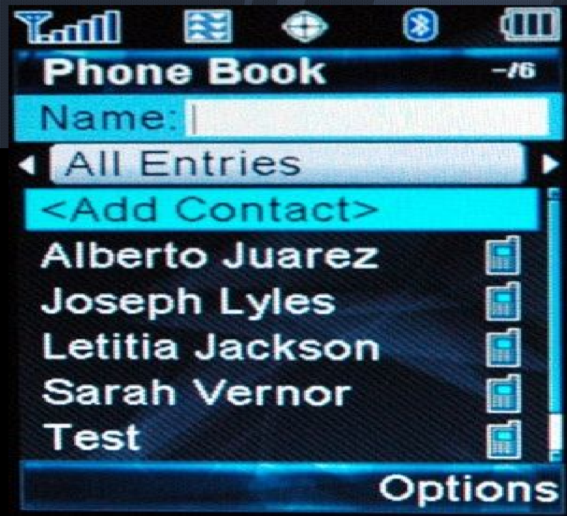
SORTING ALGORITHMS

-PINAK PATEL

Sorting Algorithms

Song Name	Time	Track #	Artist	Album
Letters from the Wasteland	4:29	1 of 10	The Wallflowers	Breach
When You're On Top	3:54	1 of 13	The Wallflowers	Red Letter Days
Hand Me Down	3:35	2 of 10	The Wallflowers	Breach
How Good It Can Get	4:11	2 of 13	The Wallflowers	Red Letter Days
Sleepwalker	3:31	3 of 10	The Wallflowers	Breach
Closer To You	3:17	3 of 13	The Wallflowers	Red Letter Days
I've Been Delivered	5:01	4 of 10	The Wallflowers	Breach
Everybody Out Of The Water	3:42	4 of 13	The Wallflowers	Red Letter Days
Witness	3:34	5 of 10	The Wallflowers	Breach
Three Ways	4:19	5 of 13	The Wallflowers	Red Letter Days
Some Flowers Bloom Dead	4:43	6 of 10	The Wallflowers	Breach
Too Late to Quit	3:54	6 of 13	The Wallflowers	Red Letter Days
Mourning Train	4:04	7 of 10	The Wallflowers	Breach
If You Never Got Sick	3:44	7 of 13	The Wallflowers	Red Letter Days
Up from Under	3:38	8 of 10	The Wallflowers	Breach
Health and Happiness	4:03	8 of 13	The Wallflowers	Red Letter Days
Murder 101	2:31	9 of 10	The Wallflowers	Breach
See You When I Get There	3:09	9 of 13	The Wallflowers	Red Letter Days
Birdcage	7:42	10 of 10	The Wallflowers	Breach
Feels Like Summer Again	3:48	10 of 13	The Wallflowers	Red Letter Days
Everything I Need	3:37	11 of 13	The Wallflowers	Red Letter Days
Here in Pleasantville	3:40	12 of 13	The Wallflowers	Red Letter Days
Empire in My Mind (Bonus Track)	3:31	13 of 13	The Wallflowers	Red Letter Days



Sorting Algorithms

Bubble Sort

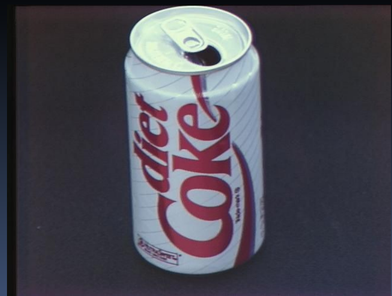
Bubble Sort .. .. ..

- Bubble sort examines the array from start to finish, comparing elements as it goes.
- Any time it finds a larger element before a smaller element, it swaps the two.
- In this way, the larger elements are passed towards the end.
- The largest element of the array therefore "bubbles" to the end of the array.
- Then it repeats the process for the unsorted portion of the array until the whole array is sorted.

Bubble Sort .. .. ..

- Bubble sort works on the same general principle as shaking a soft drink bottle.
- Right after shaking, the contents are a mixture of bubbles and soft drink, distributed randomly.
- Because bubbles are lighter than the soft drink, they rise to the surface, displacing the soft drink downwards.
- This is how bubble sort got its name, because the smaller elements "float" to the top, while

  the larger elements "sink" to the bottom.



Bubble Sort: Idea

- Idea: bubble in water.
  - Bubble in water moves upward. Why?
- How?
  - When a bubble moves upward, the water from above will move downward to fill in the space left by the bubble.

Bubble Sort Example

9, 6, 2, 12, 11, 9, 3, 7

Bubblesort compares the numbers in pairs from left to right exchanging when necessary. Here the first number is compared to the second and as it is larger they are exchanged.

The end of the list has been reached so this is the end of the first pass. The twelve at the end of the list must be largest number in the list and so is now in the correct position. We now start a new pass from left to right.

The 12 is greater than the 3 so they are exchanged.

The 12 is greater than the 7 so they are exchanged.

Bubble Sort Example

First Pass	6, 2, 9, 11, 9, 3, 7, 12
Second Pass	2, 6, 9, 9, 3, 7, 11, 12

Notice that this time we do not have to compare the last two numbers as we know the 12 is in position. This pass therefore only requires 6 comparisons.

Bubble Sort Example

First Pass	6, 2, 9, 11, 9, 3, 7, 12
Second Pass	2, 6, 9, 9, 3, 7, 11, 12
Third Pass	2, 6, 9, 3, 7, 9, 11, 12

This time the 11 and 12 are in position. This pass therefore only requires 5 comparisons.

Bubble Sort Example

First Pass	6,	2,	9,	11,	9,	3,	7,	12
Second Pass	2,	6,	9,	9,	3,	7,	11,	12
Third Pass	2,	6,	9,	3,	7,	9,	11,	12
Fourth Pass	2,	6,	3,	7,	9,	9,	11,	12

Each pass requires fewer comparisons. This time only 4 are needed.

Bubble Sort Example

First Pass	6, 2, 9, 11, 9, 3, 7, 12
Second Pass	2, 6, 9, 9, 3, 7, 11, 12
Third Pass	2, 6, 9, 3, 7, 9, 11, 12
Fourth Pass	2, 6, 3, 7, 9, 9, 11, 12
Fifth Pass	2, 3, 6, 7, 9, 9, 11, 12

The list is now sorted but the algorithm does not know this until it completes a pass with no exchanges.

Bubble Sort Example

First Pass	6, 2, 9, 11, 9, 3, 7, 12
Second Pass	2, 6, 9, 9, 3, 7, 11, 12
Third Pass	
Fourth Pass	
Fifth Pass	2, 3, 6, 7, 9, 9, 11, 12
Sixth Pass	2, 3, 6, 7, 9, 9, 11, 12

This pass no exchanges are made so the algorithm knows the list is sorted. It can therefore save time by not doing the final pass. With other lists this check could save much more work.

Bubble Sort Example - Quiz Time

1. Which number is definitely in its correct position at the end of the first pass?

Answer: The last number must be the largest.

2. How does the number of comparisons required change as the pass number increases?

Answer: Each pass requires one fewer comparison than the last.

3. How does the algorithm know when the list is sorted?

Answer: When a pass with no exchanges occurs.

4. What is the maximum number of comparisons required for a list of 10 numbers?

Answer: 9 comparisons, then 8, 7, 6, 5, 4, 3, 2, 1 so total 45

Bubble Sort: Example

1	40	2	1	43	3	65	0	-1	58	3	42	4
2	2	1	40	3	43	0	-1	58	3	42	4	65
3	1	2	3	40	0	-1	43	3	42	4	58	65
4	1	2	3	0	-1	40	3	42	4	43	58	65

- Notice that at least one element will be in the correct position each iteration.

Bubble Sort: Example

5	1	2	0	-1	3	3	40	4	42	43	58	65
6	1	0	-1	2	3	3	4	40	42	43	58	65
7	0	-1	1	2	3	3	4	40	42	43	58	65
8	-1	0	1	2	3	3	4	40	42	43	58	65

Bubble Sort: Algorithm

- BUBBLE(Data, N)
  - Here Data is an array with N elements.
  - This algorithm sorts the elements in Data.

1. Repeat steps 2 & 3 for k=1 to N-1.
2. Set ptr=1. [Initializes pass pointer ptr]
3. Repeat while ptr<=N-k: [Executes pass]
   1. If Data[ptr] > Data[ptr+1], then:
      1. Interchange Data[ptr] and Data[ptr + 1].

      [End of If structure.]
   2. Set ptr = ptr + 1.

      [End of inner loop.]

   [End of Step 1 outer loop.]
4. Exit.

Bubble Sort – Analysis

- ***Best-case:*** **→ O(n)**
  - Array is already sorted in ascending order.
  - The number of moves: 0 → O(1)
  - The number of key comparisons: (n-1) → O(n)
- ***Worst-case:*** **→ $O(n^2)$**
  - Array is in reverse order:
  - Outer loop is executed n-1 times,
  - The number of moves: 3*(1+2+...+n-1) = 3 * n*(n-1)/2 → $O(n^2)$
  - The number of key comparisons: (1+2+...+n-1)= n*(n-1)/2 → $O(n^2)$
- Average-case: **→ $O(n^2)$**
  - We have to look at all possible initial data organizations.
- **So, Bubble Sort is $O(n^2)$**

Example:

```
#include <stdio.h>
void bubble_sort(long [], long);
int main()
{
    long array[100], n, c, d, swap;
    printf("Enter number of elements:");
    scanf("%ld", &n);
    printf("Enter %ld longegers\n", n);
    for (c = 0; c < n; c++)
    scanf("%ld", &array[c]);
    bubble_sort(array, n);
    printf("Sorted list in ascending order:n");
    for ( c = 0 ; c < n ; c++ )
    printf("%ld\n", array[c]);
    return 0;
}
```

Cont…

```
void bubble_sort(long list[], long n)
 {
     long c, d, t;
     for (c = 0 ; c < ( n - 1 ); c++)
      {
                 for (d = 0 ; d < n - c - 1; d++)
         {
           if (list[d] > list[d+1])
            {
               t = list[d];
               list[d] = list[d+1];
               list[d+1]= t;
           }
        }
     }
 }
```

Sorting Algorithms

Selection Sort

Selection Sort……

- Idea:
  - Find the largest element in the array
  - Exchange it with the element in the rightmost position
  - Find the second largest element and exchange it with the element in the second rightmost position
  - Continue until the array is sorted

Before sorting	14	2	10	5	1	3	17	7
After pass 1	14	2	10	5	1	3	7	17
After pass 2	7	2	10	5	1	3	14	17
After pass 3	7	2	3	5	1	10	14	17
After pass 4	1	2	3	5	7	10	14	17

Selection Sort

5	1	3	4	6	2

This is starting state of an array

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

Start traversing or searching for the largest or smallest element in this array for sorting purpose. In this example assume for larger one.

- Comparison
- Data Movement
- Sorted

Selection Sort

5	1	3	4	6	2

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

↑
Largest

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	2	6

Swap the larger element with the last element in the unsorted list.

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	2	6

Largest element is at its sorted position

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	2	6

Again Start traversing or searching for the largest or smallest element in unsorted portion of array for sorting purpose. In this example assume for larger one.

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	2	6

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	2	6

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	2	6

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	2	6

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	2	6

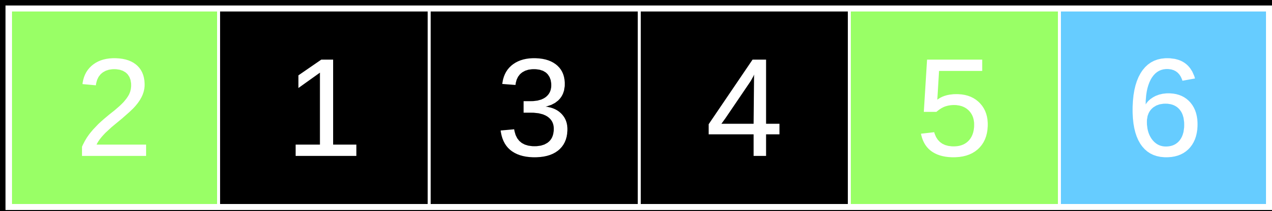
↑
Largest

Comparison

Data Movement

Sorted

Selection Sort



Swap the larger element with the last element in the unsorted list.

- Comparison
- Data Movement
- Sorted

Selection Sort

2	1	3	4	5	6

Largest element is at its sorted position

- Comparison
- Data Movement
- Sorted

Selection Sort

2	1	3	4	5	6

Again Start traversing or searching for the largest or smallest element in unsorted portion of array for sorting purpose. In this example assume for larger one.

- Comparison
- Data Movement
- Sorted

Selection Sort

2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

↑
Largest

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

Largest element is already at sorted position therefore no data swapping is required.

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

Largest element is at its sorted position

- Comparison
- Data Movement
- Sorted

Selection Sort

2	1	3	4	5	6

Again Start traversing or searching for the largest or smallest element in unsorted portion of array for sorting purpose. In this example assume for larger one.

- Comparison
- Data Movement
- Sorted

Selection Sort

2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

↑
Largest

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

Largest element is already at sorted position therefore no data swapping is required.

- Comparison
- Data Movement
- Sorted

Selection Sort

2	1	3	4	5	6

Largest element is at its sorted position

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

↑
Largest

- Comparison
- Data Movement
- Sorted

Selection Sort

1	2	3	4	5	6

Swap the larger element with the last element in the unsorted list.

- Comparison
- Data Movement
- Sorted

Selection Sort

1	2	3	4	5	6

DONE!

- Comparison
- Data Movement
- Sorted

Selection Sort: Algorithm

- SELECTION(A, N)
  - This algorithm sorts the array A with N elements.

1. Repeat steps 2 and 3 for k = 1, 2, 3, ….., N-1:
   1. Call MIN(A, k, N, loc).
   2. [Interchange A[k] and A[loc].]
      1. Set temp = A[k],
      2. A[k] = A[loc],
      3. A[loc] = temp.

   [End of step 1 loop.]
2. Exit.

Selection Sort: Algorithm (cont…)

- MIN(A, k, N, loc)
  - An array A is in memory. This procedure finds the location loc of the smallest element among A[k], A[k + 1],…….,A[N].

1. Set MIN = A[k] and loc = k. [Initializes pointers.]
2. Repeat for J = k + 1, k + 2,……, N:
   1. If MIN > A[J], then:
      1. Set MIN = A[J],
      2. loc = J.

   [End of loop.]
3. Return.

Example:

```
#include <stdio.h>
  main()
  {
   int A[20], N, Temp, i, j;
   printf(" ENTER THE NUMBER OF TERMS...: ");
   scanf("%d",&N);
   printf("\n ENTER THE ELEMENTS OF  ARRAY...:");
   for(i=1; i<=N; i++)
   {
     scanf("\n\t\t%d", &A[i]);
   }
```

Cont…

```
for(i=1; i<=N-1; i++)
     for(j=i+1; j<=N;j++)
    if(A[i]>A[j])
    {
     Temp = A[i];
     A[i] = A[j];
     A[j] = Temp;
    }
    printf("THE ASCENDING ORDER LIST IS…:\n");
    for(i=1; i<=N; i++)
     printf("\n %d",A[i]);
  }
```

Selection Sort: Analysis

- Number of comparisons:
- $(n-1) + (n-2) + \ldots + 3 + 2 + 1 =$
- $n * (n-1)/2 =$
- $(n^2 - n)/2$
- $\rightarrow O(n^2)$
- Number of exchanges (worst case):
- $n - 1$

$\rightarrow$ $O(n)$

Overall (worst case) $O(n) + O(n^2) = O(n^2)$ ('quadratic sort')

Sorting Algorithms

Insertion Sort

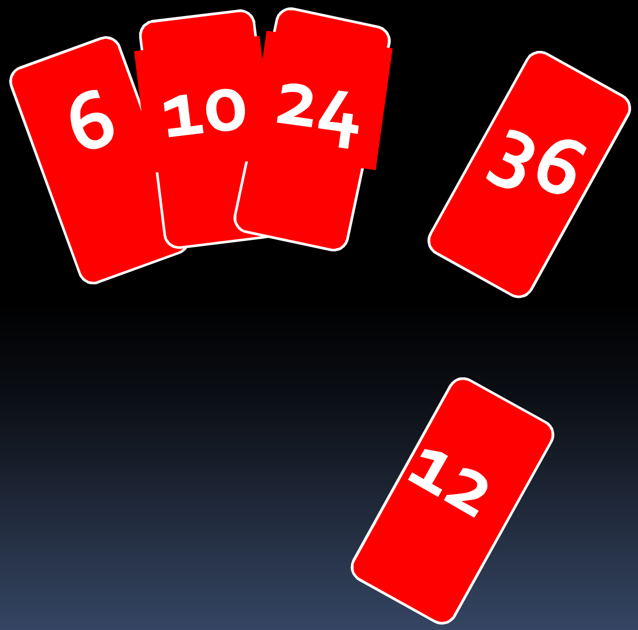
Insertion Sort

- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

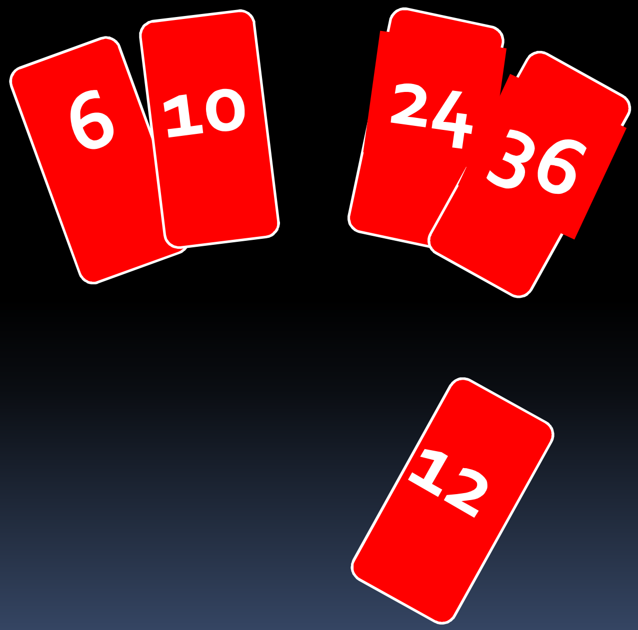
Insertion Sort

To insert 12, we need to make room for it by moving first 36 and then 24.

