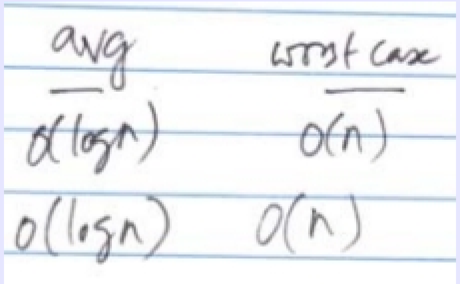
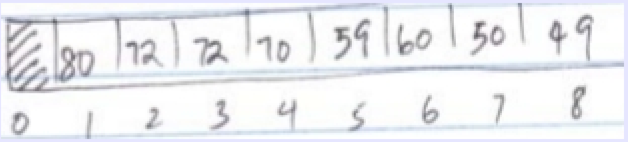
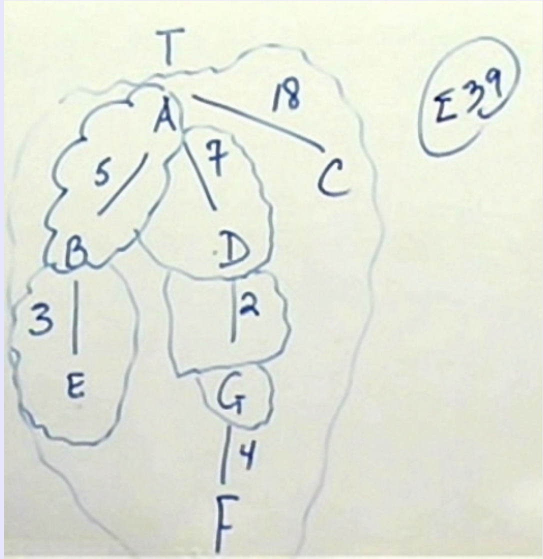
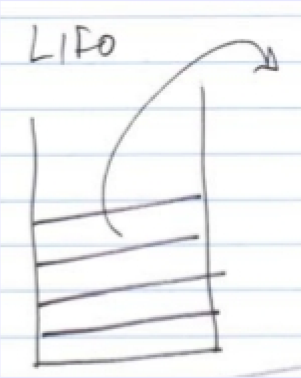
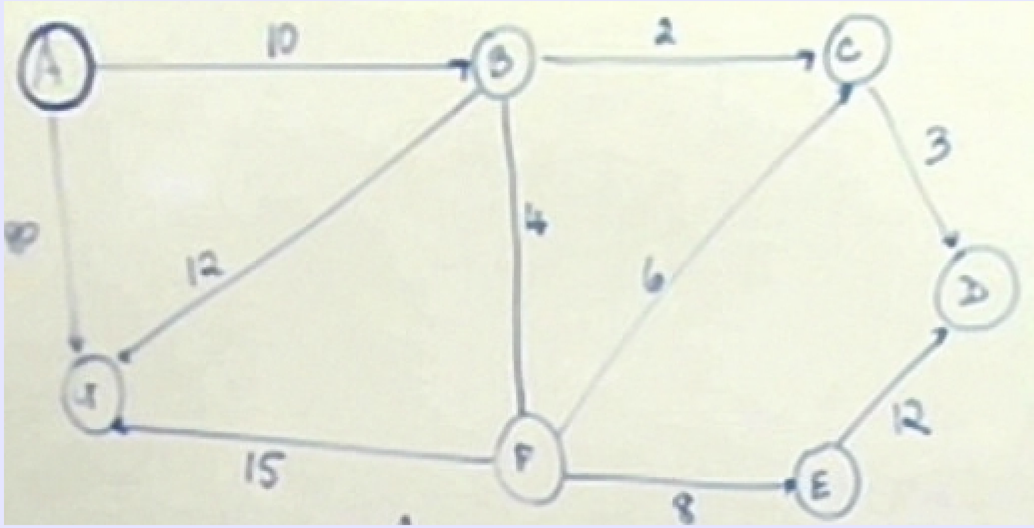










CS 401: Introduction to Advanced Studies (Data Structures)

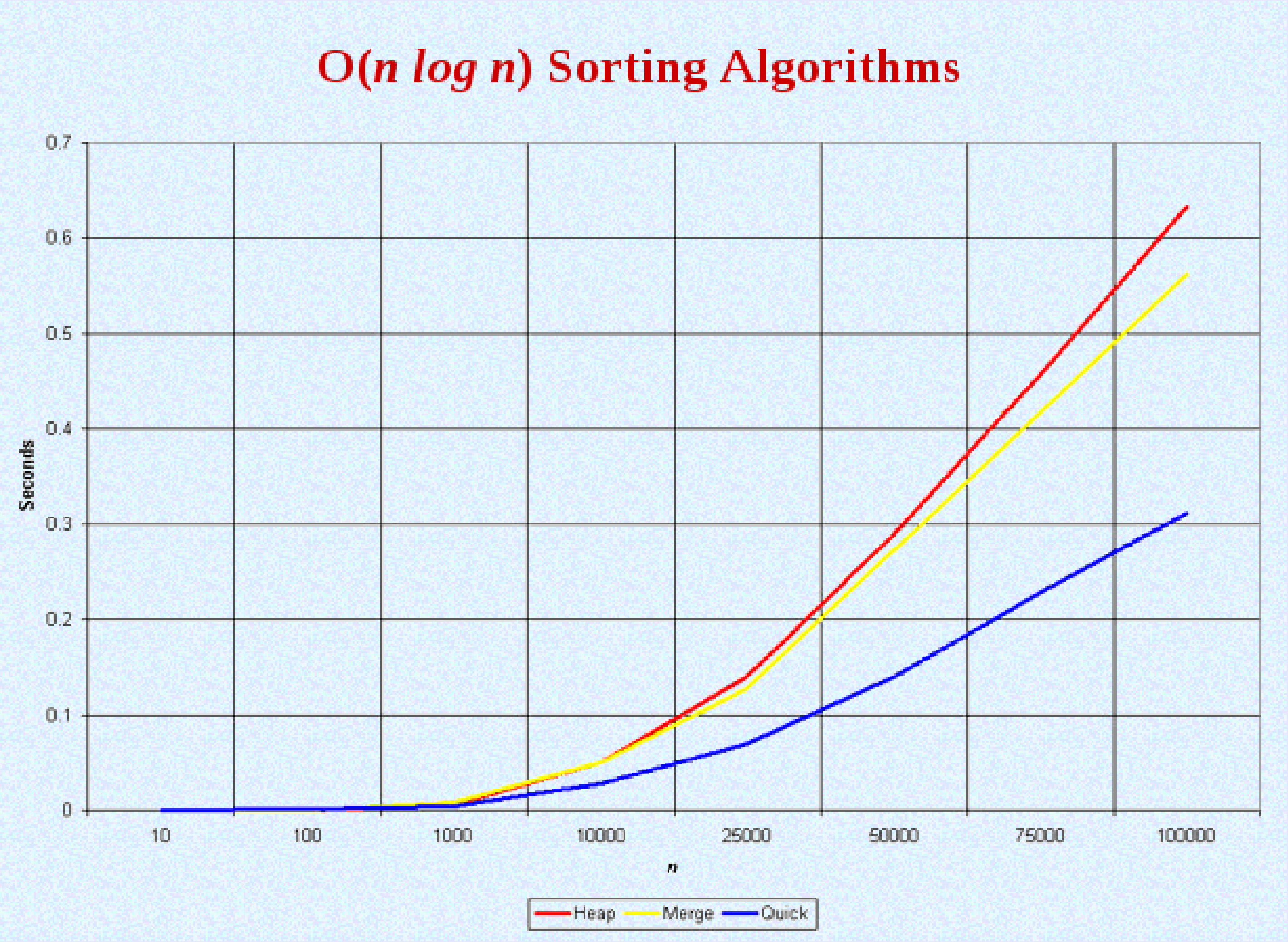
Vijay K. Gurbani, Ph.D., Illinois Institute of Technology

