analysis
  - Location of partition element determines difficulty

1) If we get lucky
   - and the partition element is ALWAYS in the middle:
   - Then this is the BEST case
     - As we will always be **halving** the amount of work left
2) If we are unlucky:
   - and we ALWAYS choose the first or the last element in the list as our partition
   - Then this is the WORST case
     - As we will have not really sorted anything
     - We simply reduced the 2-be-sorted amount by 1

Analysis of quicksort—best case

- Suppose each partition operation divides the array almost exactly in half
- Then the depth of the recursion in $\log_2 n$
  - Because that's how many times we can halve n
- However, there are many recursions!
  - How can we figure this out?
  - We note that
    - Each partition is linear over its subarray
    - All the partitions at one level cover the array

Partitioning at various levels



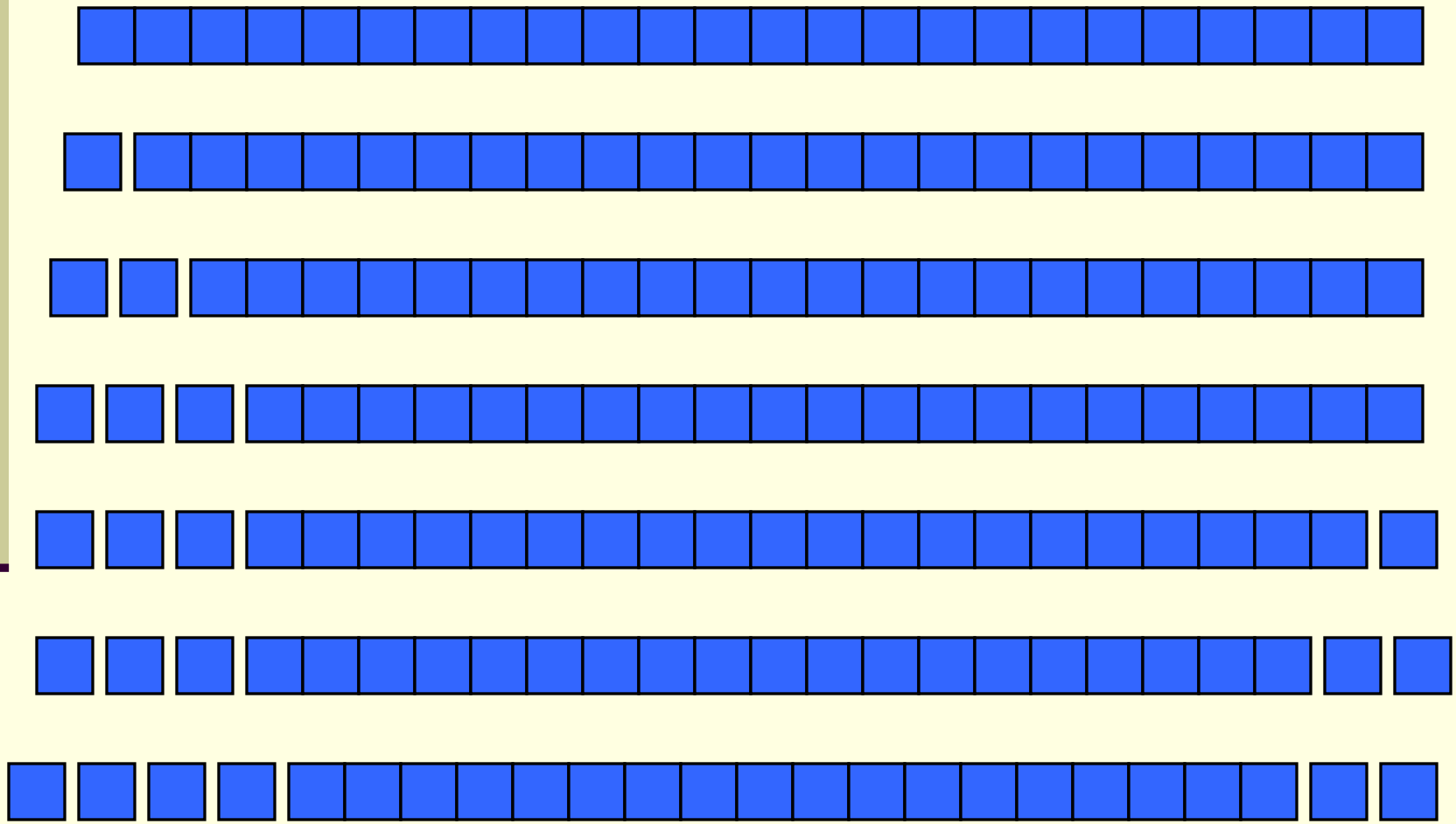
Best case II

- We cut the array size in half each time
- So the depth of the recursion in $\log_2 n$
- At each level of the recursion, all the partitions at that level do work that is linear in n
- $O(\log_2 n) * O(n) = O(n \log_2 n)$
- Hence in the average case, quicksort has time complexity $O(n \log_2 n)$
- What about the worst case?