Insertion Sort



Insertion Sort



Insertion Sort

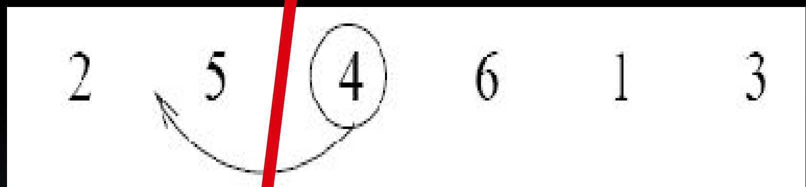
input array

5 2 4 6 1 3

at each iteration, the array is divided in two sub-arrays:

left sub-array

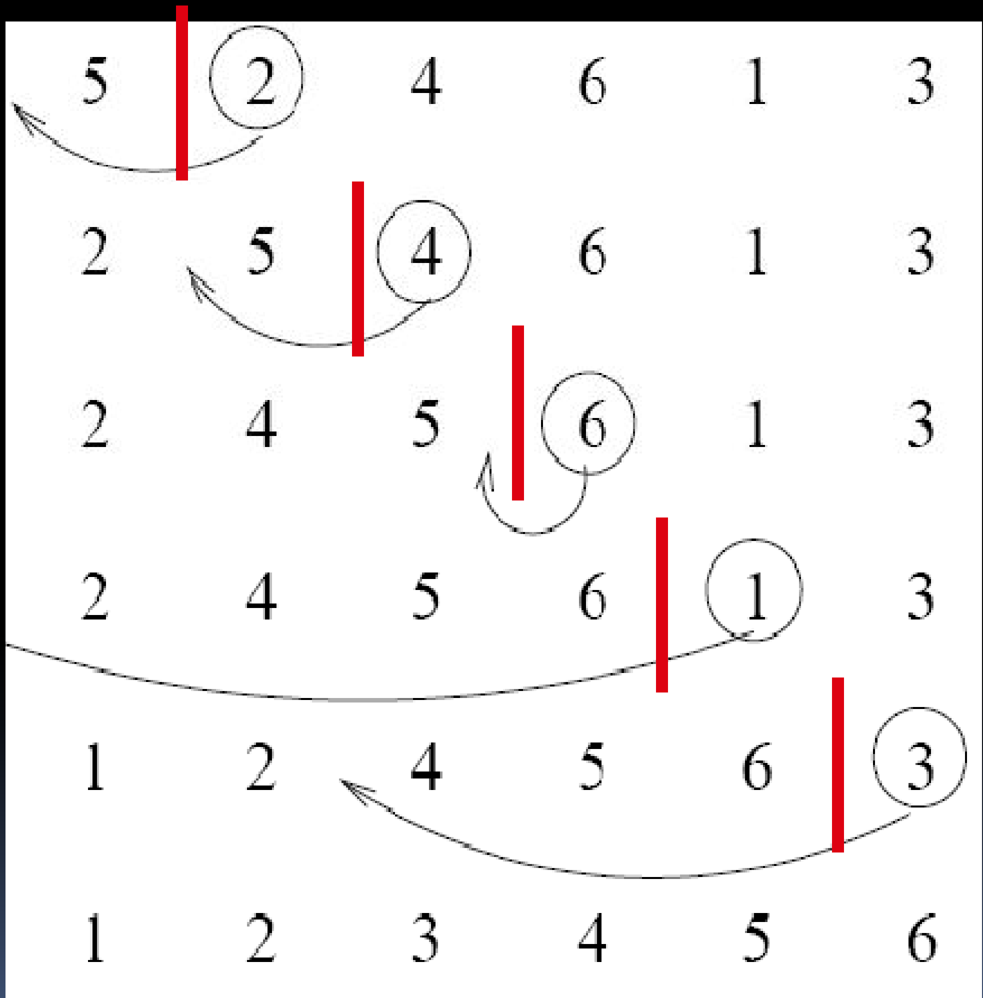
right sub-array

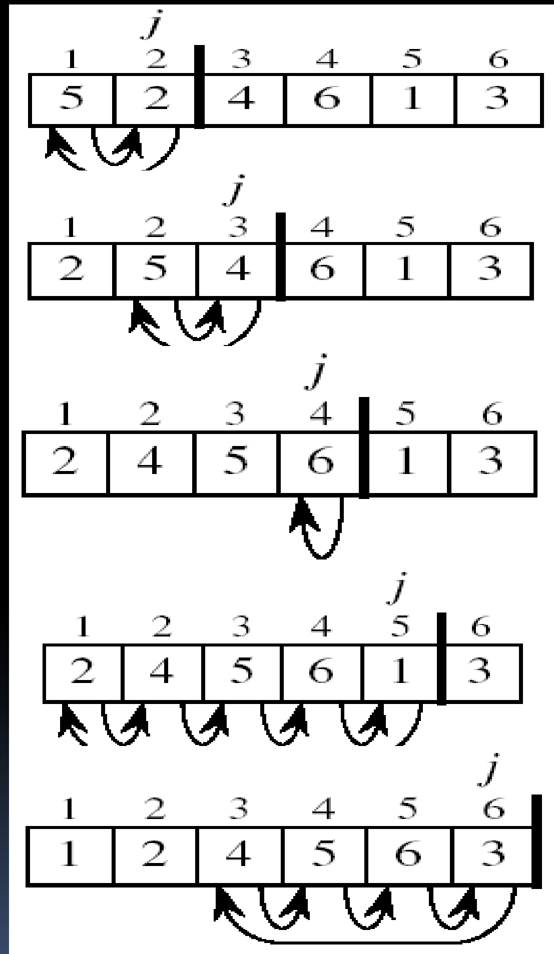


sorted

unsorted

Insertion Sort





Insertion Sort: Algorithm

- INSERTION(A, N)
  - This algorithm sorts the array A with N elements.

1. Set A[0]=$. [Initializes sentinel element]
2. Repeat steps 3 to 5 for k = 2, 3, ….., N:
   1. Set temp = A[k] and ptr = k-1.
   2. Repeat while temp < A[ptr]:
      1. Set A[ptr + 1] = A[ptr]. [Moves element forward]
      2. Set ptr = ptr + 1.

      [End of loop.]
   3. Set A[ptr + 1] = temp. [Inserts element in proper place.]

   [End of Step 2 loop.]
3. Return.

Insertion Sort: Analysis

- Number of comparisons (worst case):
- $(n-1) + (n-2) + \ldots + 3 + 2 + 1 \rightarrow O(n^2)$
- Number of comparisons (best case):
- $n - 1 \rightarrow O(n)$
- Number of exchanges (worst case):
- $(n-1) + (n-2) + \ldots + 3 + 2 + 1 \rightarrow O(n^2)$
- Number of exchanges (best case):
- $0 \rightarrow O(1)$
- Overall worst case: $O(n^2) + O(n^2) = O(n^2)$

Example:

```
#include<stdio.h>
  void main()
  {
   int A[20], N, Temp, i, j;
   printf("ENTER THE NUMBER OF TERMS...: ");
   scanf("%d", &N);
   printf("\n ENTER THE ELEMENTS OF THE ARRAY...:");
   for(i=0; i<N; i++)
   {
    scanf("\n\t\t%d",&A[i]);
   }
```

Cont…

```
for(i=1; i<N; i++)
    {
     Temp = A[i];
     j = i-1;
     while(Temp<A[j] && j>=0)
     {
       A[j+1] = A[j];
       j = j-1;
     }
     A[j+1] = Temp;
    }
    printf("\nTHE ASCENDING ORDER LIST IS...:\n");
    for(i=0; i<N; i++)
     printf("\n%d", A[i]);
    }
```

Comparison of Quadratic Sorts

	Comparisons		Exchanges	
	Best	Worst	Best	Worst
Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$	$O(n)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(1)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(1)$	$O(n^2)$

Sorting Algorithms

Merge Sort

Merging Cars by key
[Aggressiveness of driver].
Most aggressive goes first.

Mergesort

- Merge sort (divide-and-conquer)
  - Divide array into two halves.

A	L	G	O	R	I	T	H	M	S

A	L	G	O	R

I	T	H	M	S

divide

Mergesort

- Merge sort (divide-and-conquer)
  - Divide array into two halves.
  - Recursively sort each half.

A	L	G	O	R	I	T	H	M	S

A	L	G	O	R	I	T	H	M	S	divide
A	G	L	O	R	H	I	M	S	T	sort

Mergesort

- Merge sort (divide-and-conquer)
  - Divide array into two halves.
  - Recursively sort each half.
  - Merge two halves to make sorted whole.

A	L	G	O	R	I	T	H	M	S

A	L	G	O	R	I	T	H	M	S	divide
A	G	L	O	R	H	I	M	S	T	sort

A	G	H	I	L	M	O	R	S	T	merge

Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.

smallest

A	G	L	O	R

smallest

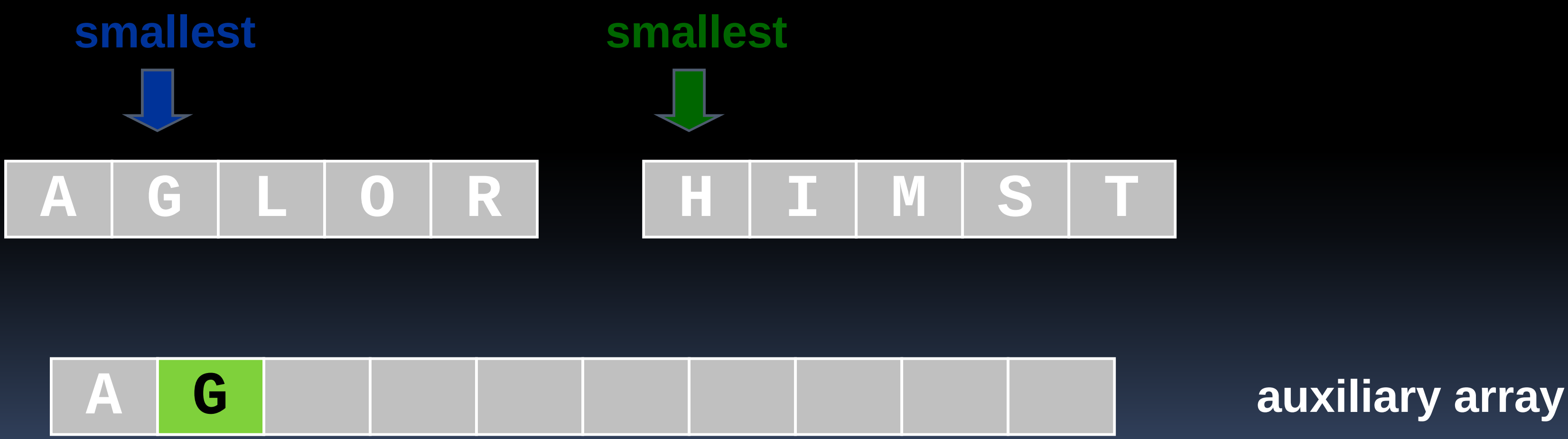
H	I	M	S	T

A									

auxiliary array

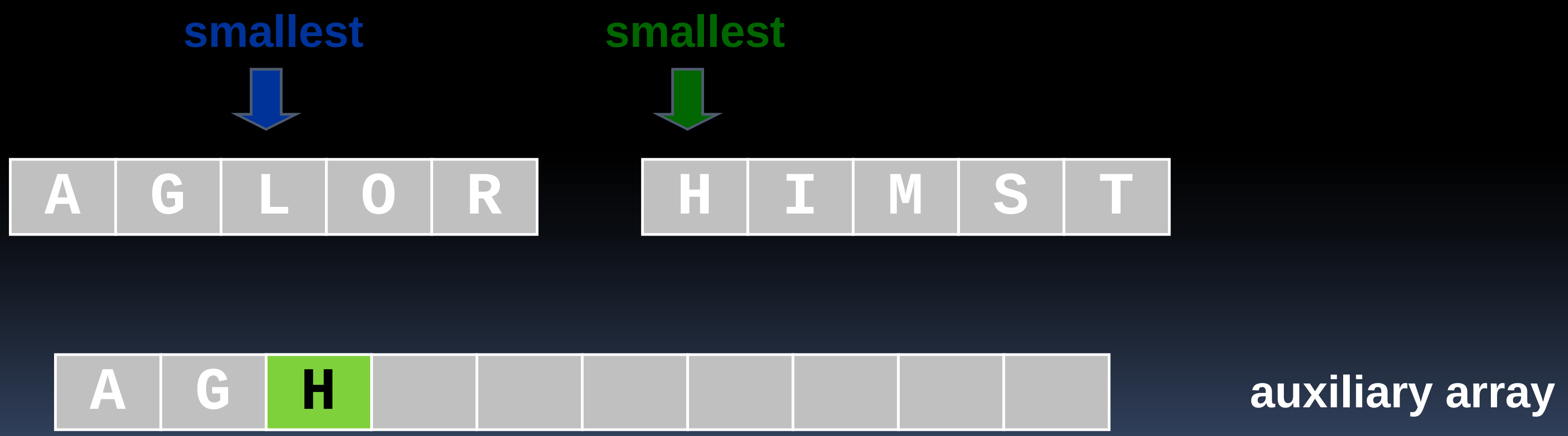
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



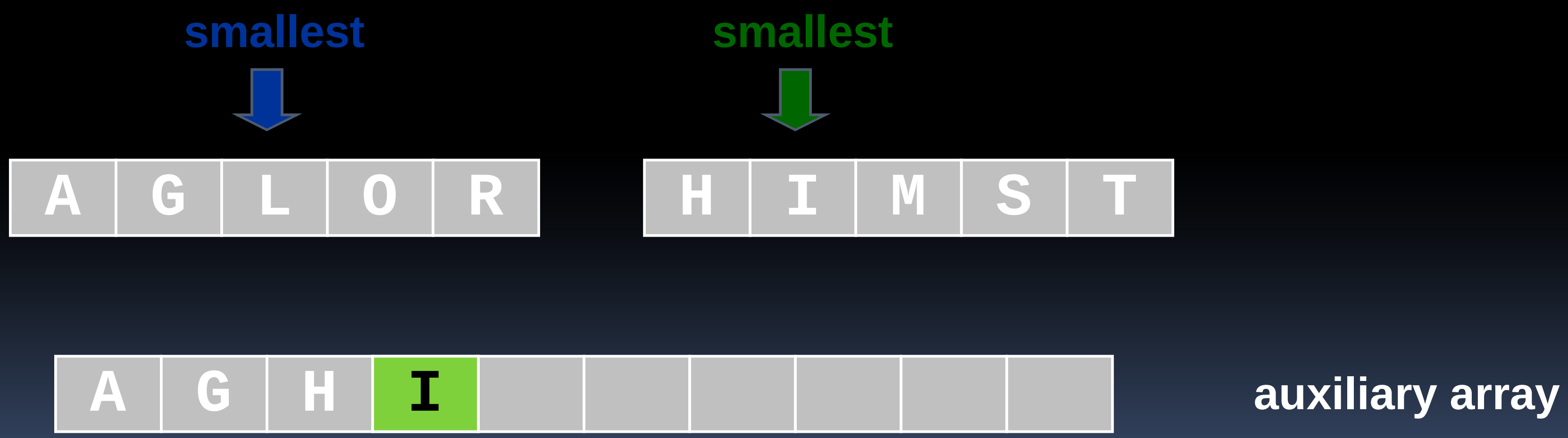
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



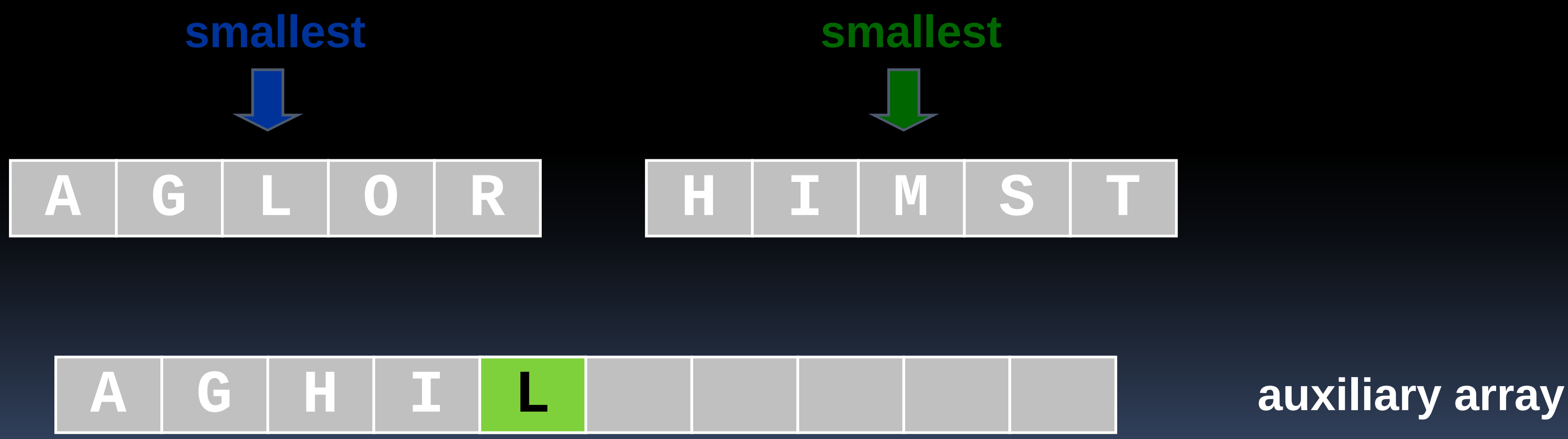
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.