Lecture 12: **Sorting II --- Divide and Conquer Sorts**

Sorting: Divide and Conquer Sorts

- Heap sort
- Merge sort
- Quick sort

Divide and Conquer Sort Performance



Slide source: http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/sortingIntro.htm

Heap sort

- Slides to appear.

Merge sort

- Start with an array of size *n*.
- Recurse: Keep dividing array by half until *k* elements in each subarray (generally, *k* = *7*).
- Use insertion sort on arrays of size *k*.
- Merge arrays of size *k.*
- Start unraveling recursion by merging subarrays.

Merge sort

```
function merge_sort(list m)  {
   // Base case
   if (length(m) <= 1)  {
      return m;
   }

   // Divide list in two equal-sized sublists and recurse
   var list left, right;
   var integer middle := length(m)/2
   left  := list[0:middle-1]
   right := list[middle:length(m)-1]

   merge_sort(left)
   merge_sort(right)

   // Conquer: merge sorted sublists
   return merge(left, right)
}
```

Merge sort

Function merge(left, right). Pre-condition: sublists must be sorted.

20	29	80		15	30	45

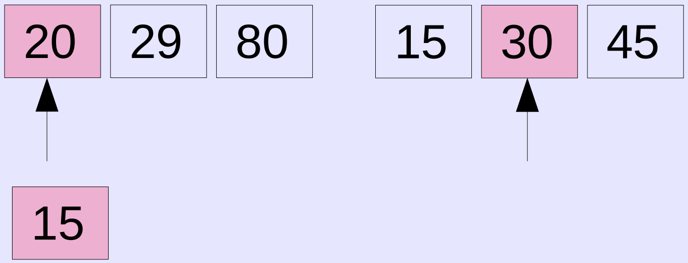
Merge sort

Function merge(left, right). Pre-condition: sublists must be sorted.

20	29	80		15	30	45

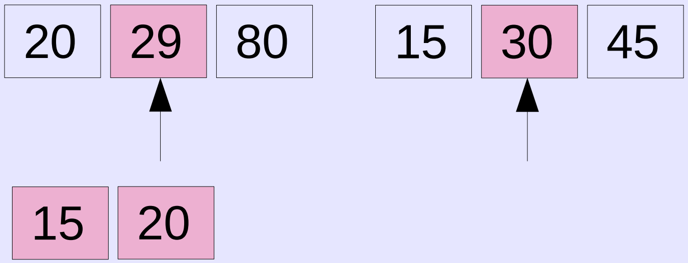
Merge sort

Function merge(left, right). Pre-condition: sublists must be sorted.



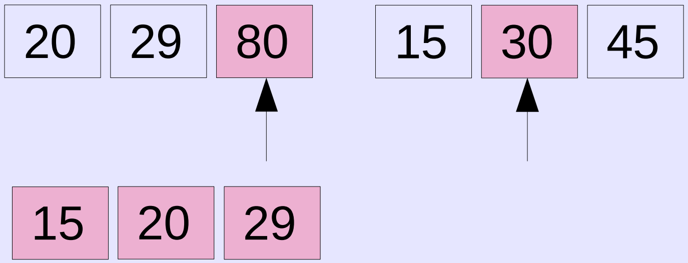
Merge sort

Function merge(left, right). Pre-condition: sublists must be sorted.



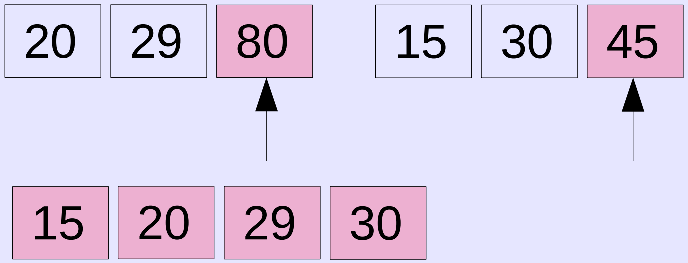
Merge sort

Function merge(left, right). Pre-condition: sublists must be sorted.



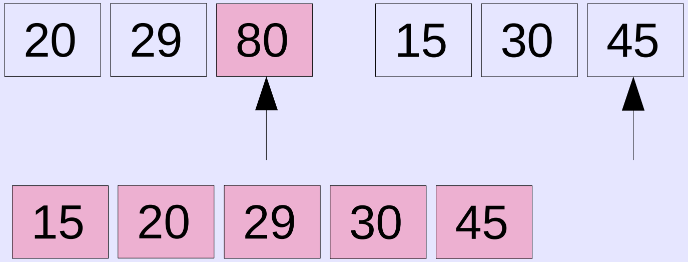
Merge sort

Function merge(left, right). Pre-condition: sublists must be sorted.



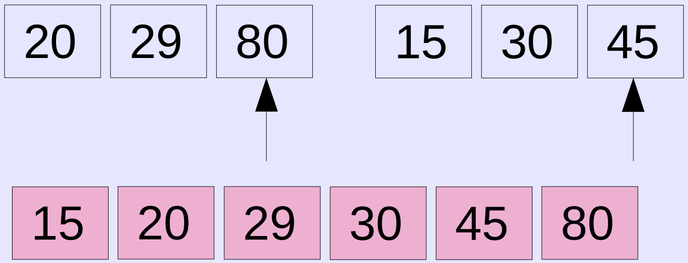
Merge sort

Function merge(left, right). Pre-condition: sublists must be sorted.



Merge sort

Function merge(left, right). Pre-condition: sublists must be sorted.



Merge sort

Function merge(left, right). Pre-condition: sublists must be sorted.

20	29	80	85	90	91

If sublists are such that they are non-overlapping, the algorithm simply concatenates the sublist with the larger numbers after the sublist with the smallest numbers.

20	29	80	85	90	91

Now for a complete example ...