Worst case

- In the worst case, partitioning always divides the size `n` array into these three parts:
  - A length one part, containing the pivot itself
  - A length zero part, and
  - A length `n-1` part, containing everything else
- We don't recur on the zero-length part
- Recurring on the length `n-1` part requires (in the worst case) recurring to depth `n-1`

Worst case partitioning



Worst case for quicksort

- In the worst case, recursion may be n levels deep (for an array of size n)
- But the partitioning work done at each level is still n
- $O(n) * O(n) = O(n^2)$
- So worst case for Quicksort is $O(n^2)$
- When does this happen?
  - There are many arrangements that *could* make this happen
  - Here are two common cases:
    - When the array is already sorted
    - When the array is *inversely* sorted (sorted in the opposite order)

Typical case for quicksort

- If the array is sorted to begin with, Quicksort is terrible: $O(n^2)$
- It is possible to construct other bad cases
- However, Quicksort is *usually* $O(n \log_2 n)$
- The constants are so good that Quicksort is generally the fastest algorithm known
- Most real-world sorting is done by Quicksort

Picking a better pivot

- Before, we picked the *first* element of the subarray to use as a pivot
  - If the array is already sorted, this results in $O(n^2)$ behavior
  - It's no better if we pick the *last* element
- We could do an *optimal* quicksort (guaranteed $O(n \log n)$) if we always picked a pivot value that exactly cuts the array in half
  - Such a value is called a median: half of the values in the array are larger, half are smaller
  - The easiest way to find the median is to *sort* the array and pick the value in the middle (!)

Median of three

- Obviously, it doesn't make sense to sort the array in order to find the median to use as a pivot
- Instead, compare just *three* elements of our (sub)array—the first, the last, and the middle
  - Take the *median* (middle value) of these three as pivot
  - It's possible (but not easy) to construct cases which will make this technique $O(n^2)$
- Suppose we rearrange (sort) these three numbers so that the smallest is in the first position, the largest in the last position, and the other in the middle
  - This lets us simplify and speed up the partition loop

Sorting: Quick Sort

- Quick Sort Analysis
  - Location of partition element determines difficulty
  3) If we are neither lucky or unlucky:
     - Most likely, we will have some great partitions
     - Some bad partitions
     - And some okay partitions
  - So we need to analyze each case:
     - Best case
     - Average case
     - Worst case

Sorting: Quick Sort

- Quick Sort Analysis
  - Analysis of Best Case:
    - As mentioned, in the best case, we get a perfect partition every single time
    - Meaning, if we have n elements before the partition,
      - we “luckily” pick the middle element as the partition element
      - Then we end up with n/2 - 1 elements on each side of the partition
    - So if we had 101 unsorted elements
      - we “luckily” pick the 51st element as the partition element
      - Then we end up with 50 elements smaller than this element, on the left
      - And 50 elements, greater than this element, on the right

Sorting: Quick Sort

- Quick Sort Analysis
  - Analysis of Best Case:
    - Again, here are the steps of Quick Sort:
      1) Partition the elements
      2) Quick Sort the smaller half (recursive)
      3) Quick Sort the larger half (recursive)
    - So at each recursive step, the input size is **halved**
    - Let T(n) be the running time of Quick Sort on n elements
      - And remember that Partition runs on O(n) time
    - So we get our recurrence relation for the best case:
      - T(n) = 2*T(n/2) + O(n)
        - This is the same recurrence relation as Merge Sort
      - So in the best case, Quick Sort runs in O(nlogn) time

Sorting: Quick Sort

- Quick Sort Analysis
  - Analysis of Worst Case:
    - Assume that we are horribly unlucky
    - And when choosing the partition element, we somehow end up always choosing the greatest value remaining
    - **Now for this worst case:**
      - How many times will the Partition function run?
        - Think: when we choose the greatest element (for example)
        - We have the partition element, then ALL other elements are to the left in one partition
        - The “partition” to the right will have ZERO elements
      - So Partition will run n-1 times
        - The first time results in comparing n-1 values, then comparing n-2 values the second time, followed by n-3, etc.

Sorting: Quick Sort

- Quick Sort Analysis
  - Analysis of Worst Case:
    - How many times will the Partition function run?
      - Partition will run n-1 times
        - The first time results in comparing n-1 values, then comparing n-2 values the second time, followed by n-3, etc.
    - When we sum the number of compares, we get:
      - 1 + 2 + 3 + … + (n - 1)
      - You should know what this equals:

$$\frac{(n-1)n}{2}$$

    - Thus, the worst case running time is $O(n^2)$

Sorting: Quick Sort

- Quick Sort Analysis
  - Summary:
    - Best Case: O(nlogn)
    - **Average Case: O(nlogn)**
    - Worst Case: $O(n^2)$
  - Compare Merge Sort and Quick Sort:
    - Merge Sort: guaranteed O(nlogn)
    - Quick Sort: best and average case is O(nlogn) but worst case is $O(n^2)$

Final comments

- Quicksort is the fastest known sorting algorithm
- For optimum efficiency, the pivot must be chosen carefully
- “Median of three” is a good technique for choosing the pivot
- However, no matter what you do, there will be some cases where Quicksort runs in $O(n^2)$ time