smallest

smallest

A	G	L	O	R

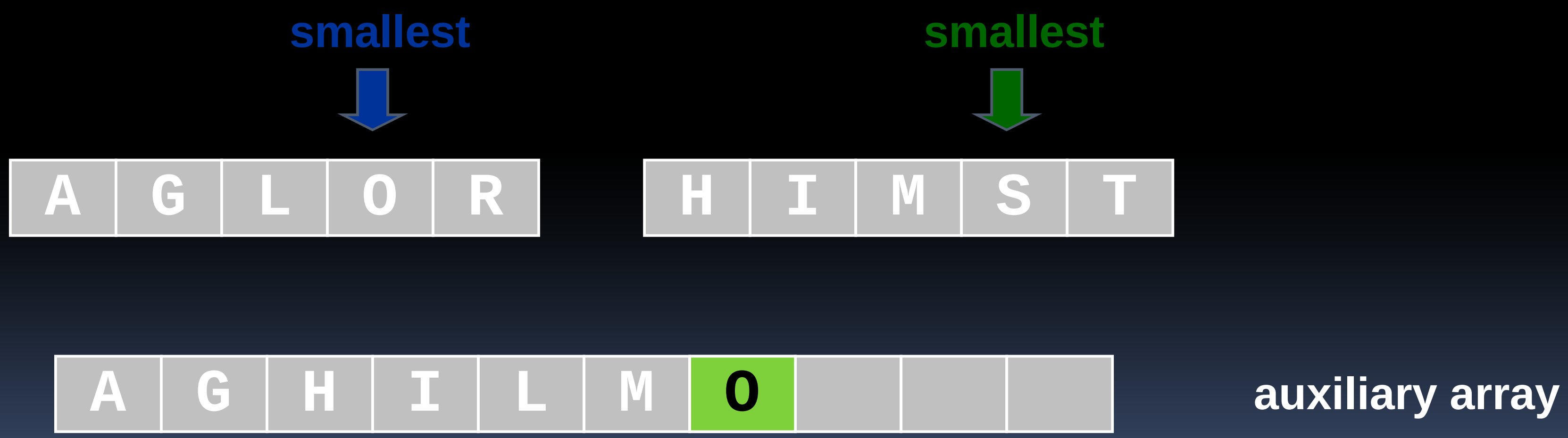
H	I	M	S	T

A	G	H	I	L	M				

auxiliary array

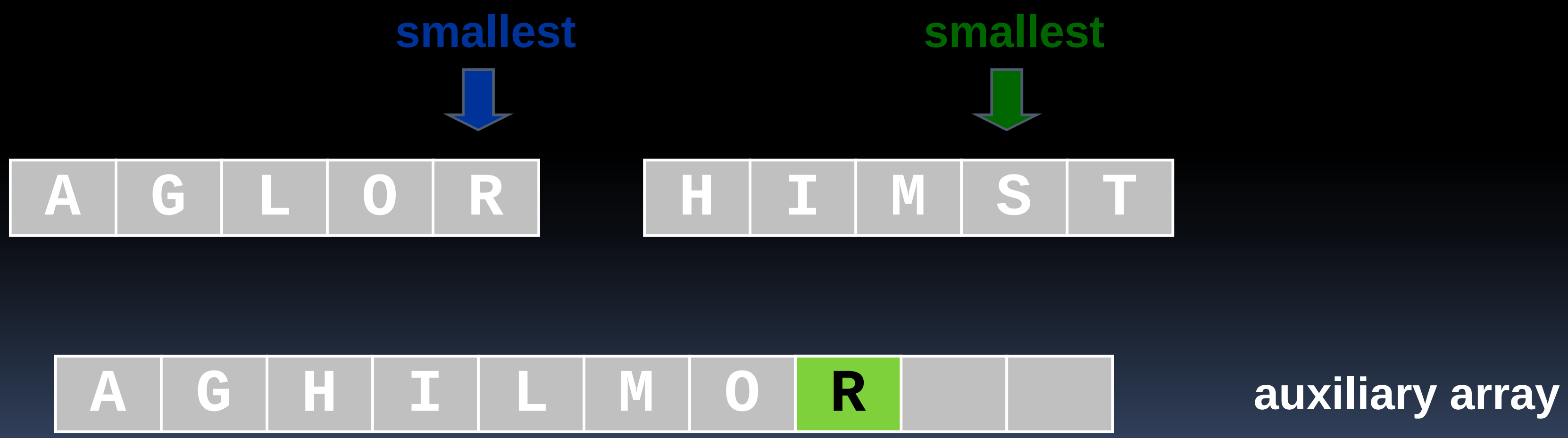
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



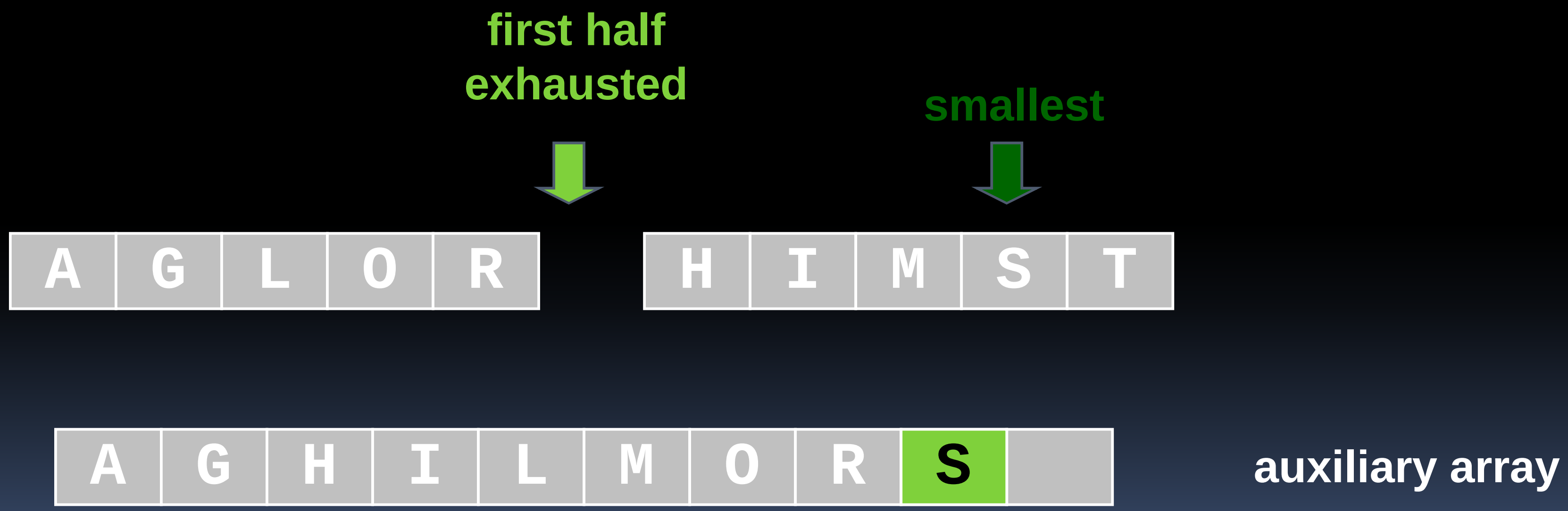
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



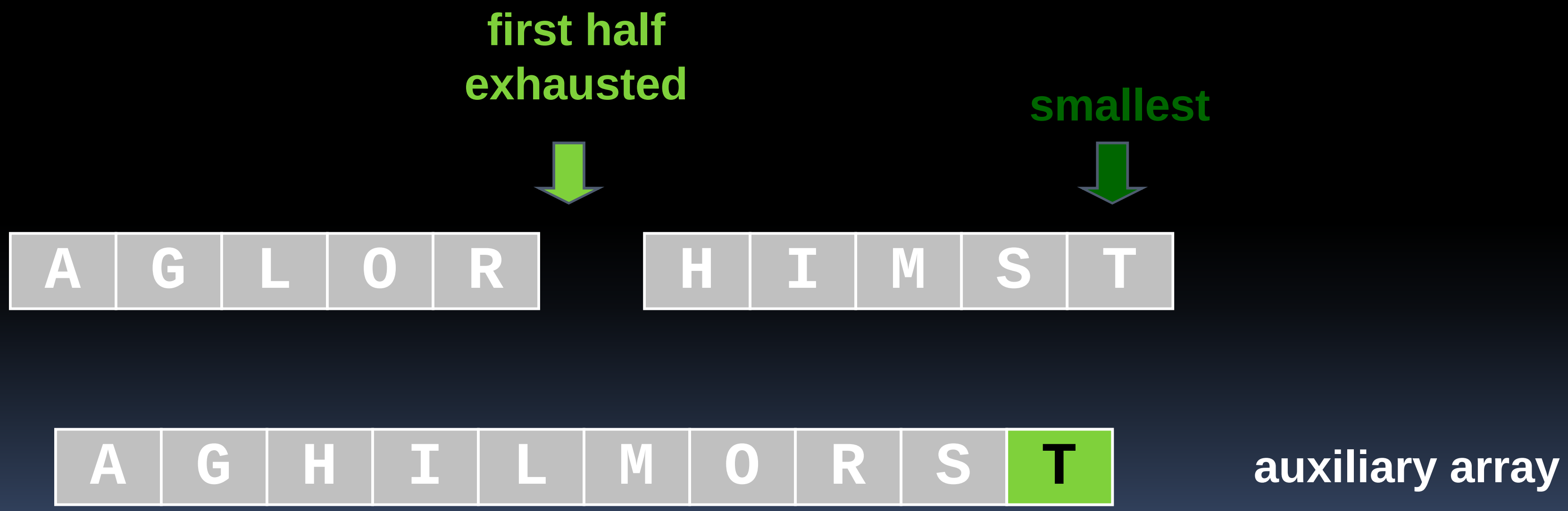
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