Merge Sort: Divide step

aux

mergeSort(aux, a, 0, 19)

59	46	32	80	46	55	50	43	44	81	12	95	17	80	75	33	40	61	16	87

Merge Sort: Divide step

aux

mergeSort(aux, a, 0, 19)

59	46	32	80	46	55	50	43	44	81	12	95	17	80	75	33	40	61	16	87

a

mergeSort(a, aux, 0, 9)

mergeSort(a, aux, 10, 19)

59	46	32	80	46	55	50	43	44	81	12	95	17	80	75	33	40	61	16	87

Merge Sort: Divide step

aux

mergeSort(aux, a, 0, 19)

59	46	32	80	46	55	50	43	44	81	12	95	17	80	75	33	40	61	16	87

a

mergeSort(a, aux, 0, 9)

mergeSort(a, aux, 10, 19)

59	46	32	80	46	55	50	43	44	81	12	95	17	80	75	33	40	61	16	87

aux

mergeSort(aux, a, 0, 4)

mergeSort(aux, a, 5, 9)

mergeSort(aux, a, 10, 14)

mergeSort(aux, a, 15, 19)

59	46	32	80	46	55	50	43	44	81	12	95	17	80	75	33	40	61	16	87

Merge Sort: Divide step

aux

mergeSort(aux, a, 0, 19)

59	46	32	80	46	55	50	43	44	81	12	95	17	80	75	33	40	61	16	87

a

mergeSort(a, aux, 0, 9) mergeSort(a, aux, 10, 19)

59	46	32	80	46	55	50	43	44	81	12	95	17	80	75	33	40	61	16	87

aux

mergeSort(aux, a, 0, 4) mergeSort(aux, a, 5, 9) mergeSort(aux, a, 10, 14) mergeSort(aux, a, 15, 19)

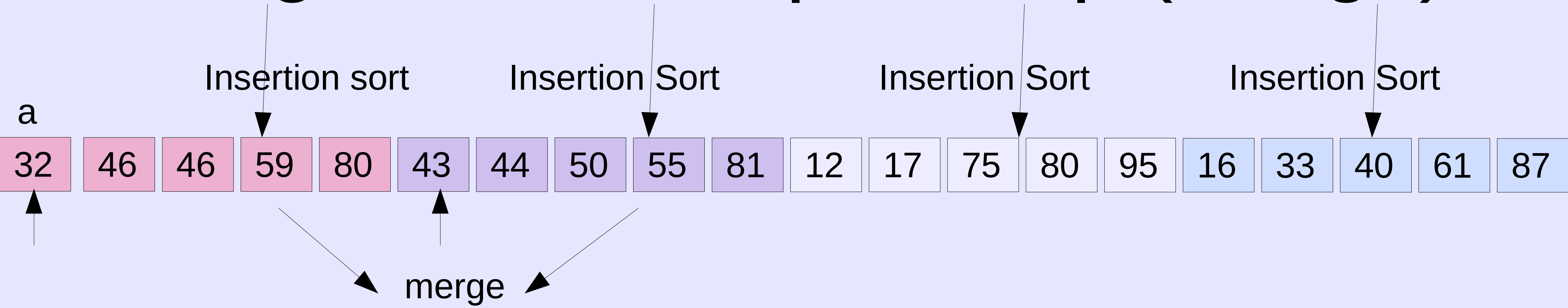
59	46	32	80	46	55	50	43	44	81	12	95	17	80	75	33	40	61	16	87

Insertion sort Insertion Sort Insertion Sort Insertion Sort

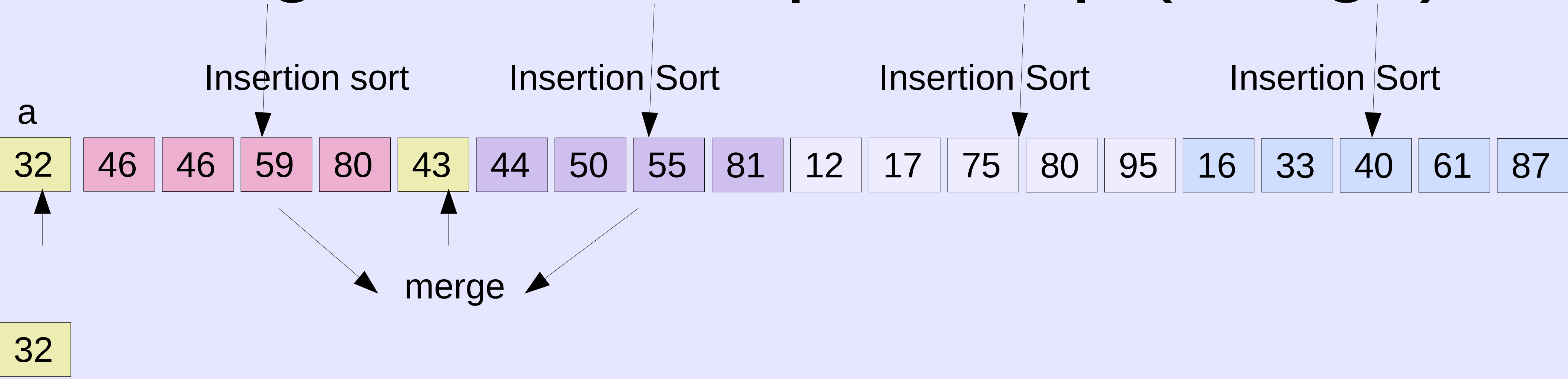
a

32	46	46	59	80	43	44	50	55	81	12	17	75	80	95	16	33	40	61	87

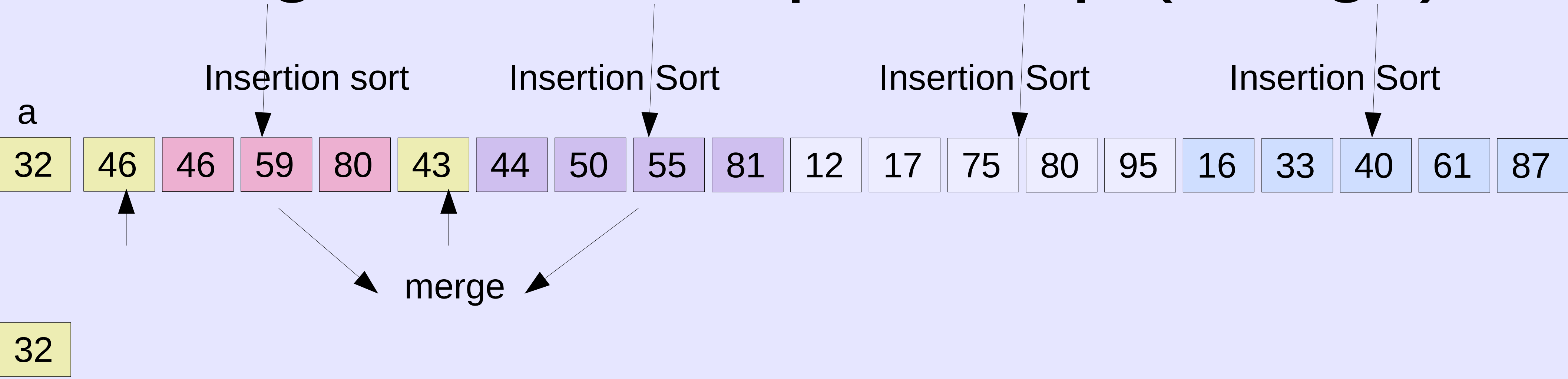
Merge Sort: Conquer step (merge)



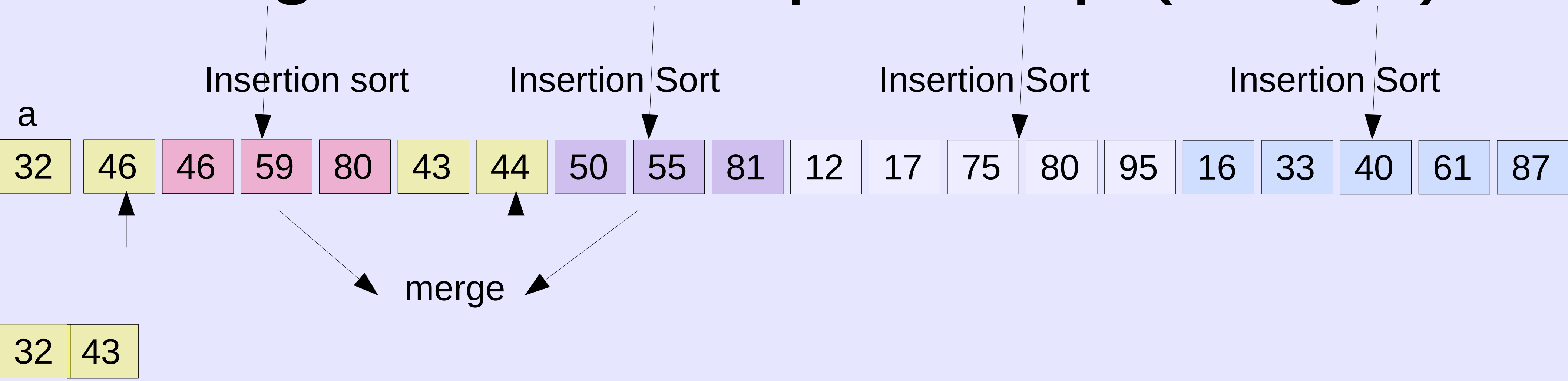
Merge Sort: Conquer step (merge)



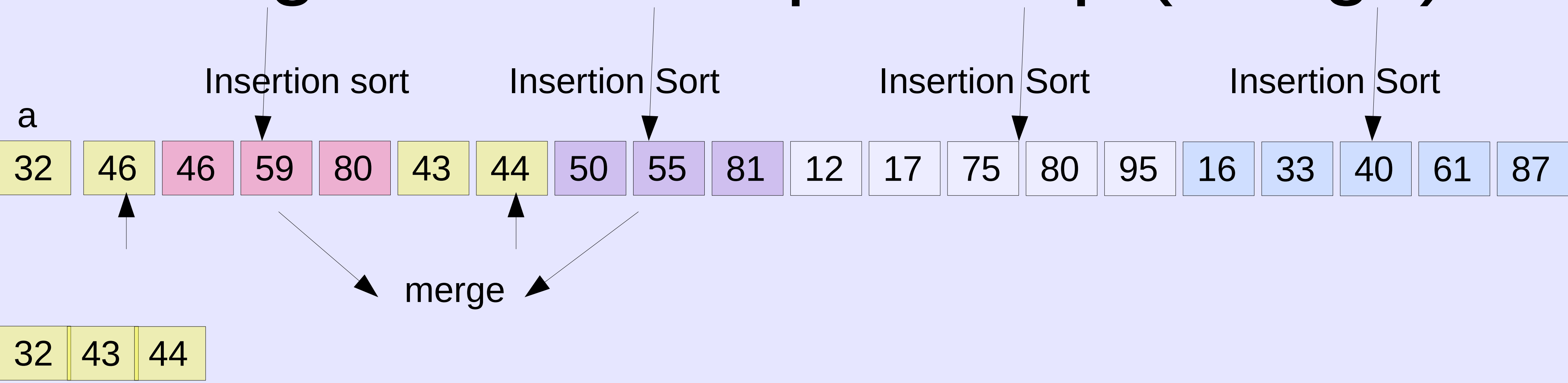
Merge Sort: Conquer step (merge)



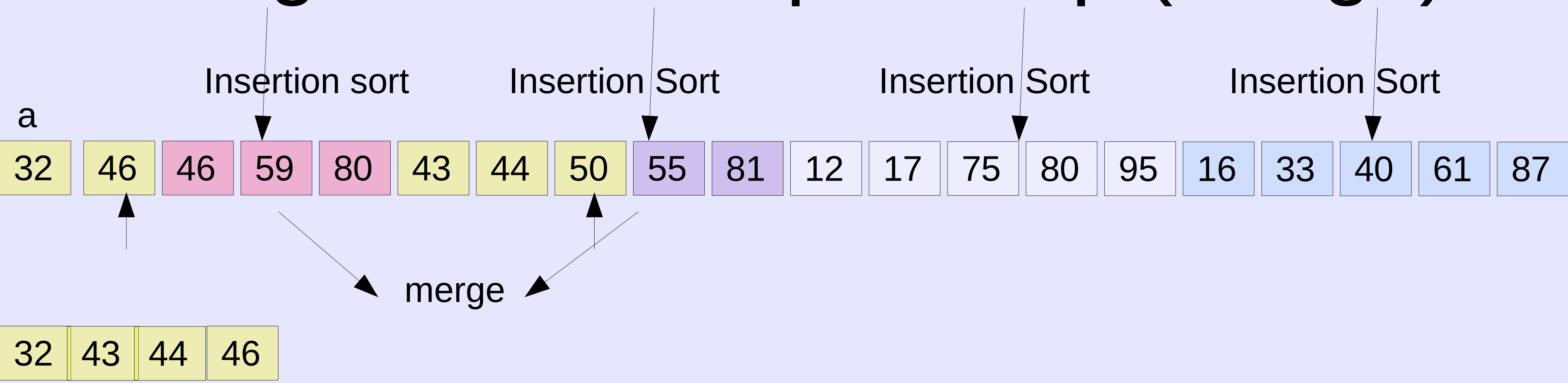
Merge Sort: Conquer step (merge)