Mergesort : example

98	23	45	14	6	67	33	42

98	23	45	14	6	67	33	42

98	23	45	14

6	67	33	42

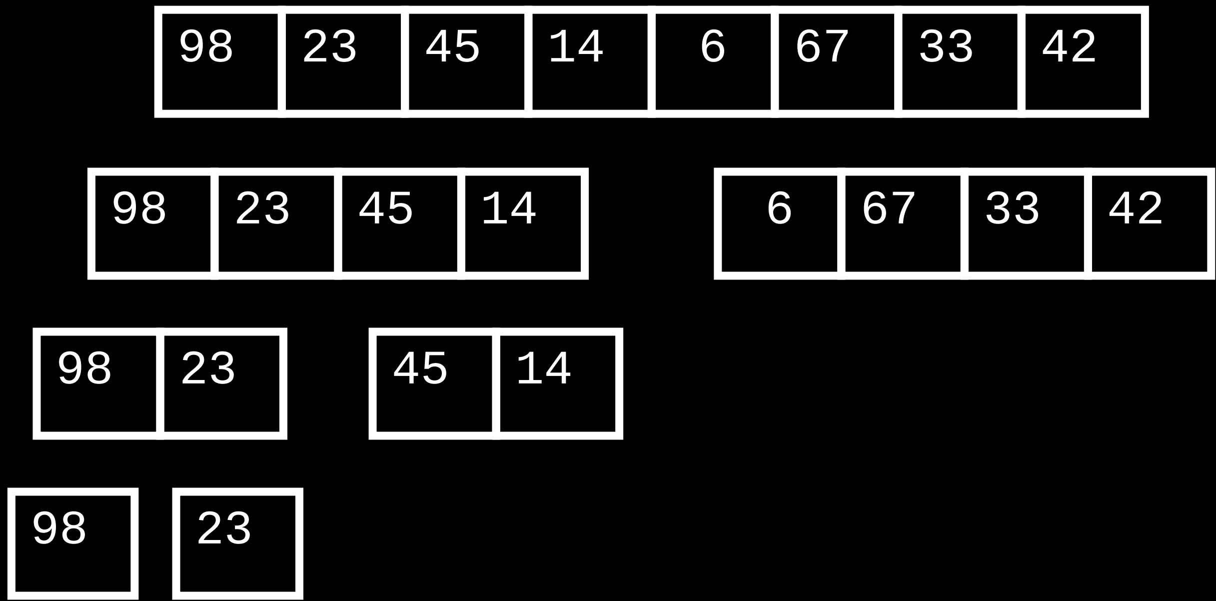
98	23	45	14	6	67	33	42

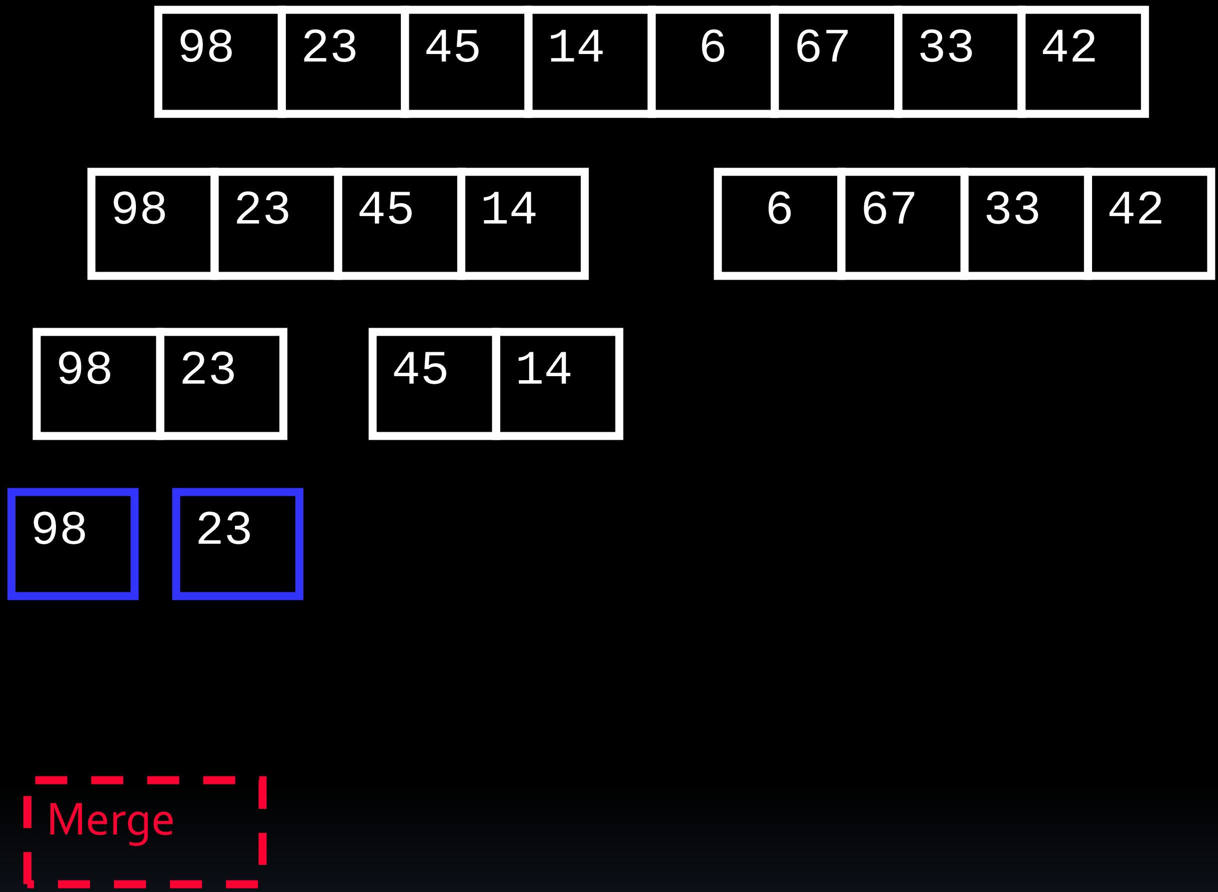
98	23	45	14

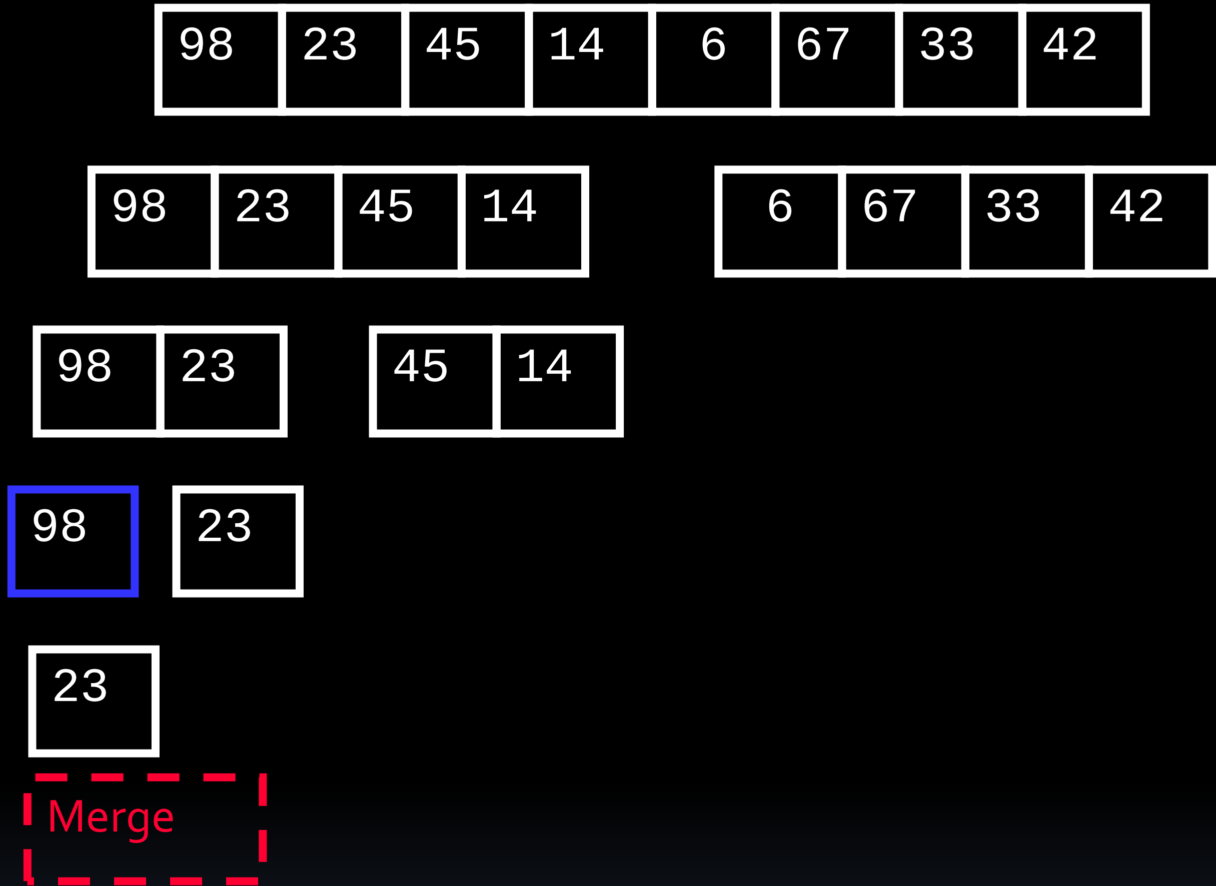
6	67	33	42

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

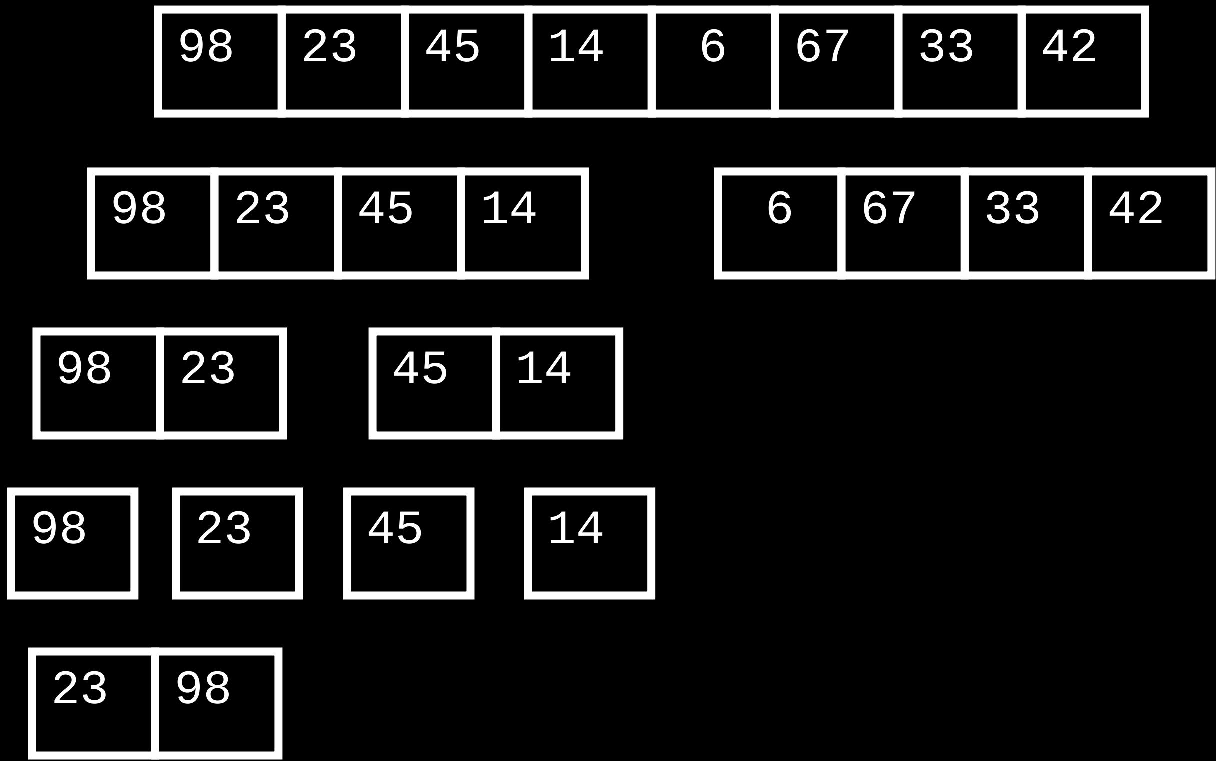
45	14

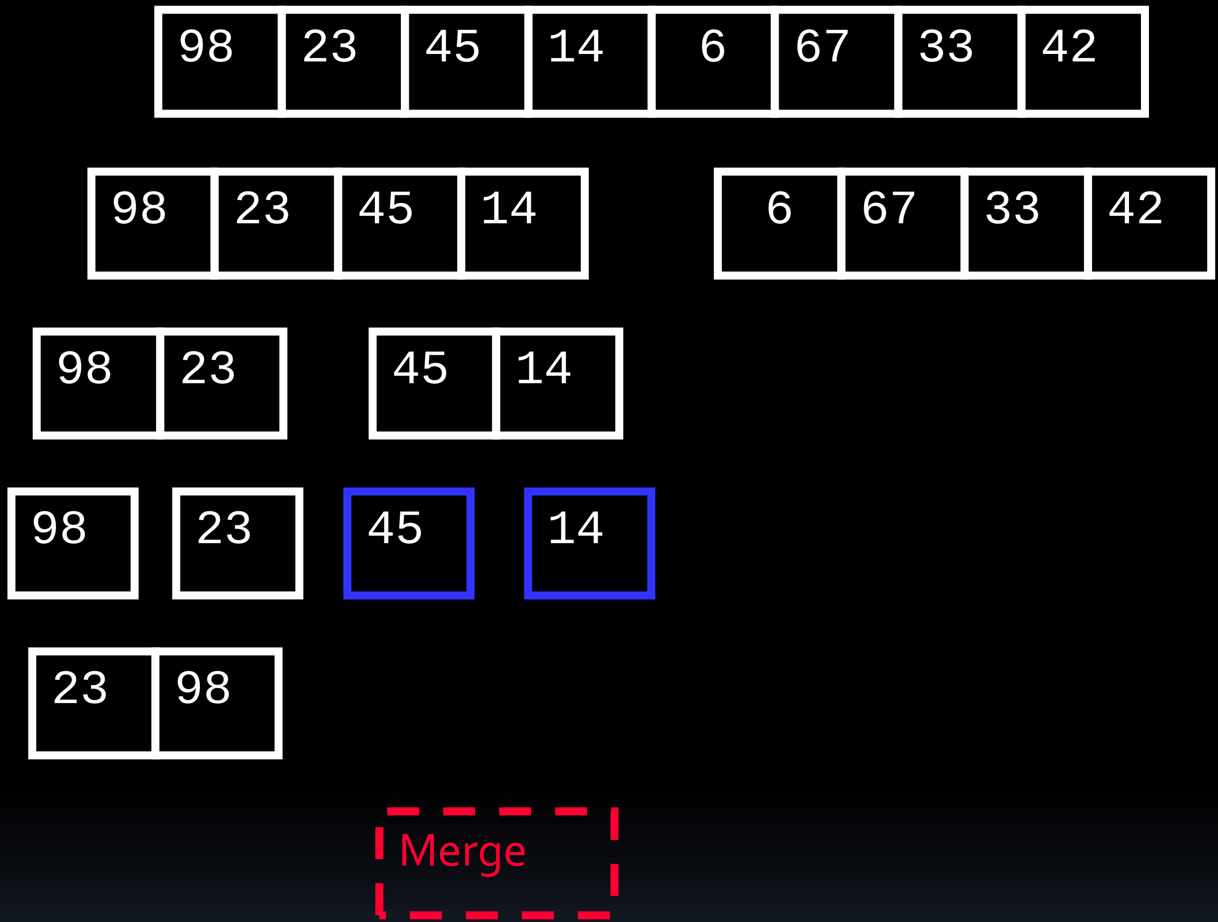
98

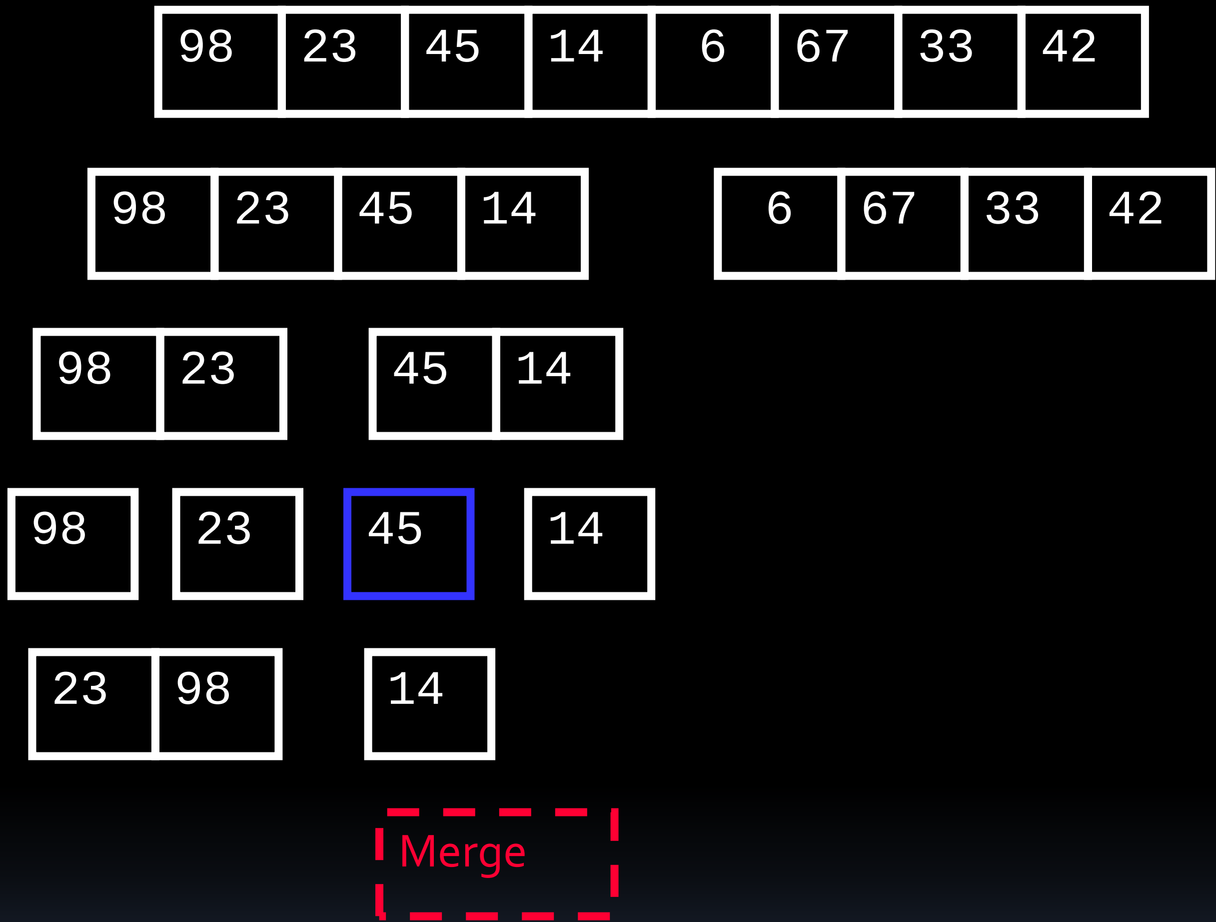
23

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

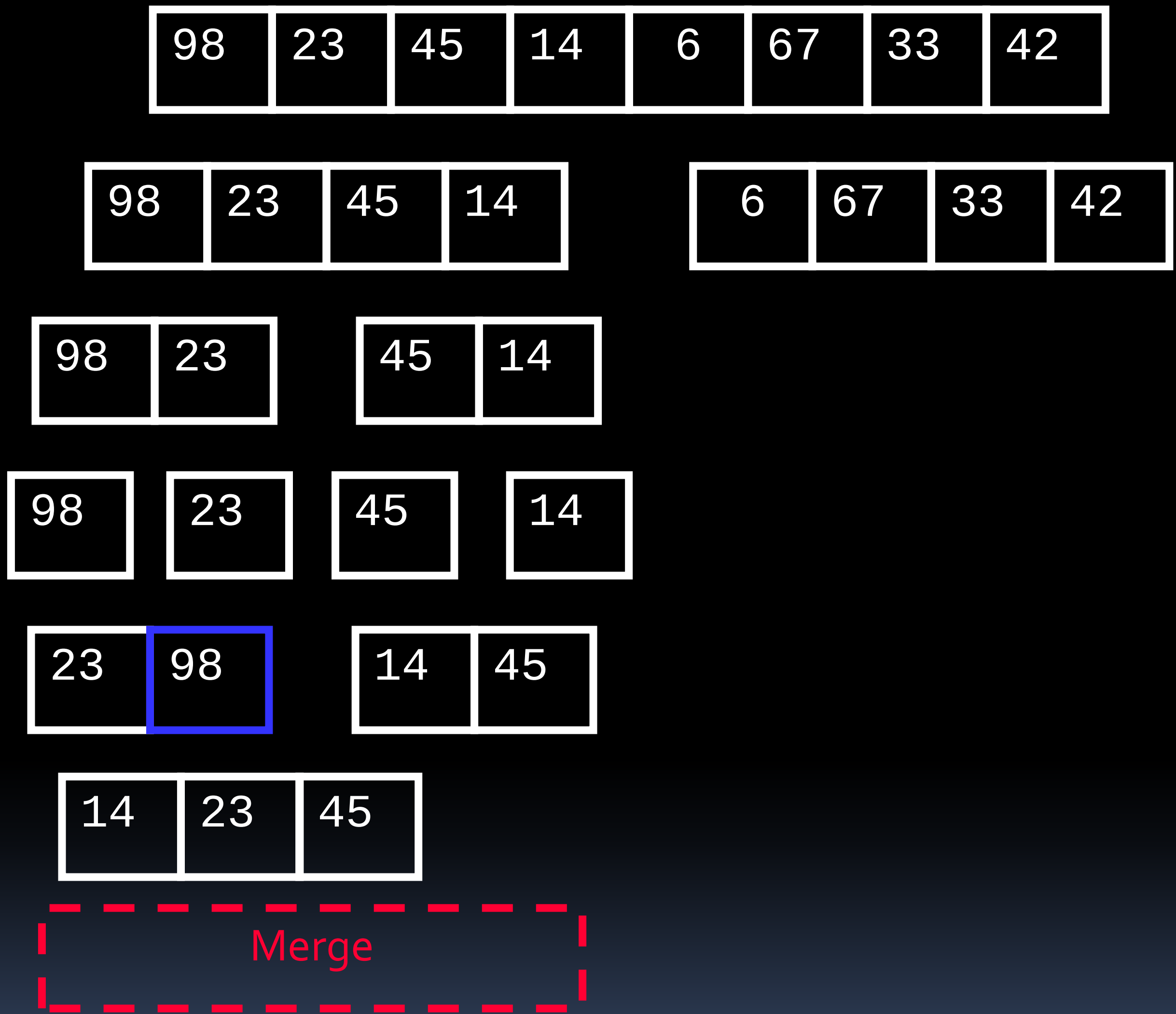
14	45

Merge

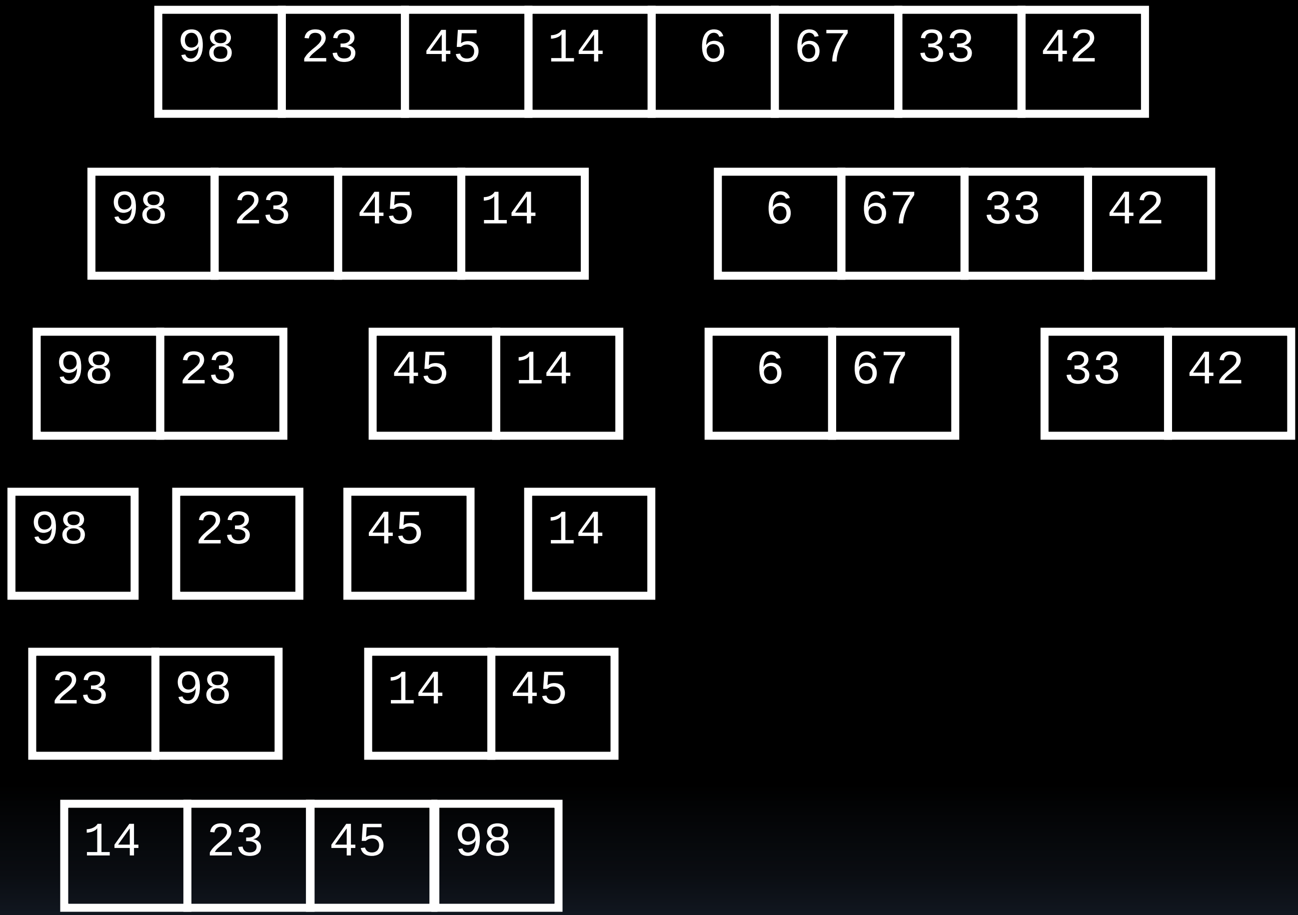

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