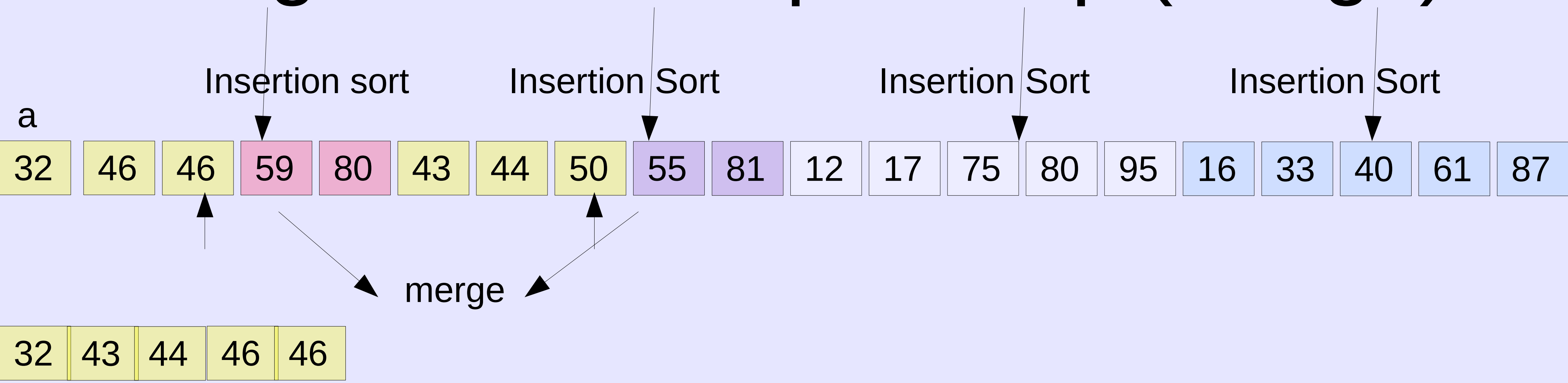
Merge Sort: Conquer step (merge)



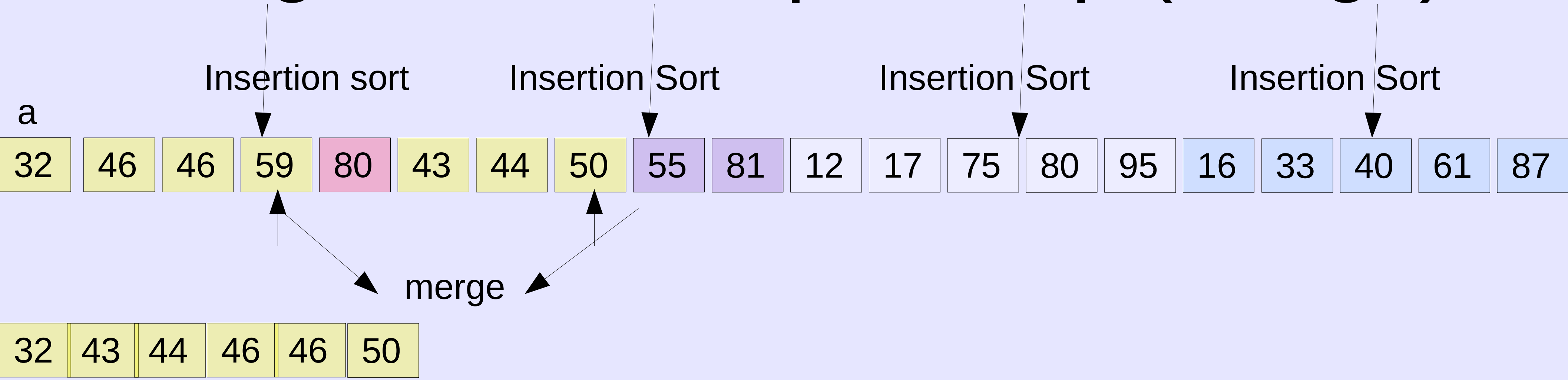
Merge Sort: Conquer step (merge)



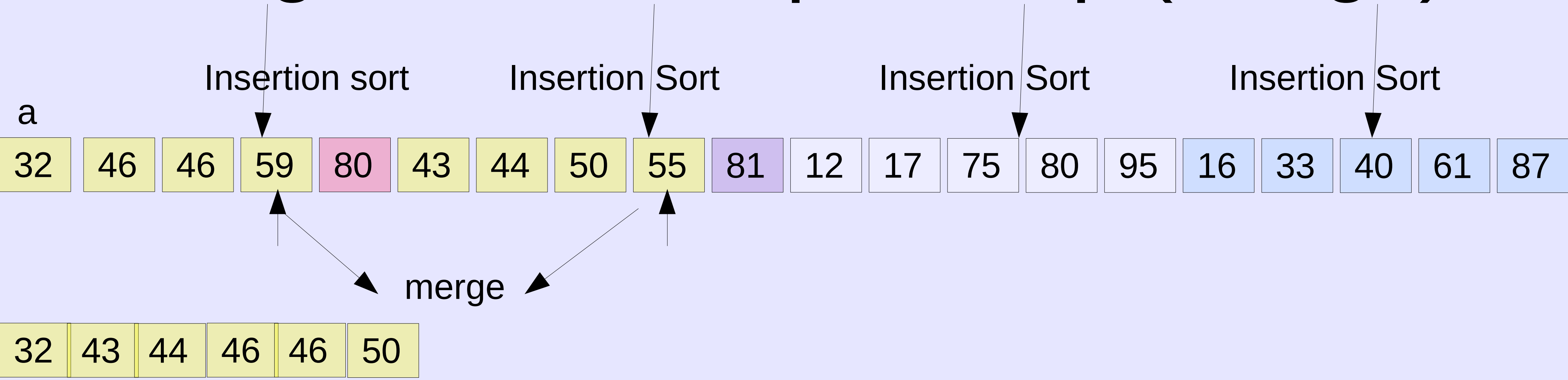
Merge Sort: Conquer step (merge)



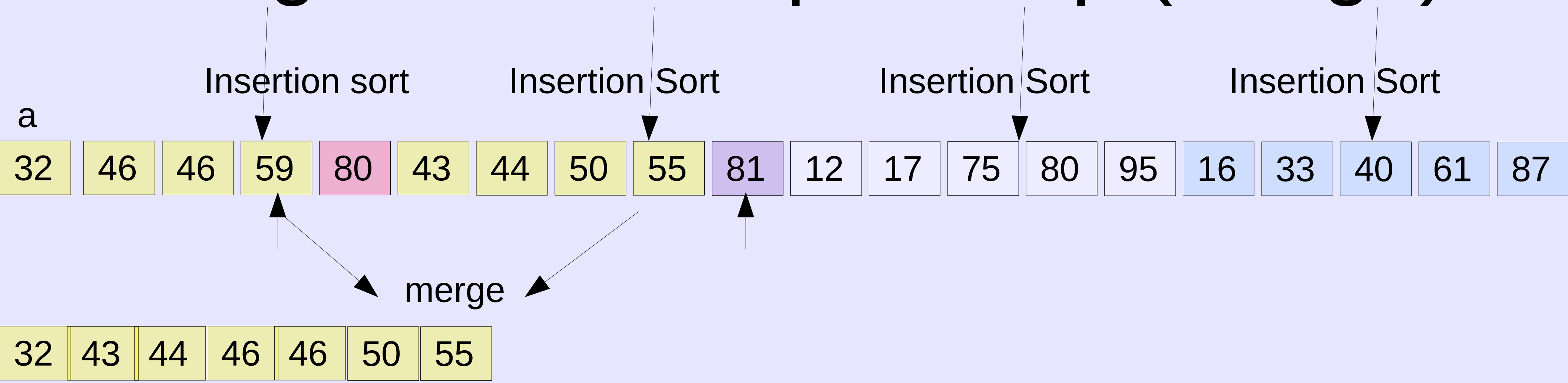
Merge Sort: Conquer step (merge)



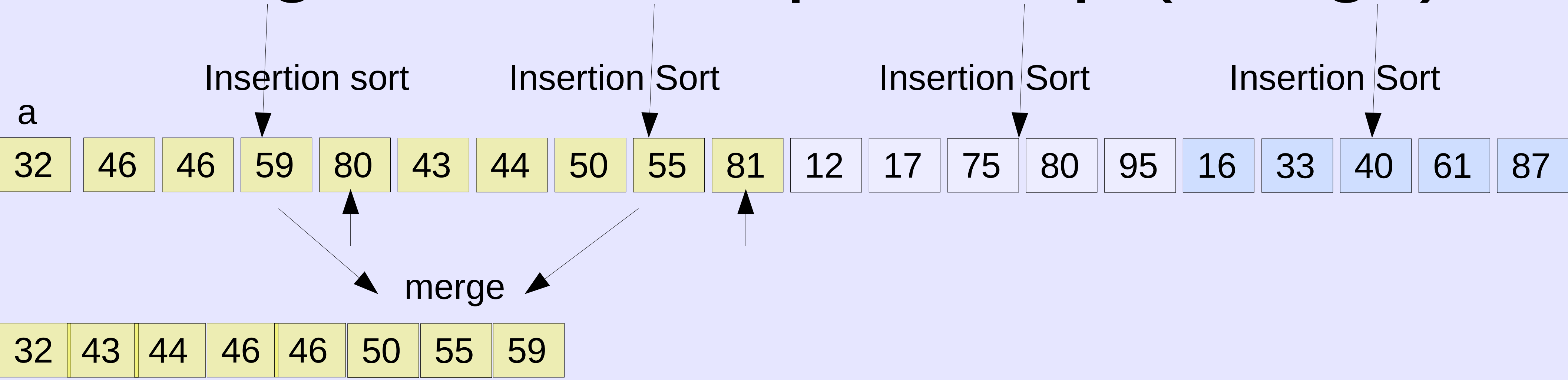
Merge Sort: Conquer step (merge)



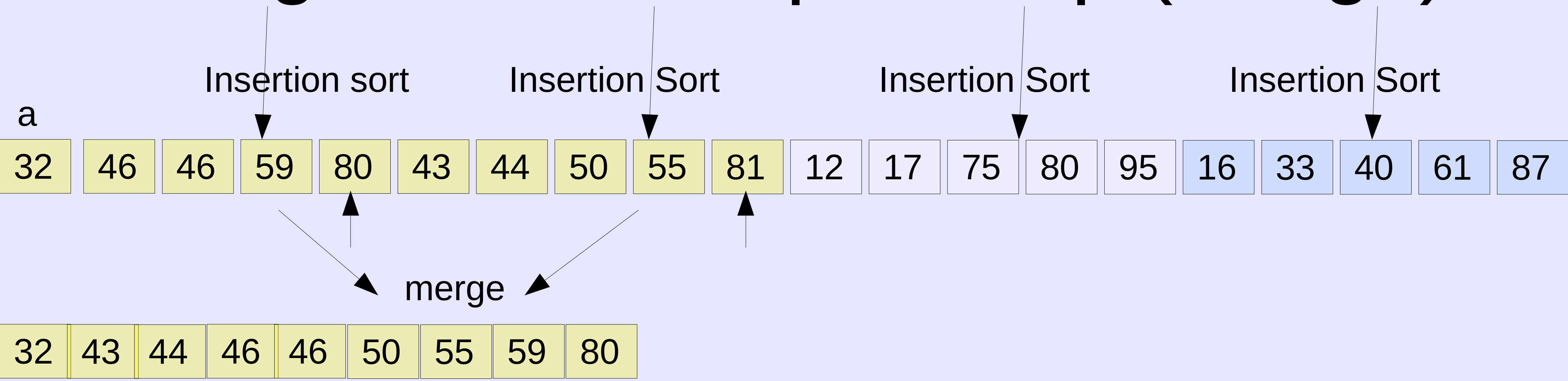
Merge Sort: Conquer step (merge)



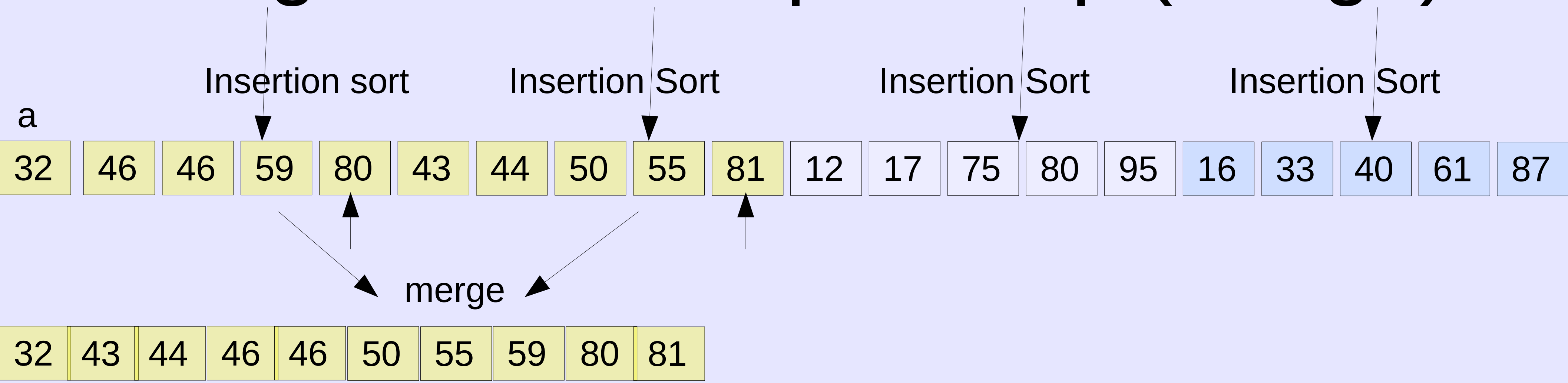
Merge Sort: Conquer step (merge)



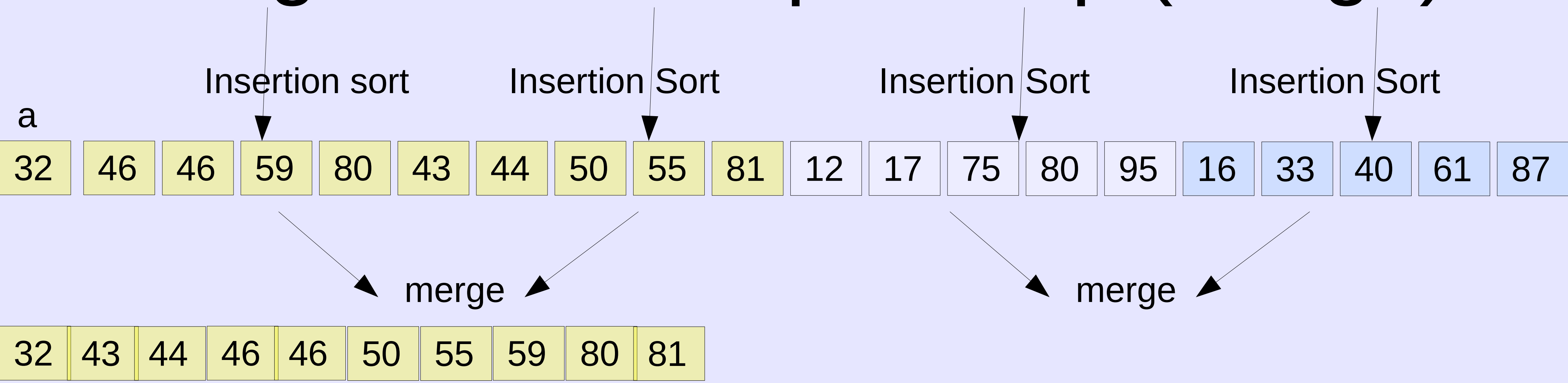
Merge Sort: Conquer step (merge)