Merge

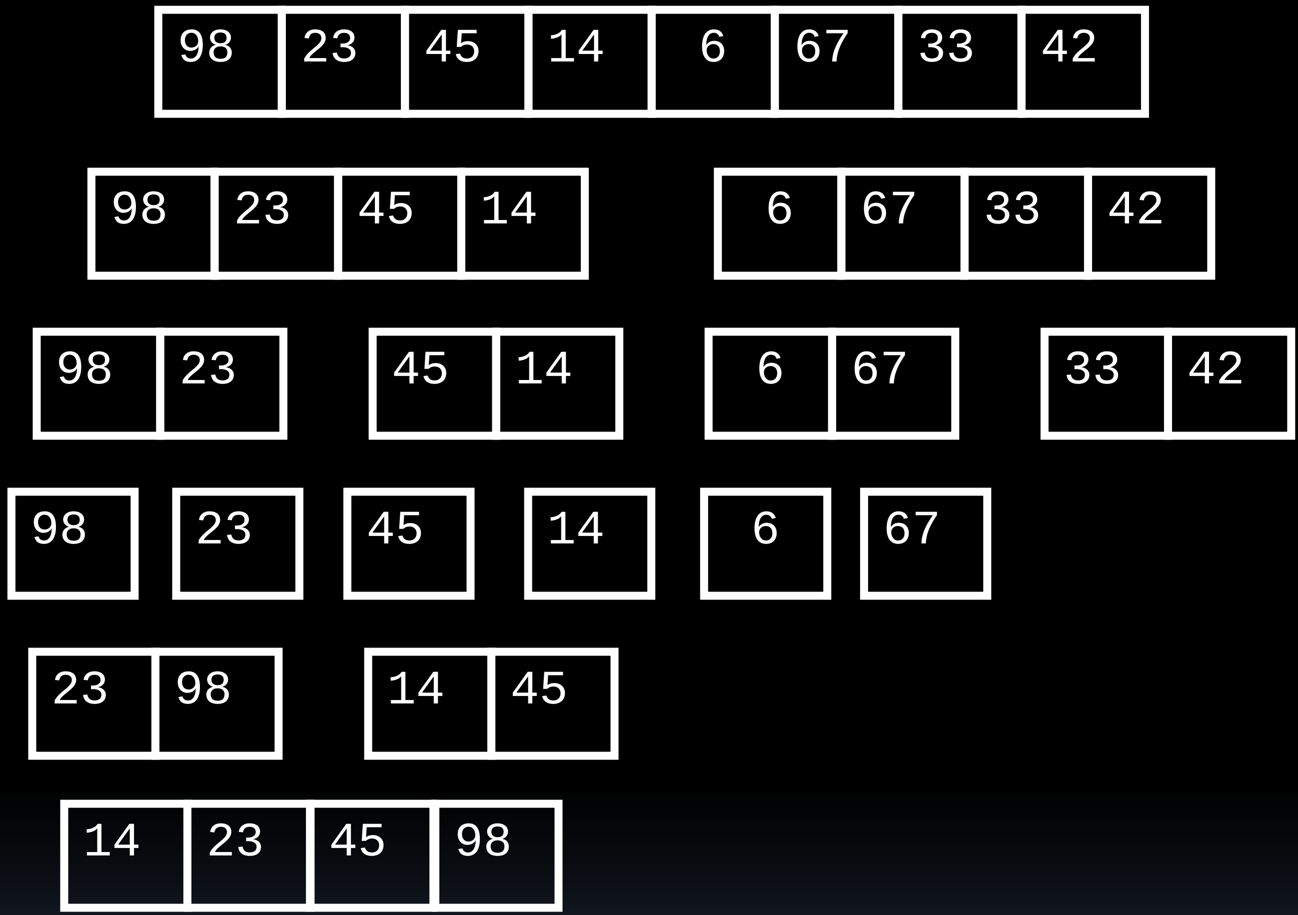

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14
Merge

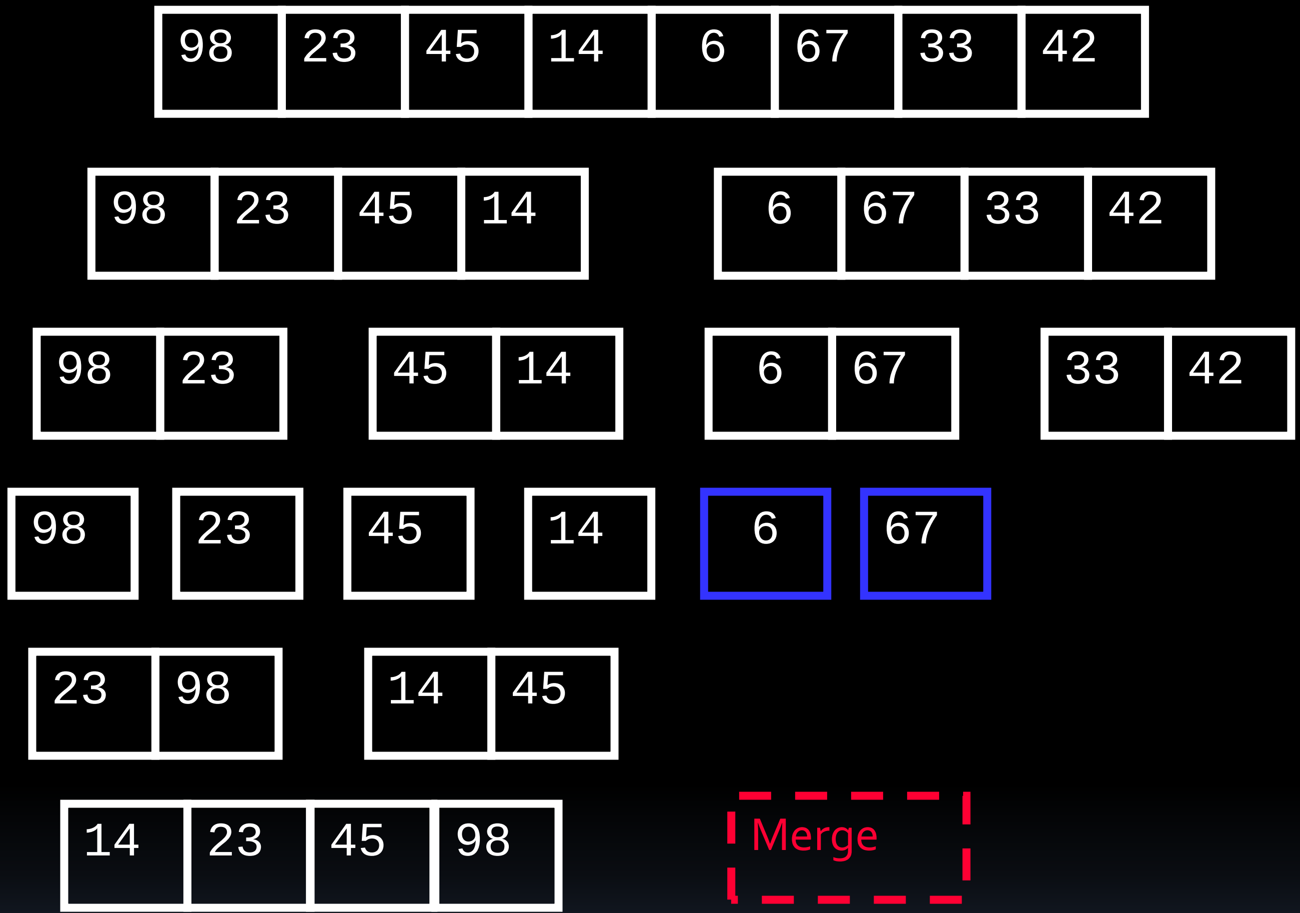

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23
Merge


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

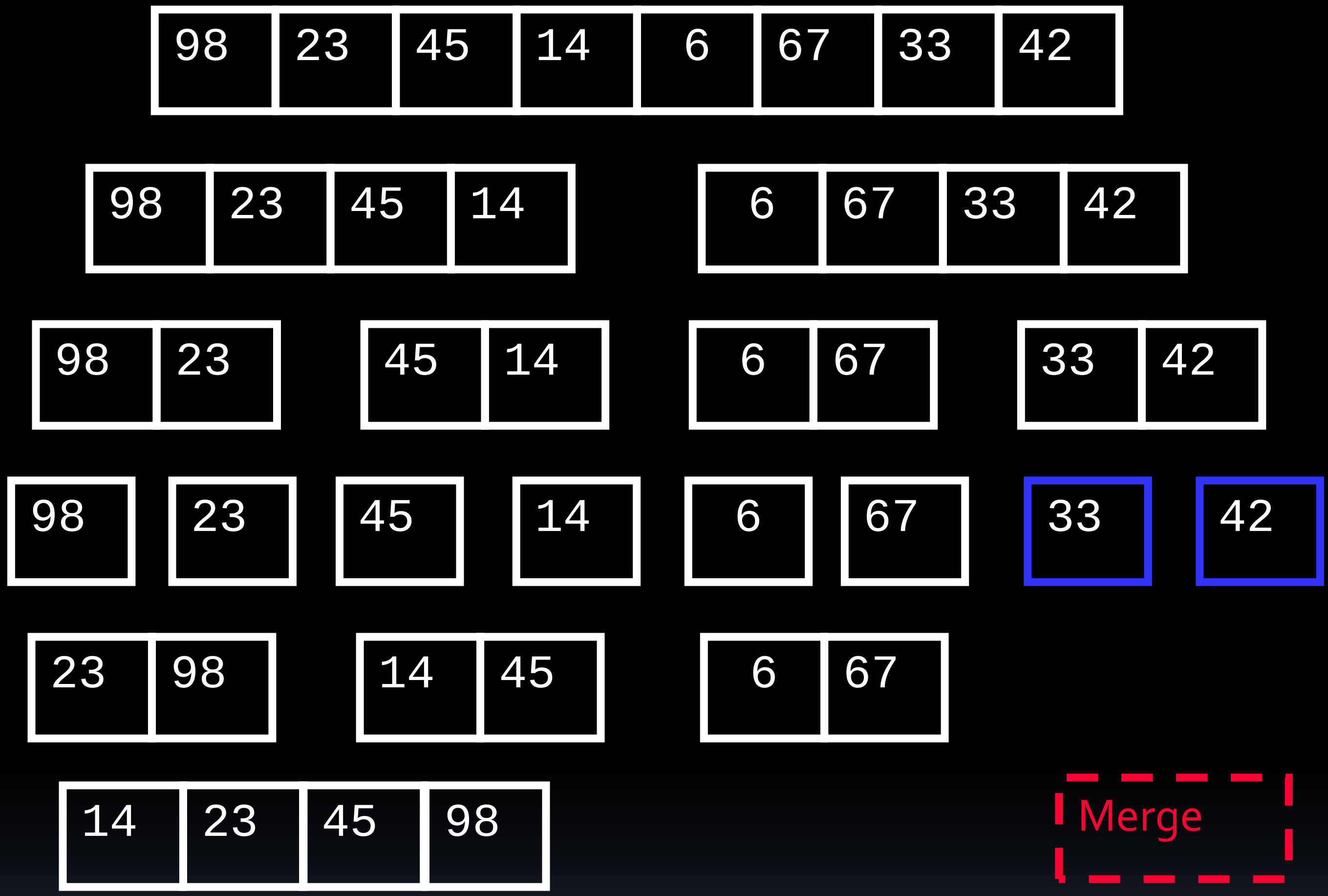
98 23 45 14 | 6 67 33 42

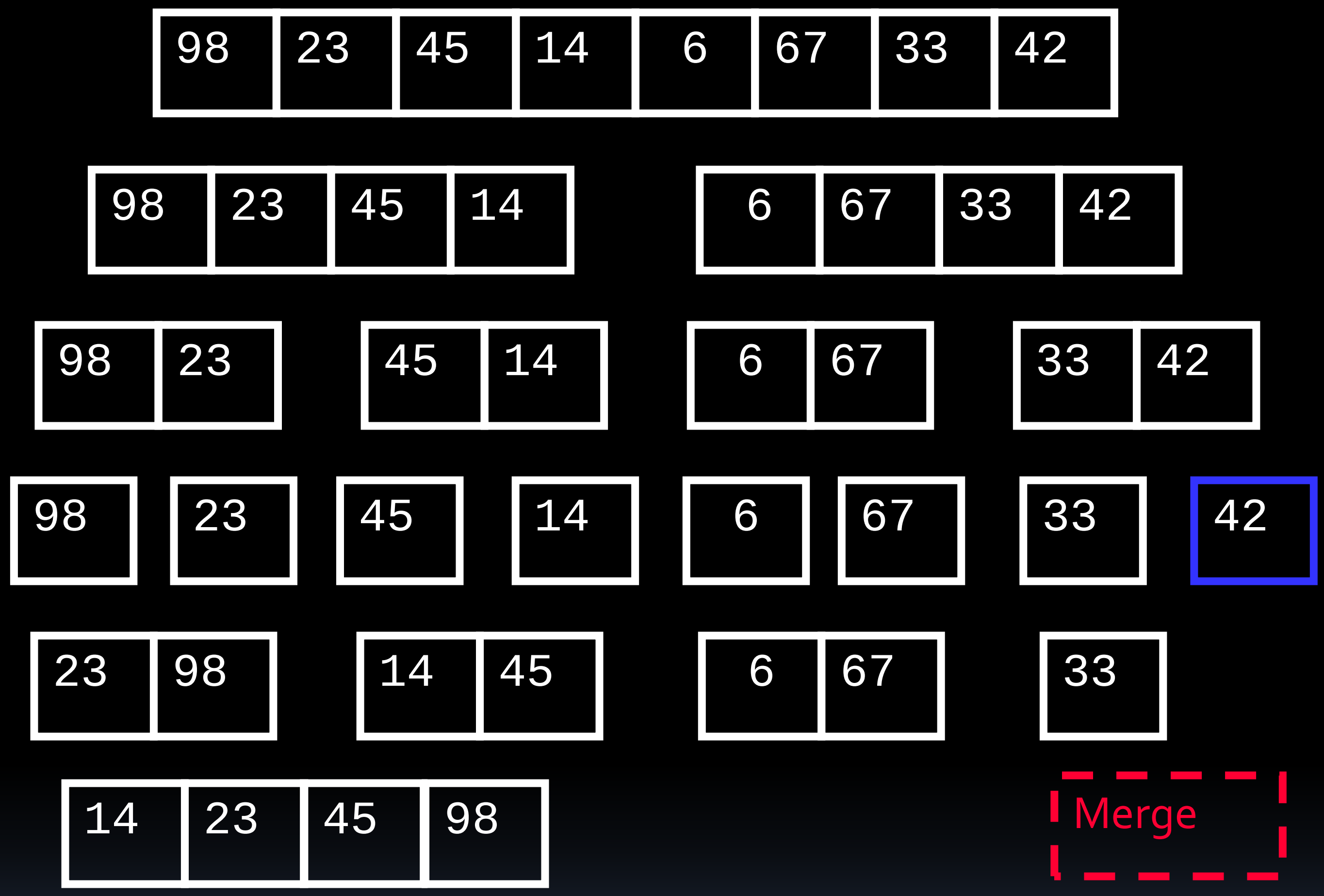
98 23 | 45 14 | 6 67 | 33 42

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

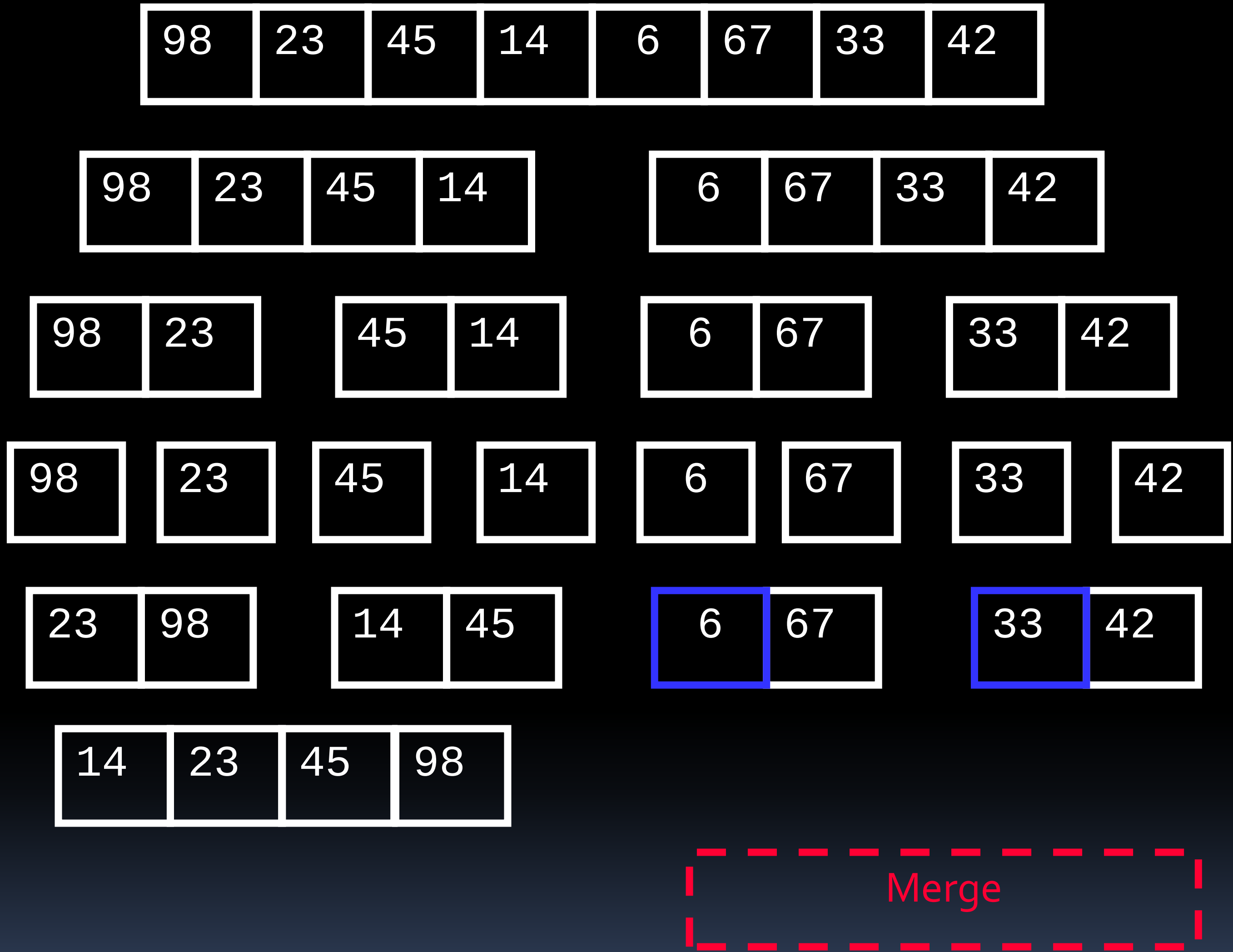
14	45

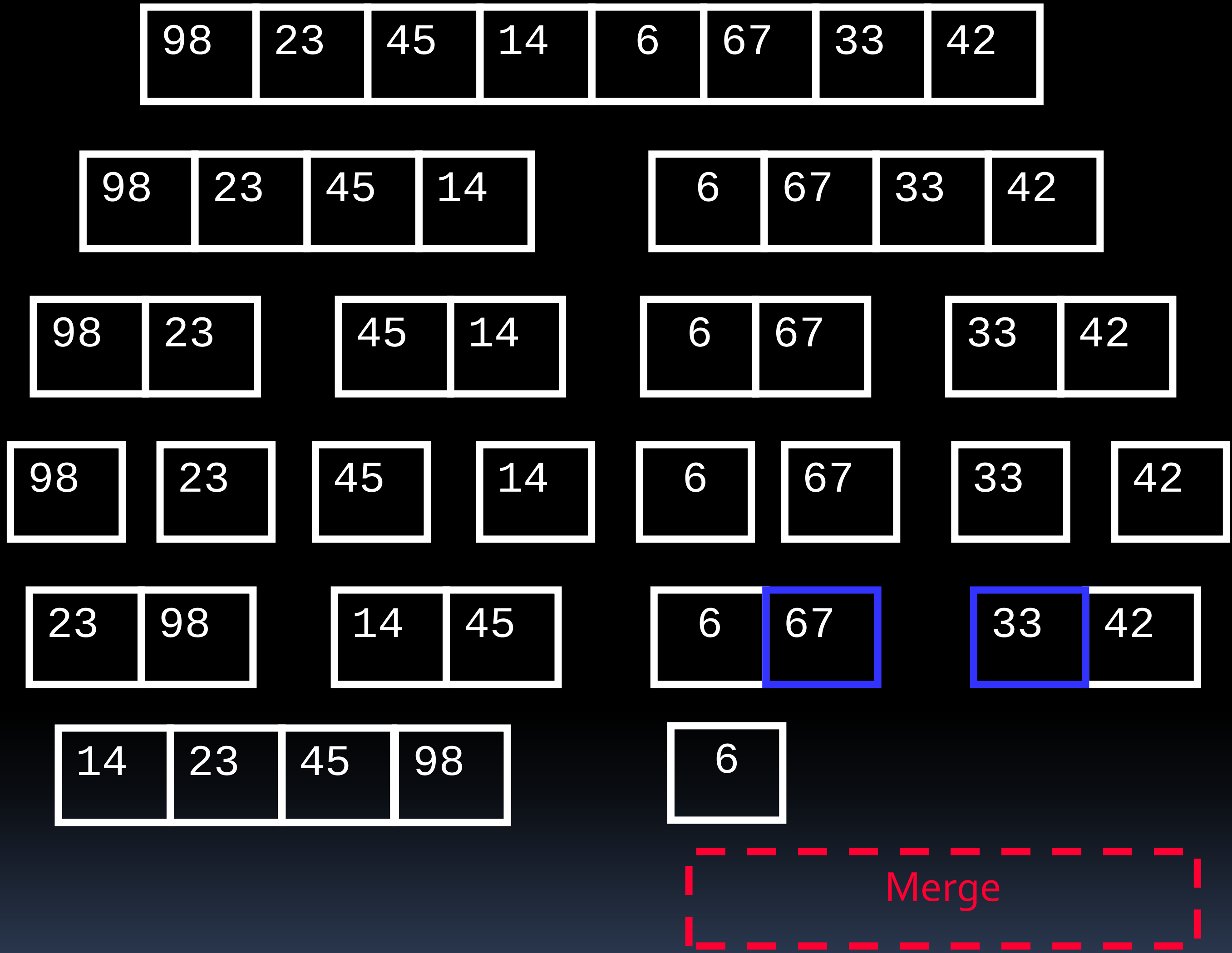
6	67

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

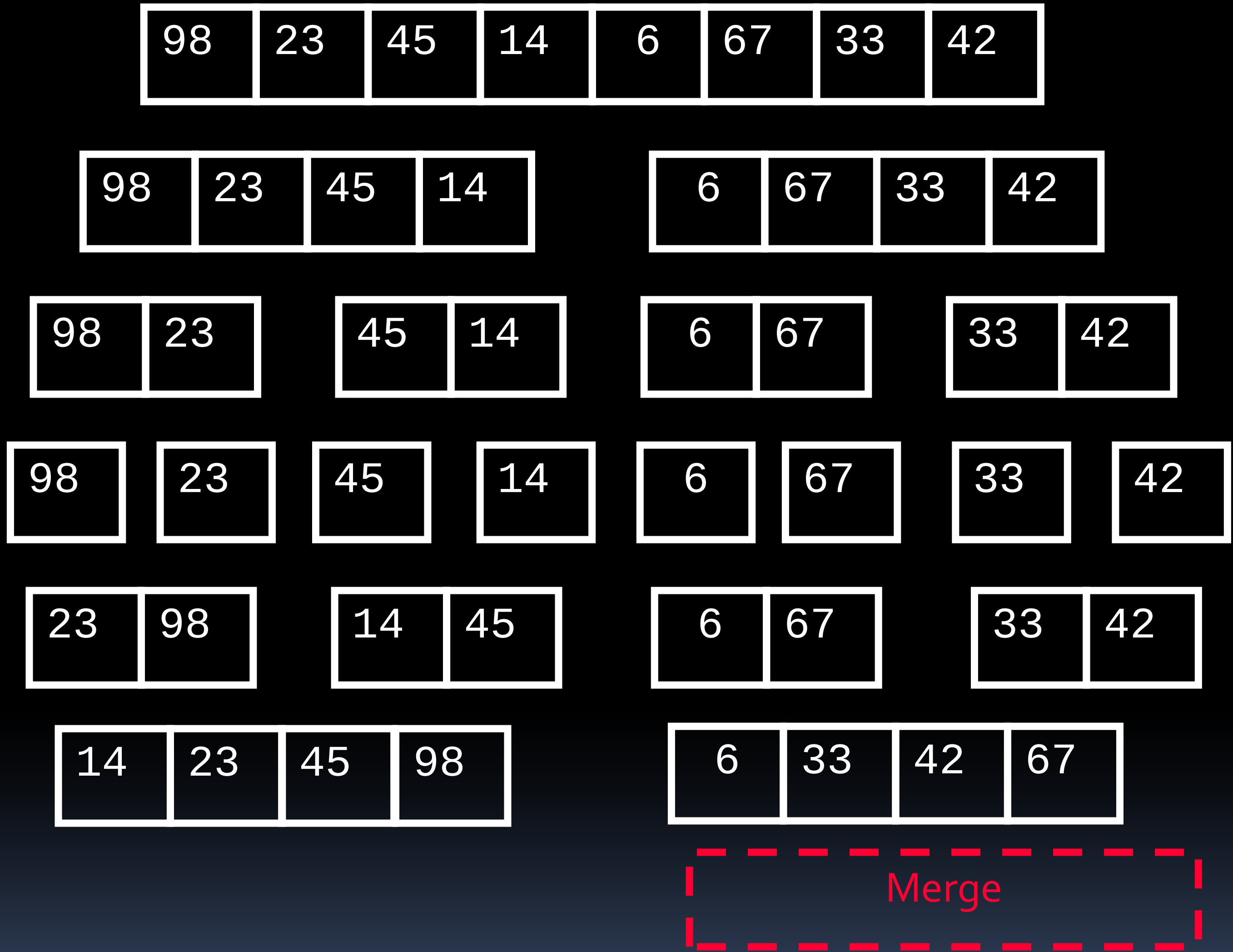