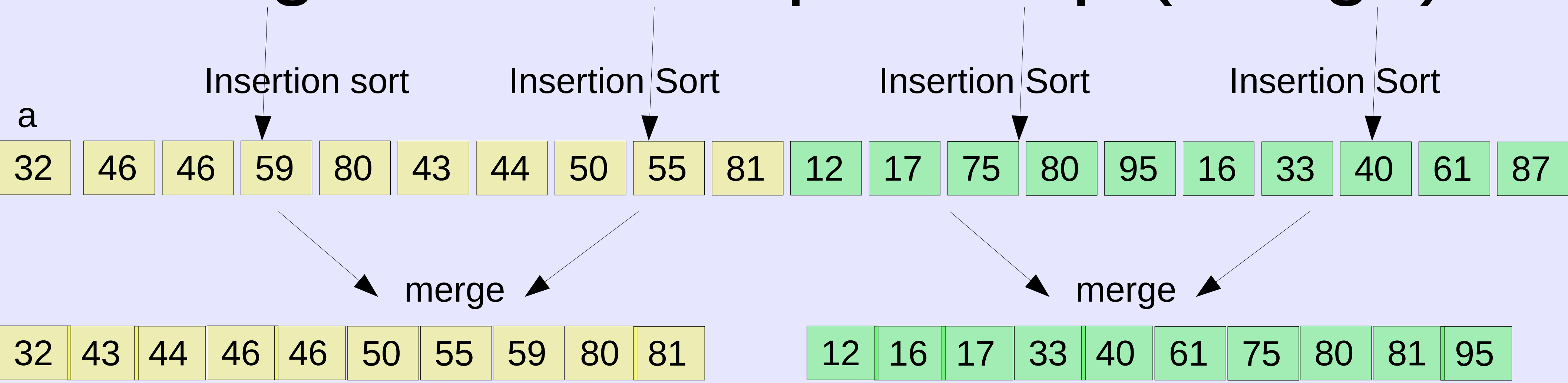
Merge Sort: Conquer step (merge)



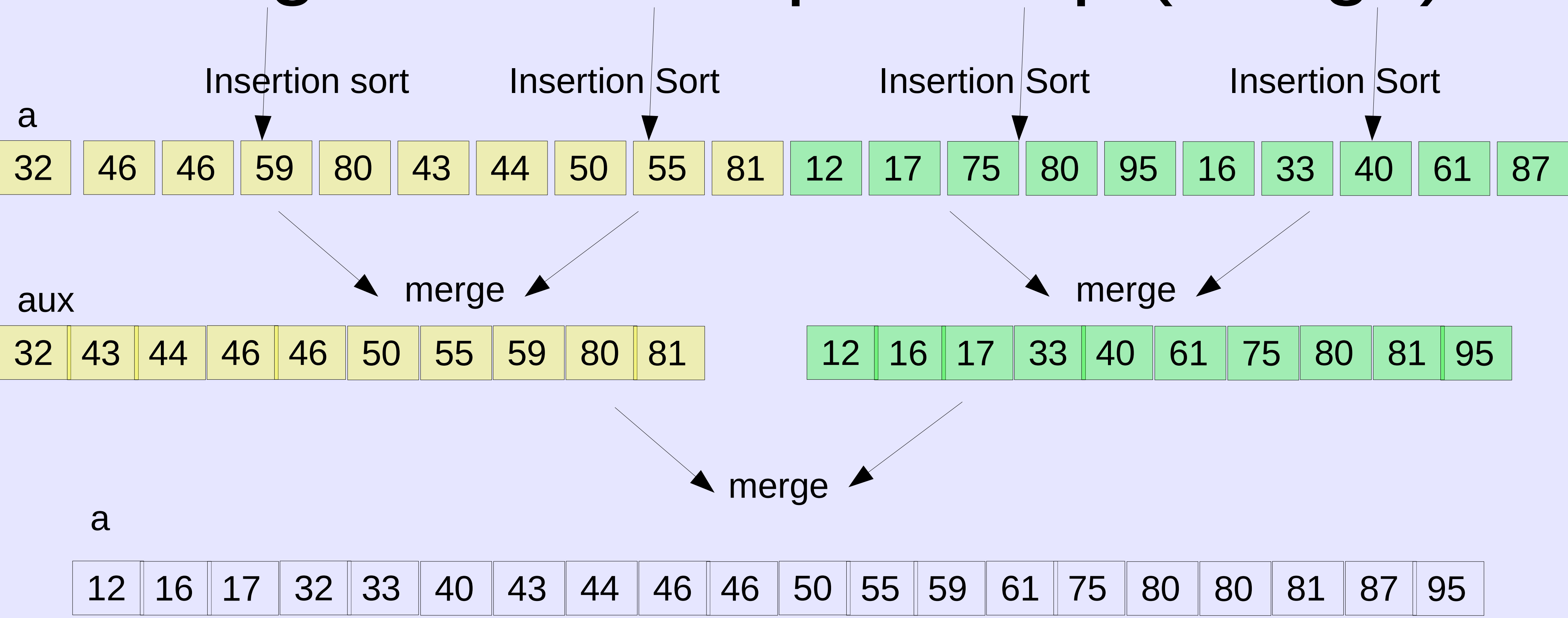
Merge Sort: Conquer step (merge)



Merge Sort: Conquer step (merge)



Merge Sort: Conquer step (merge)



Merge Sort: Complexity Analysis

Time taken to execute Mergesort on input of size N =
Time taken on the left half + Time taken on the right half + The merge step
T(N/2) T(N/2) N

$T(N) = T(N/2) + T(N/2) + N$
$= 2T(N/2) + N$, for $N > 1$ with $T(1) = 0$.

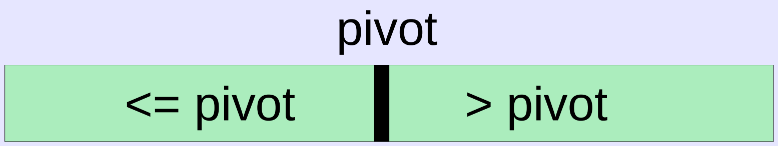
Solution of Mergesort recurrence: $T(N) = N \log_2 N$

See whiteboard for proof by induction on N, when N is a power of 2.