98 23 | 45 14 | 6 67 | 33 42

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

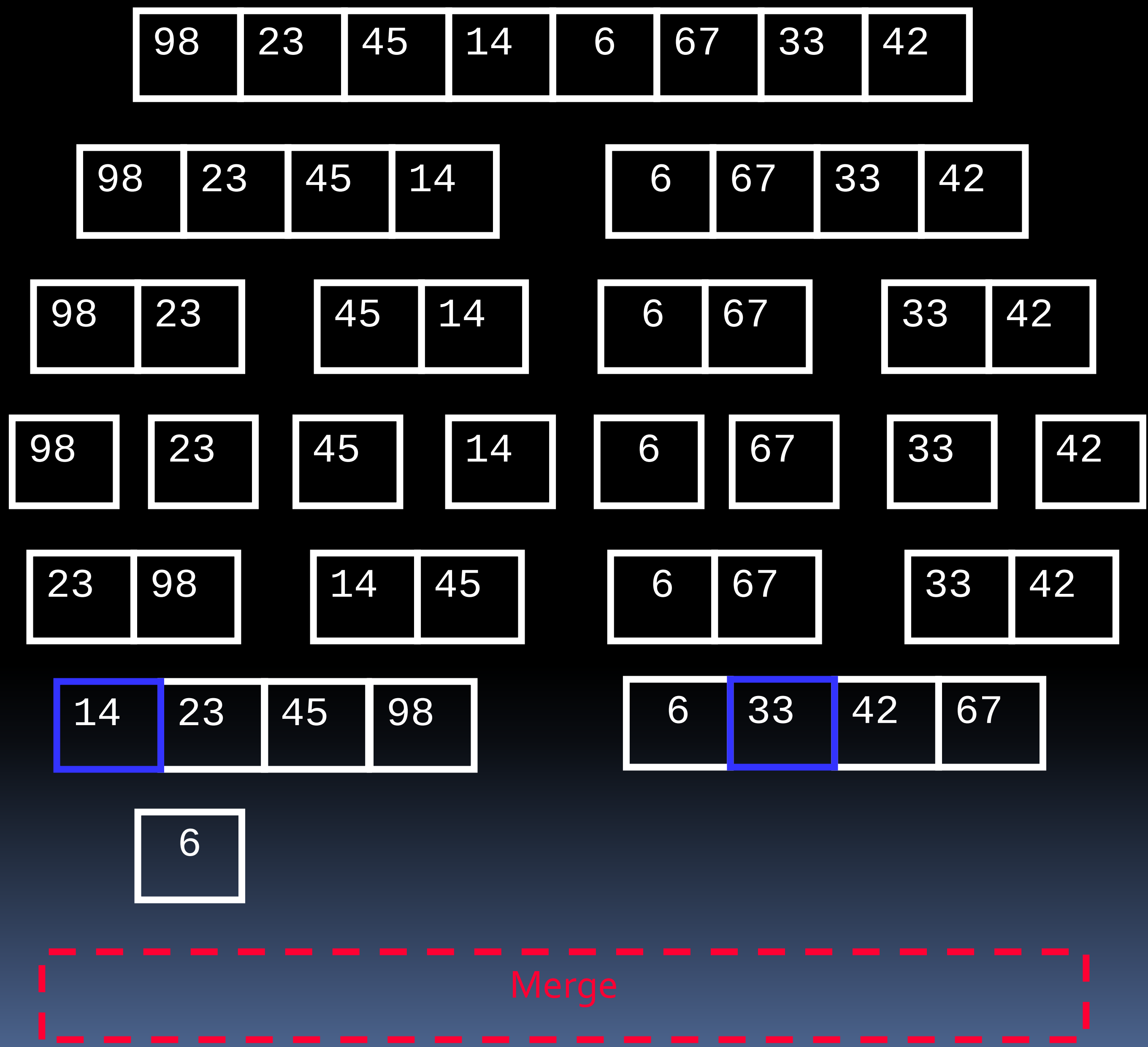
23 98 | 14 45 | 6 67 | 33 42

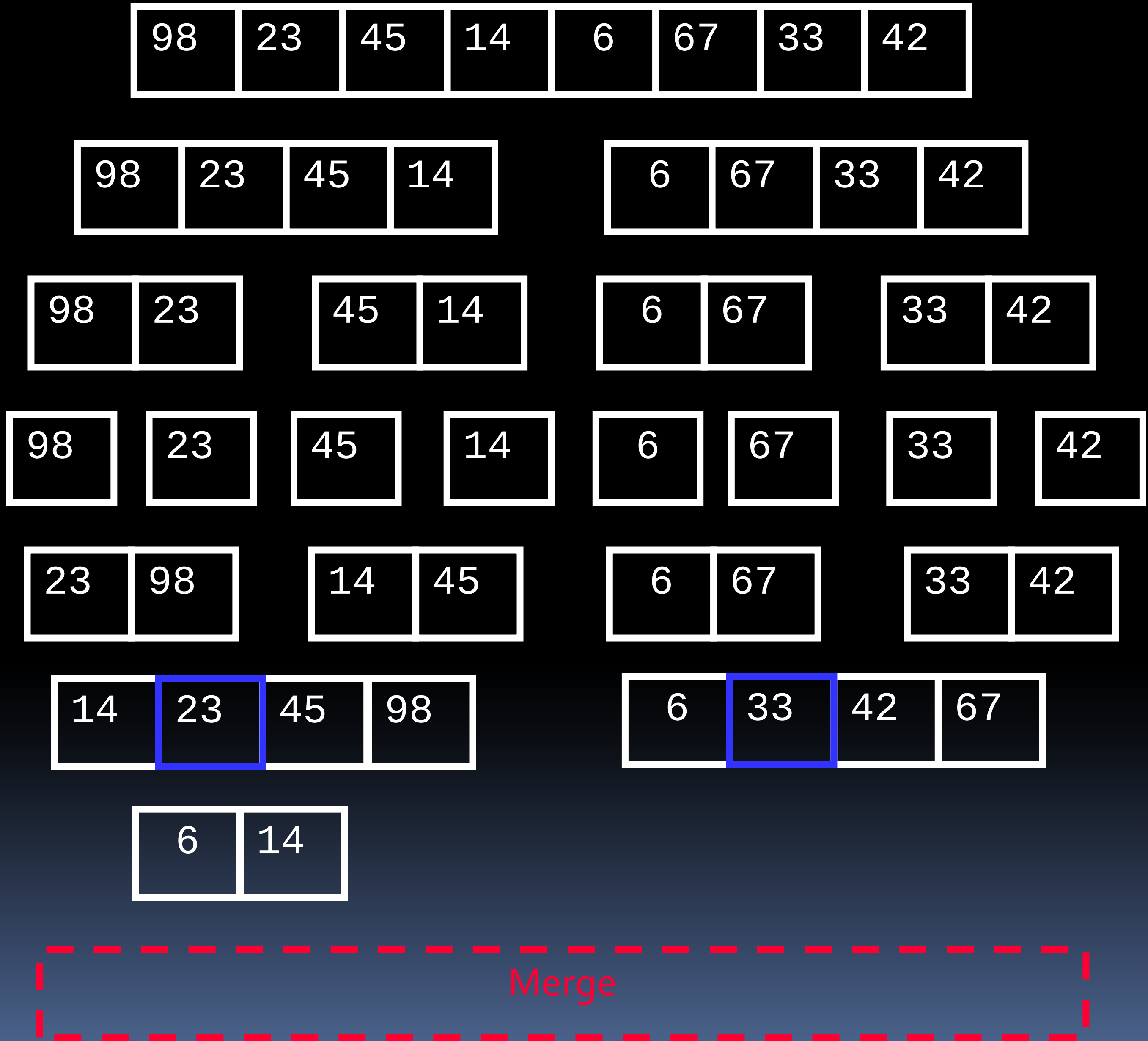
14 23 45 98 | 6 33 42

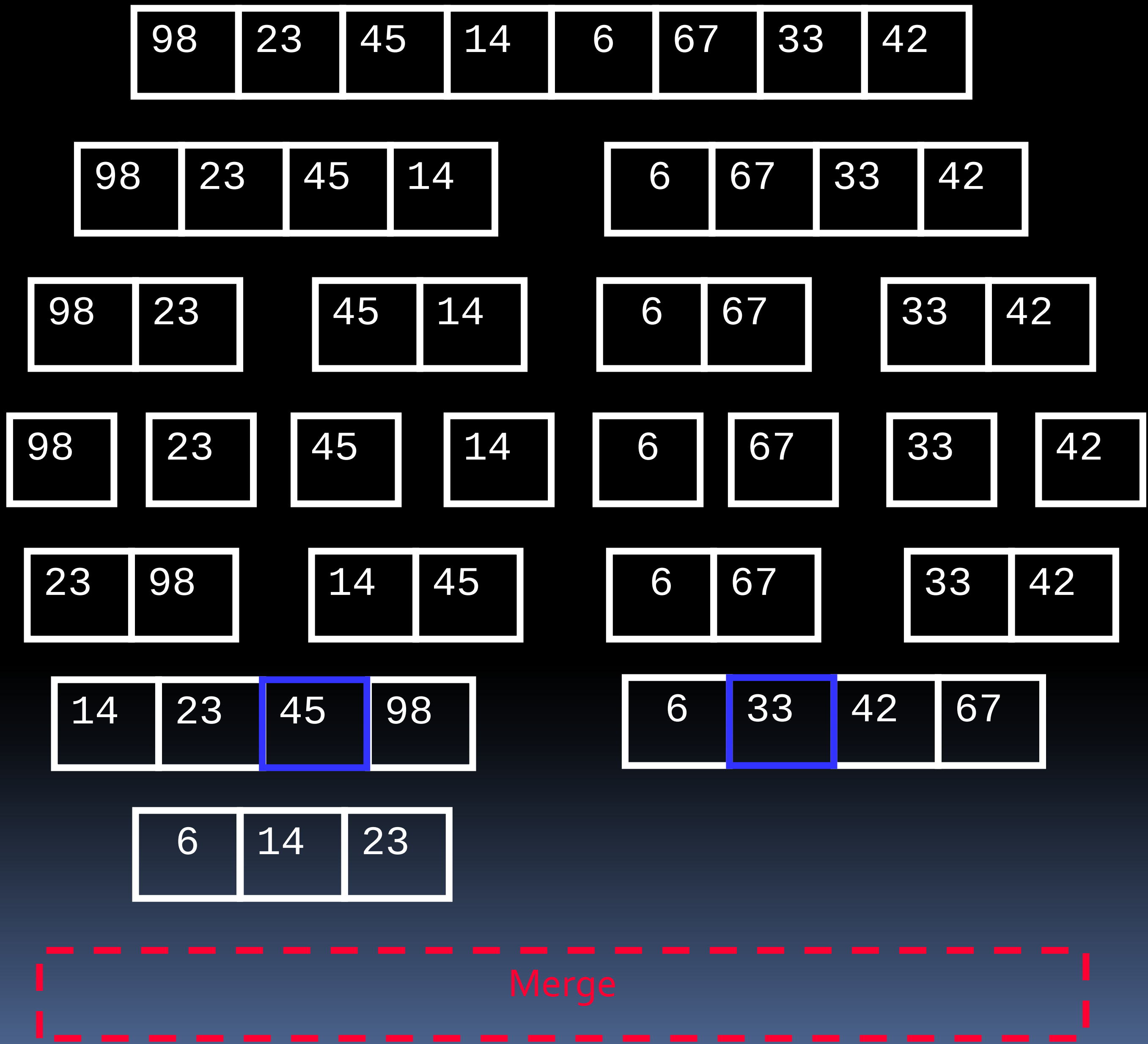
Merge

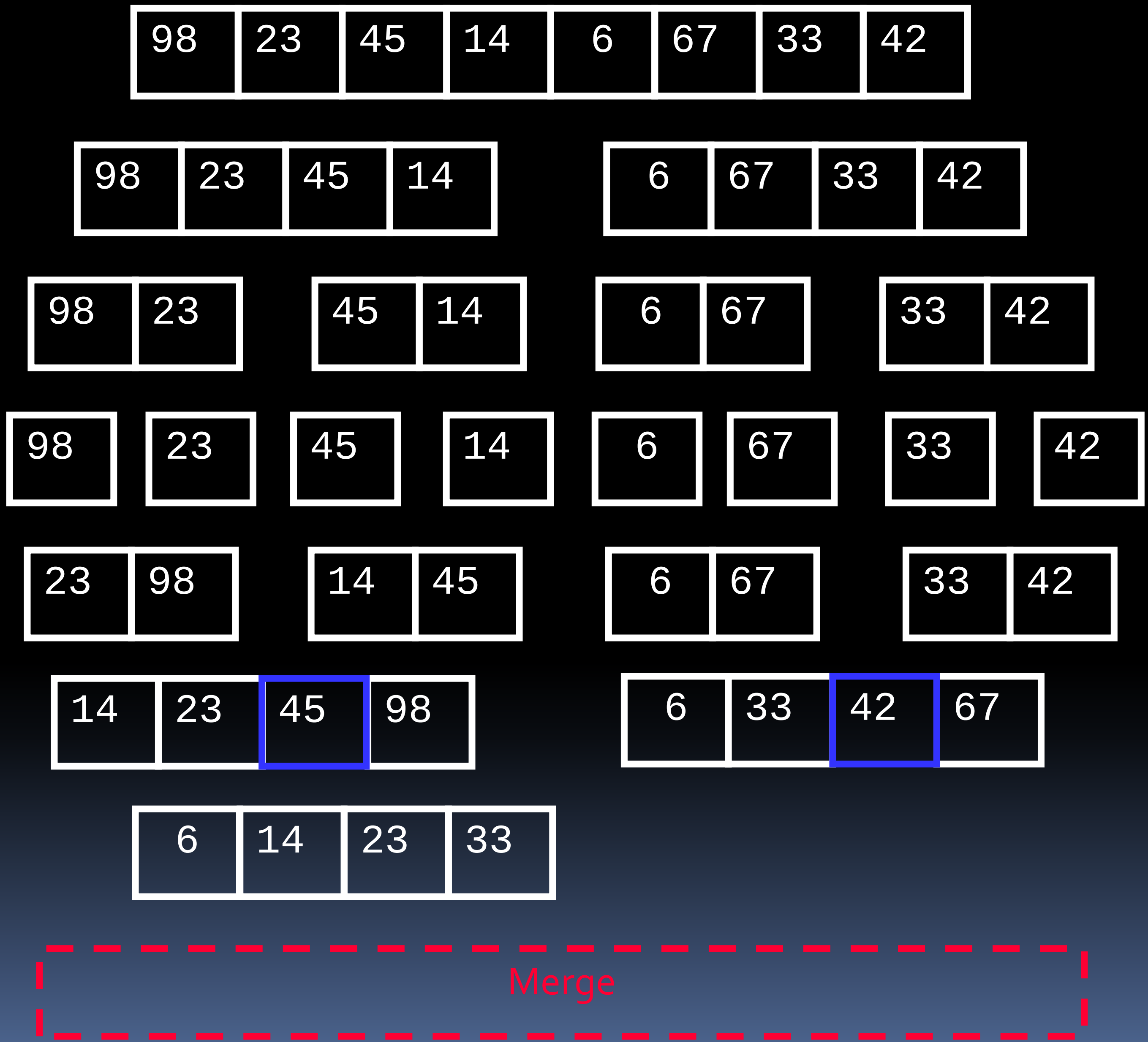

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
Merge

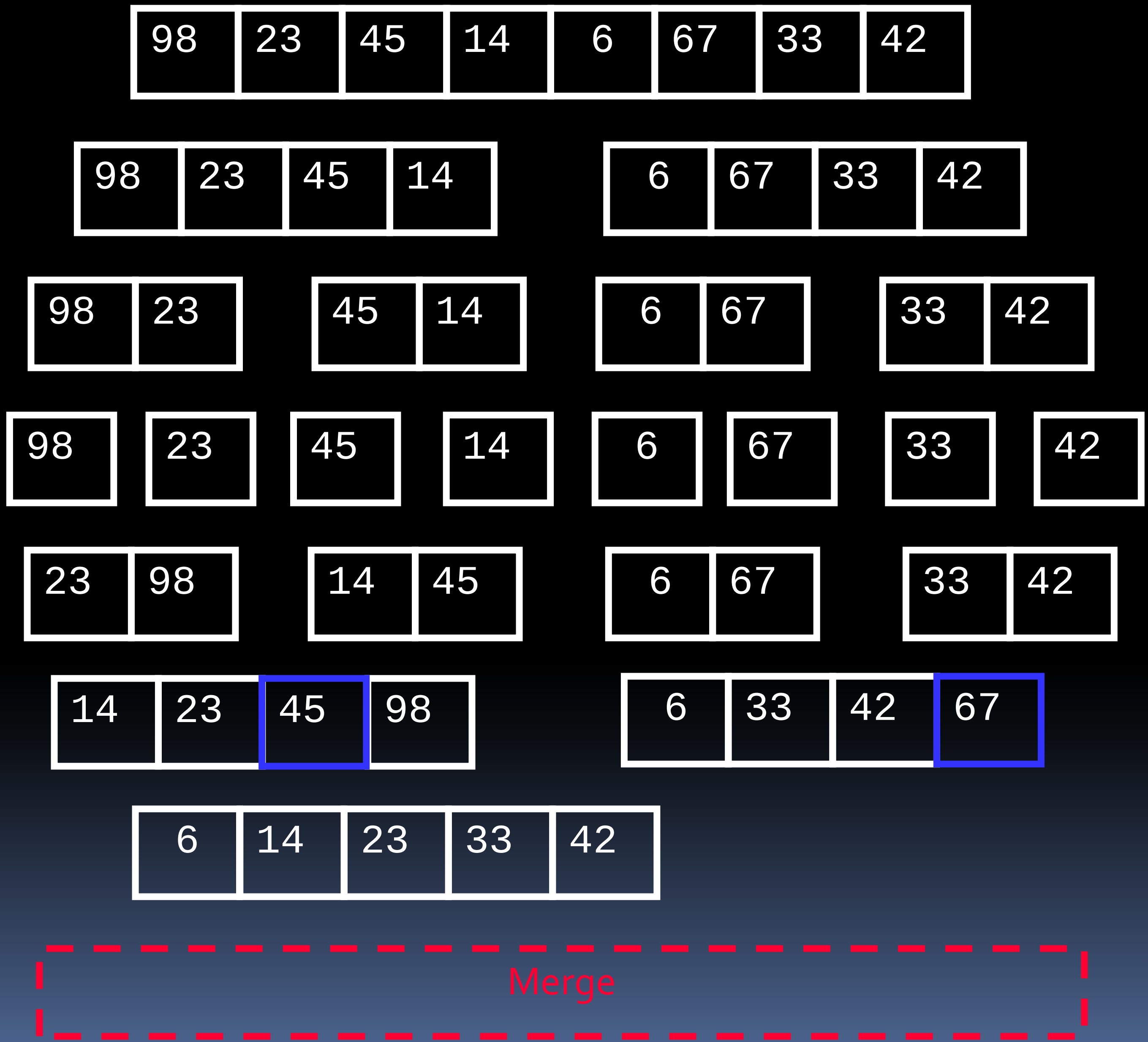

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
Merge


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]

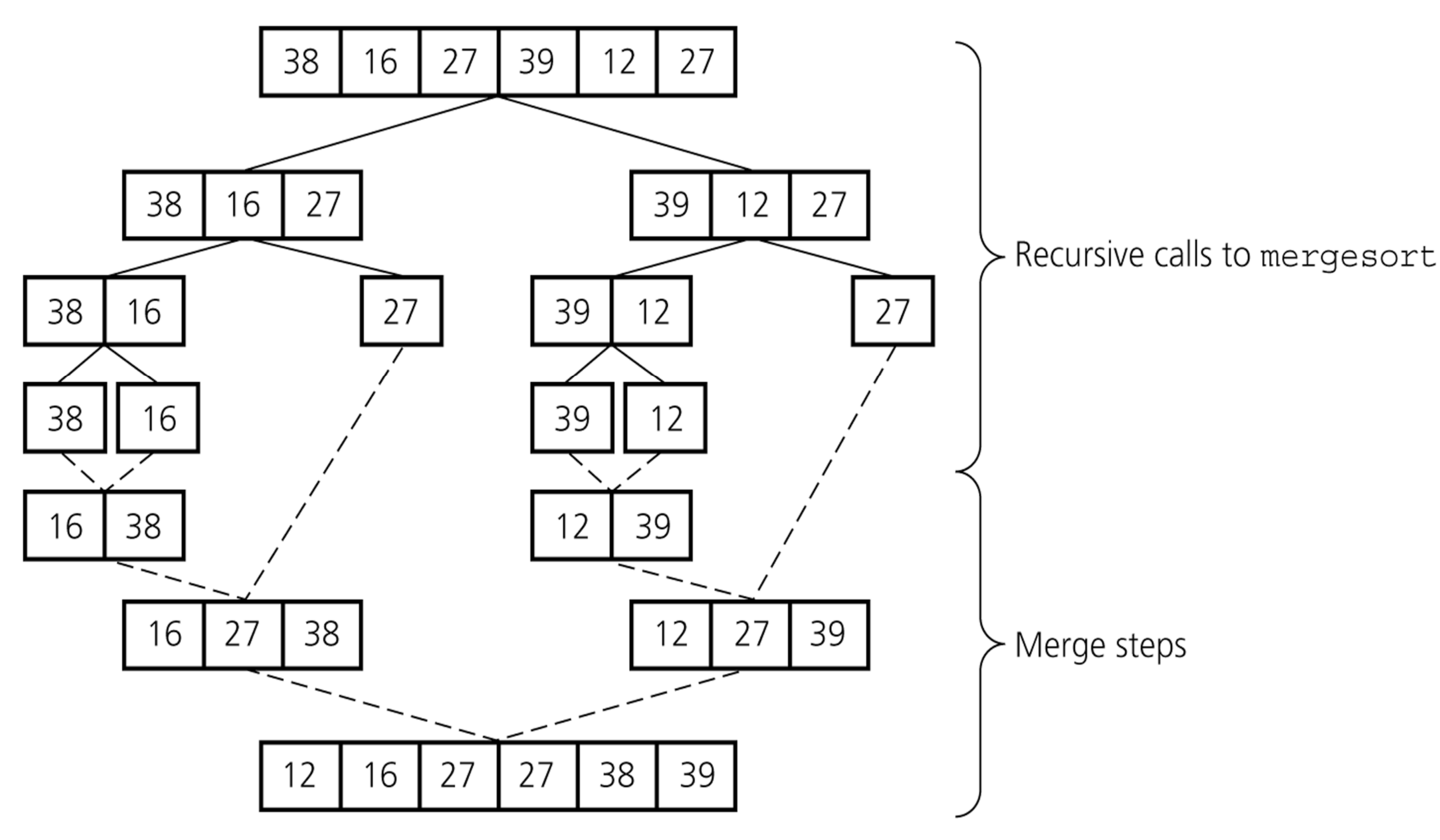
[23 | 98]   [14 | 45]   [6 | 67]   [33 | 42]

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

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

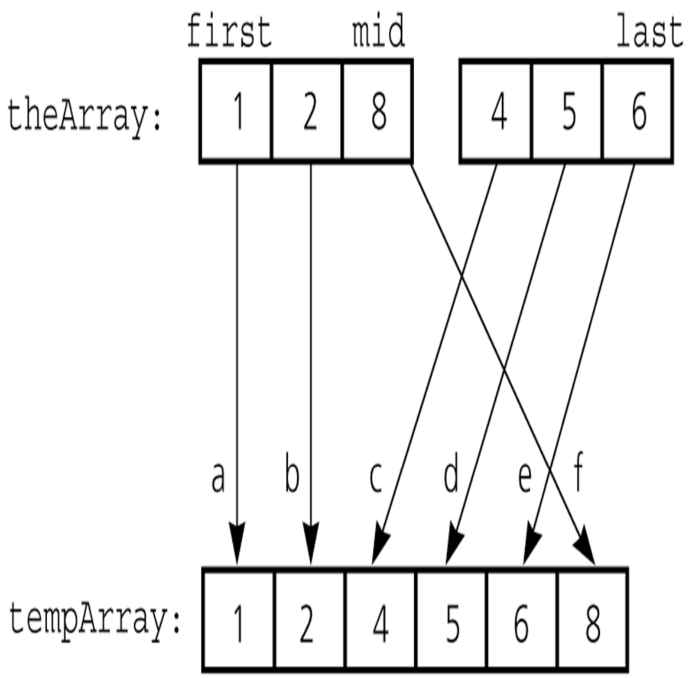
Merge

Mergesort – Example



Mergesort–Analysis of Merge

A worst-case instance of the merge step in *mergesort*

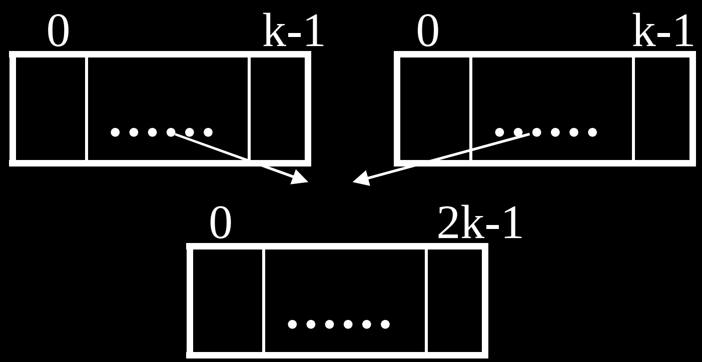


Merge the halves:

a. 1 < 4, so move 1 from theArray[first..mid] to tempArray
b. 2 < 4, so move 2 from theArray[first..mid] to tempArray
c. 8 > 4, so move 4 from theArray[mid+1..last] to tempArray
d. 8 > 5, so move 5 from theArray[mid+1..last] to tempArray
e. 8 > 6, so move 6 from theArray[mid+1..last] to tempArray
f. theArray[mid+1..last] is finished, so move 8 to tempArray

Mergesort – Analysis of Merge

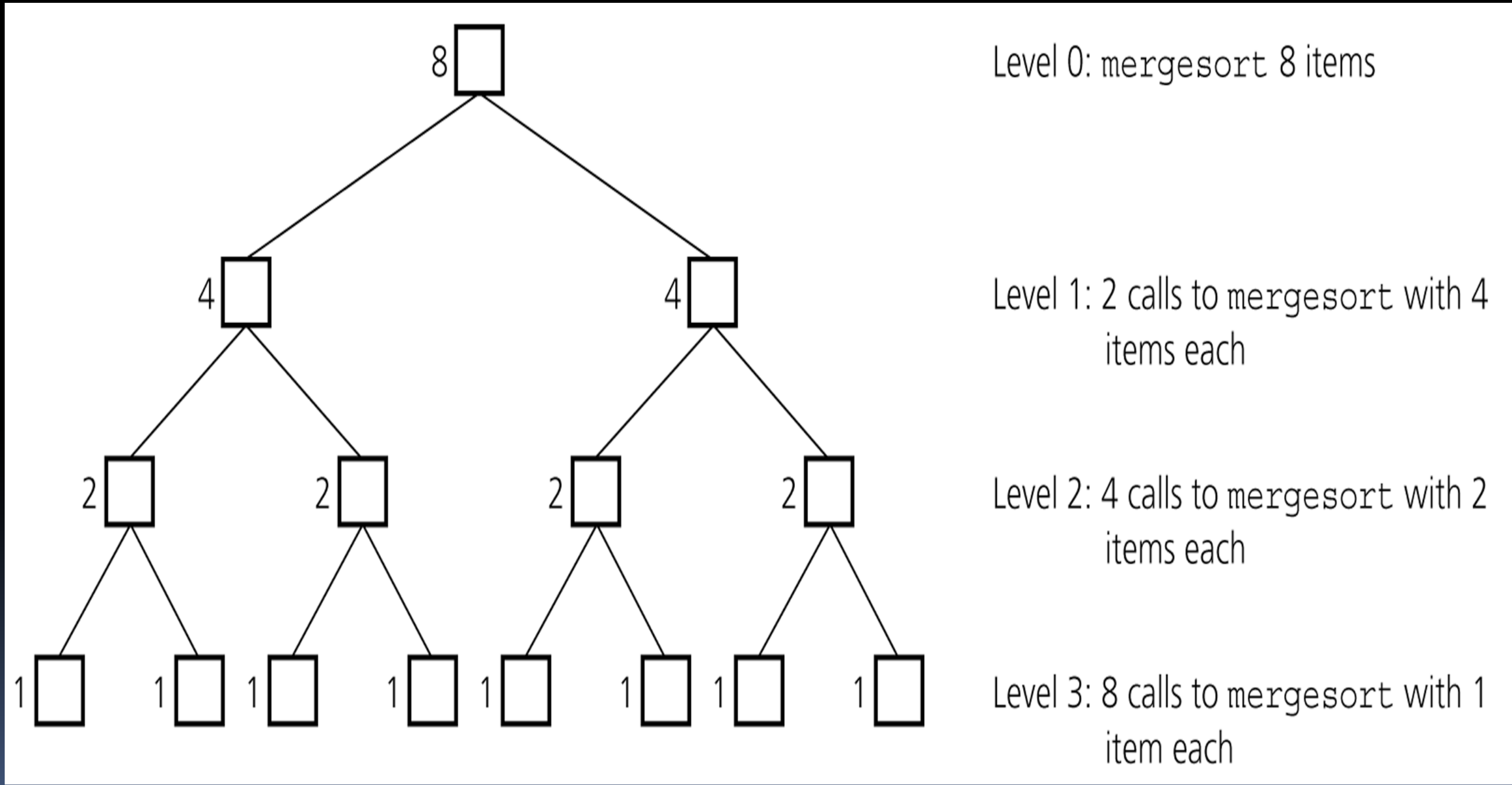
Merging two sorted arrays of size ***k***



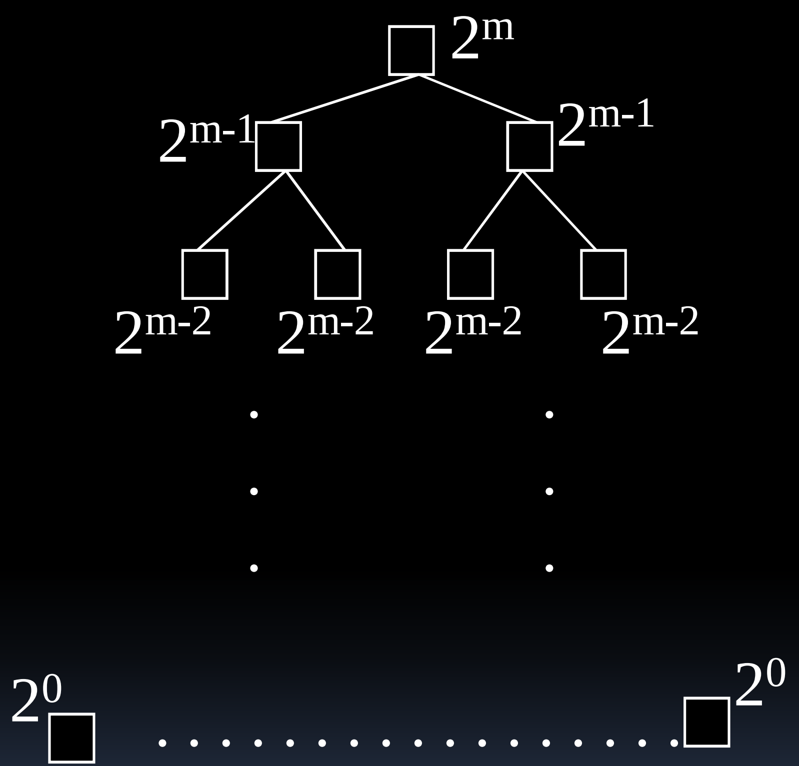
- ***Best-case:***
  - All the elements in the first array are smaller (or larger) than all the elements in the second array.
  - The number of moves: 2k + 2k
  - The number of key comparisons: k
- ***Worst-case:***
  - The number of moves: 2k + 2k
  - The number of key comparisons: 2k-1

Mergesort - Analysis

Levels of recursive calls to *mergesort*, given an array of eight items



Mergesort - Analysis



level 0 : 1 merge (size 2^{m-1})

level 1 : 2 merges (size 2^{m-2})

level 2 : 4 merges (size 2^{m-3})

level m-1 : 2^{m-1} merges (size 2^0)

level m

Mergesort - Analysis

- ***Worst-case –***

The number of key comparisons:

$= 2^0*(2*2^{m-1}-1) + 2^1*(2*2^{m-2}-1) + \ldots + 2^{m-1}*(2*2^0-1)$

$= (2^m - 1) + (2^m - 2) + \ldots + (2^m – 2^{m-1})$ (m terms)

2^i

$= m*2^m –$

$= m*2^m – 2^m – 1$

Using m = log n

$\mathbf{= n * \log_2 n – n – 1}$

➔ **$O(n * \log_2 n)$**

Sorting Algorithms

Quick Sort

Quick sort

- Quick sort is more widely used than any other sort.
- Quick sort is well-studied, not difficult to implement, works well on a variety of data, and consumes fewer resources that other sorts in nearly all situations.
- Quick sort is O(n*log n) time, and O(log n) additional space due to recursion.

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

Q	U	I	C	K	S	O	R	T	I	S	C	O	O	L

partition element **unpartitioned** **left**

partitioned **right**

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

swap me

Q	U	I	C	K	S	O	R	T	I	S	C	O	O	L

partition element

unpartitioned

left

partitioned

right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

swap me

Q	U	I	C	K	S	O	R	T	I	S	C	O	O	L

partition element

unpartitioned

left

partitioned

right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

swap me

Q	U	I	C	K	S	O	R	T	I	S	C	O	O	L

partition element

unpartitioned

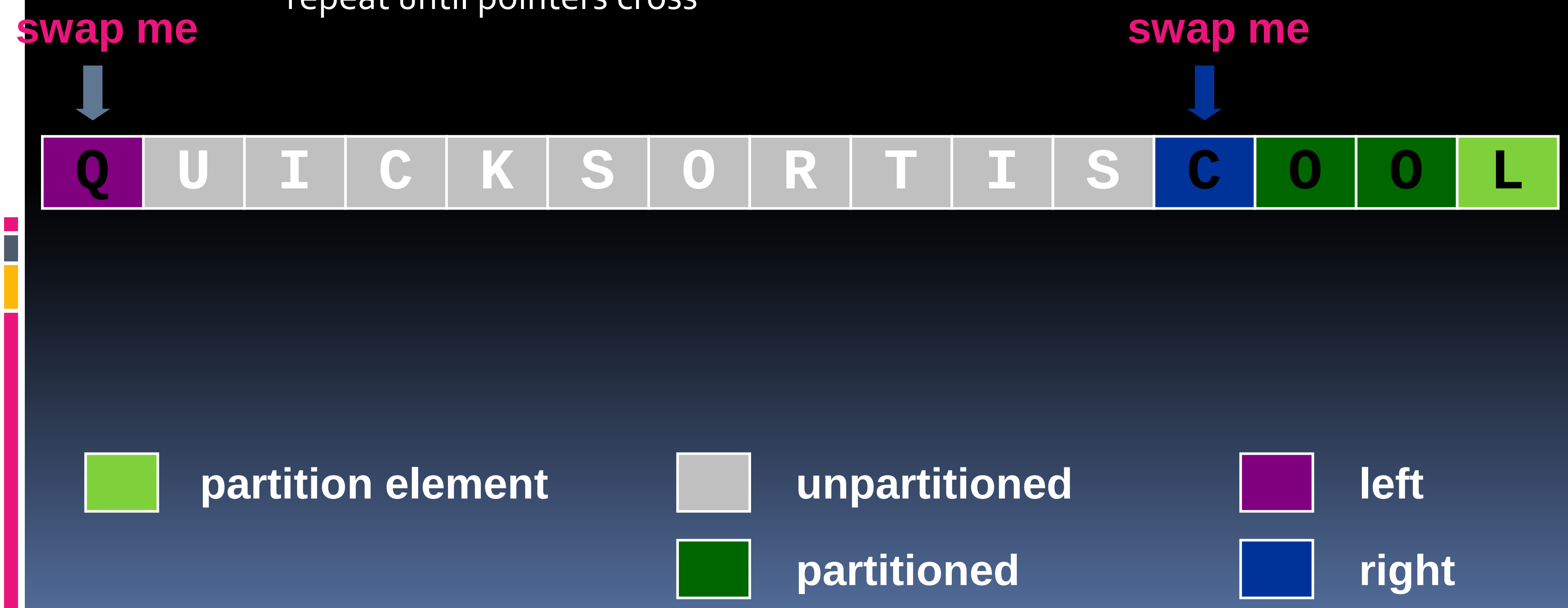
left

partitioned

right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross



Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

C	U	I	C	K	S	O	R	T	I	S	Q	O	O	L

partition element

unpartitioned

left

partitioned

right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

swap me

C	U	I	C	K	S	O	R	T	I	S	Q	O	O	L

partition element

unpartitioned

left

partitioned

right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

swap me

C	U	I	C	K	S	O	R	T	I	S	Q	O	O	L

partition element

unpartitioned

left

partitioned

right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

swap me **swap me**

C	U	I	C	K	S	O	R	T	I	S	Q	O	O	L

partition element **unpartitioned** **left**

partitioned **right**

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

partition element

unpartitioned

left

partitioned

right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

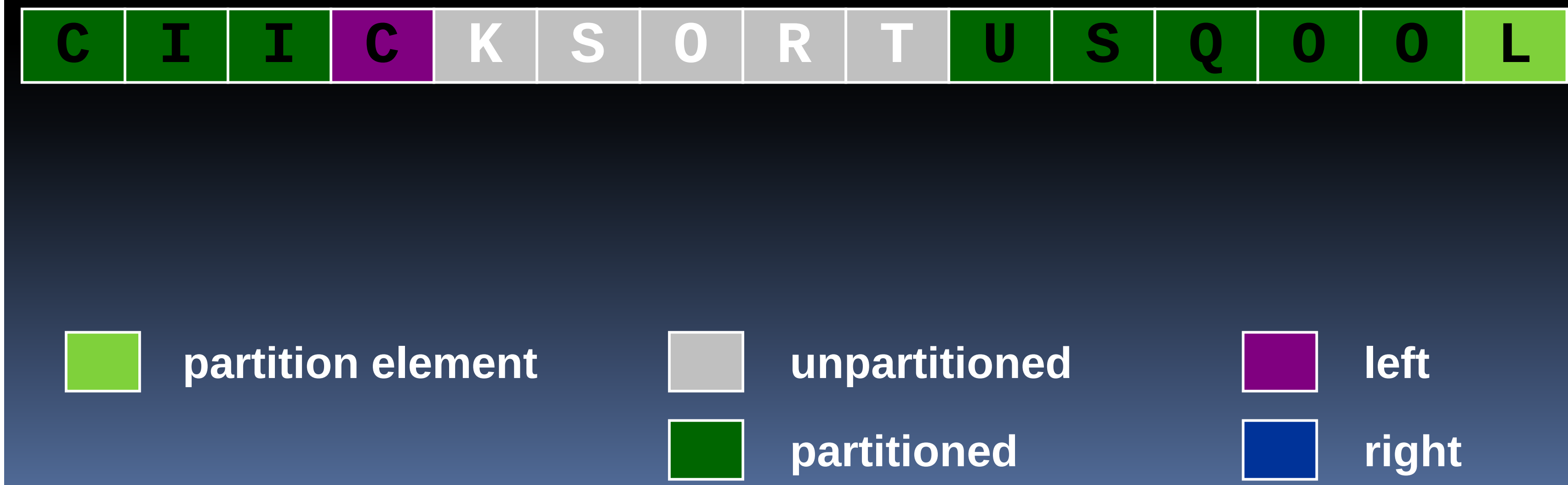
C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

partition element **unpartitioned** **left**

partitioned **right**

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross



Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - Exchange and repeat until pointers cross

C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

partition element

unpartitioned

left

partitioned

right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - Exchange and repeat until pointers cross

swap me

C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

partition element

unpartitioned

left

partitioned

right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - Exchange and repeat until pointers cross

swap me

C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

partition element

unpartitioned

left

partitioned

right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - Exchange and repeat until pointers cross

swap me

C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

- partition element
- unpartitioned
- left
- partitioned
- right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - Exchange and repeat until pointers cross

swap me

C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

partition element

unpartitioned

left

partitioned

right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - Exchange and repeat until pointers cross

pointers cross

swap with partitioning element

C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

partition element

unpartitioned

left

partitioned

right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - Exchange and repeat until pointers cross

partition is complete

C	I	I	C	K	L	O	R	T	U	S	Q	O	O	S

partition element | **unpartitioned** | **left**

partitioned | **right**

Sorting Algorithms

Radix Sort

Radix Sort

- Sort N numbers, each with k bits
- E.g., input {4, 1, 0, 10, 5, 6, 1, 8}

Radix Sort

- Sort N numbers, each with k bits
- E.g. , input {4, 1, 0, 10, 5, 6, 1, 8}

4	0100	0100	0100	0000	0000	0
1	0001	0000	0000	1000	0001	1
0	0000	1010	1000	0001	0001	1
10	1010	0110	0001	0001	0100	4
5	0101	1000	0101	1010	0101	5
6	0110	0001	0001	0100	0110	6
1	0001	0101	1010	0101	1000	8
8	1000	0001	0110	0110	1010	10

lsb msb

Radix sort example

- To sort:
  - 123, 12, 313, 321, 212, 112, 221, 132, 131
- Pass 1 assignment to buckets:
  - 0:
  - 1: 321, 221, 131
  - 2: 12, 212, 112, 132
  - 3: 123, 313
- Concatenated result
  - 321, 221, 131, 12, 212, 112, 132, 123, 313

Pass 2

- From previous pass
  - 321, 221, 131, 212, 112, 132, 123, 313
- Pass 2 assignment to buckets:
  - 0:
  - 1: 12, 212, 112, 313
  - 2: 321, 221, 123
  - 3: 131, 132
- Concatenated result
  - 12, 212, 112, 313, 321, 221, 123, 131, 132

Pass 3

- From previous pass
  - 12, 212, 112, 313, 321, 221, 123, 131, 132
- Pass 3 assignment to buckets:
  - 0: 12
  - 1: 112, 123, 131, 132
  - 2: 212, 221
  - 3: 313, 321
- Concatenated result
  - 12, 112, 123, 131, 132, 212, 221, 313, 321

THANK YOU