Same recurrence solution holds for many Divide and Conquer algorithms.

Quick Sort

- Developed by C.A.R. Hoare in 1960.
- Honored as one of the top 10 algorithms of 20th century in science and engineering.
- Recursive and elegant algorithm:
  - Step 1: Partition* the array such that
    - element[i] is in its final place for some i
    - All elements <= element[i] to the left of i
    - All elements > element[i] to the right of i
  - Step 2:
    - Perform Step 1 on left sub-array
    - Perform Step 1 on right sub-array



Partitioning is the key to Quicksort

* As with Mergesort, partitioning can stop when number of elements <= 7. Insertion sort used to sort such subarrays.

Quick Sort: In place partitioning

```
// left is the index of the leftmost element of the subarray
// right is the index of the rightmost element of the subarray (inclusive)
// number of elements in subarray = right-left+1
function partition(array, left, right, pivotIndex)
   pivotValue := array[pivotIndex]
   swap array[pivotIndex] and array[right]  // Move pivot to end
   storeIndex := left
   for i from left to right - 1  // left ≤ i < right
       if array[i] ≤ pivotValue
           swap array[i] and array[storeIndex]
           storeIndex := storeIndex + 1  // only increment storeIndex on a swap
   swap array[storeIndex] and array[right]  // Move pivot to its final place
   return storeIndex
```

Source: http://en.wikipedia.org/Quicksort

Quick Sort: In place partitioning

0	1	2	3	4	5	6
56	20	89	12	90	33	16

```
function partition(array, left, right, pivotIndex)
   pivotValue := array[pivotIndex]
   swap array[pivotIndex] and array[right]  // Move pivot to end
   storeIndex := left
   for i from left to right - 1  // left ≤ i < right
       if array[i] ≤ pivotValue
           swap array[i] and array[storeIndex]
           storeIndex := storeIndex + 1  // only increment storeIndex on a swap
   swap array[storeIndex] and array[right]  // Move pivot to its final place
   return storeIndex
```

Quick Sort: In place partitioning

0	1	2	3	4	5	6
56	20	89	12	90	33	16

```
function partition(array, left, right, pivotIndex)
   pivotValue := array[pivotIndex]
   swap array[pivotIndex] and array[right]  // Move pivot to end
   storeIndex := left
   for i from left to right - 1  // left ≤ i < right
      if array[i] ≤ pivotValue
         swap array[i] and array[storeIndex]
         storeIndex := storeIndex + 1  // only increment storeIndex on a swap
   swap array[storeIndex] and array[right]  // Move pivot to its final place
   return storeIndex
```

Invoke: partition(x, 0, 6, 5)

Quick Sort: In place partitioning

0	1	2	3	4	5	6
56	20	89	12	90	33	16

```
function partition(array, left, right, pivotIndex)
  pivotValue := array[pivotIndex]
  swap array[pivotIndex] and array[right]  // Move pivot to end
  storeIndex := left
  for i from left to right - 1  // left ≤ i < right
     if array[i] ≤ pivotValue
         swap array[i] and array[storeIndex]
         storeIndex := storeIndex + 1  // only increment storeIndex on a swap
  swap array[storeIndex] and array[right]  // Move pivot to its final place
  return storeIndex
```

Invoke: partition(x, 0, 6, 5)

pivotValue = 33

Quick Sort: In place partitioning

0	1	2	3	4	5	6
56	20	89	12	90	16	33

```
function partition(array, left, right, pivotIndex)
  pivotValue := array[pivotIndex]
  swap array[pivotIndex] and array[right]  // Move pivot to end
  storeIndex := left
  for i from left to right - 1  // left ≤ i < right
     if array[i] ≤ pivotValue
        swap array[i] and array[storeIndex]
        storeIndex := storeIndex + 1  // only increment storeIndex on a swap
  swap array[storeIndex] and array[right]  // Move pivot to its final place
  return storeIndex
```

Invoke: partition(x, 0, 6, 5)

pivotValue = 33

Quick Sort: In place partitioning

0	1	2	3	4	5	6
56	20	89	12	90	16	33

```
function partition(array, left, right, pivotIndex)
   pivotValue := array[pivotIndex]
   swap array[pivotIndex] and array[right]  // Move pivot to end
   storeIndex := left
   for i from left to right - 1  // left ≤ i < right
       if array[i] ≤ pivotValue
           swap array[i] and array[storeIndex]
           storeIndex := storeIndex + 1  // only increment storeIndex on a swap
   swap array[storeIndex] and array[right]  // Move pivot to its final place
   return storeIndex
```

Invoke: partition(x, 0, 6, 5)

pivotValue = 33

storeIndex = 0

Quick Sort: In place partitioning

0	1	2	3	4	5	6
56	20	89	12	90	16	33

i

```
function partition(array, left, right, pivotIndex)
   pivotValue := array[pivotIndex]
   swap array[pivotIndex] and array[right]  // Move pivot to end
   storeIndex := left
   for i from left to right - 1  // left ≤ i < right
       if array[i] ≤ pivotValue
           swap array[i] and array[storeIndex]
           storeIndex := storeIndex + 1  // only increment storeIndex on a swap
   swap array[storeIndex] and array[right]  // Move pivot to its final place
   return storeIndex
```

Invoke: partition(x, 0, 6, 5)

pivotValue = 33

storeIndex = 0

Quick Sort: In place partitioning

0	1	2	3	4	5	6
56	20	89	12	90	16	33

i

```
function partition(array, left, right, pivotIndex)
  pivotValue := array[pivotIndex]
  swap array[pivotIndex] and array[right]  // Move pivot to end
  storeIndex := left
  for i from left to right - 1  // left ≤ i < right
     if array[i] ≤ pivotValue
        swap array[i] and array[storeIndex]
        storeIndex := storeIndex + 1  // only increment storeIndex on a swap
  swap array[storeIndex] and array[right]  // Move pivot to its final place
  return storeIndex
```

Invoke: partition(x, 0, 6, 5)

pivotValue = 33

storeIndex = 0

Quick Sort: In place partitioning

0	1	2	3	4	5	6
56	20	89	12	90	16	33

i

```
function partition(array, left, right, pivotIndex)
   pivotValue := array[pivotIndex]
   swap array[pivotIndex] and array[right]  // Move pivot to end
   storeIndex := left
   for i from left to right - 1  // left ≤ i < right
       if array[i] ≤ pivotValue
           swap array[i] and array[storeIndex]
           storeIndex := storeIndex + 1  // only increment storeIndex on a swap
   swap array[storeIndex] and array[right]  // Move pivot to its final place
   return storeIndex
```

Invoke: partition(x, 0, 6, 5)

pivotValue = 33

storeIndex = 0

Quick Sort: In place partitioning

0	1	2	3	4	5	6
20	56	89	12	90	16	33

i

```
function partition(array, left, right, pivotIndex)
    pivotValue := array[pivotIndex]
    swap array[pivotIndex] and array[right]  // Move pivot to end
    storeIndex := left
    for i from left to right - 1  // left ≤ i < right
        if array[i] ≤ pivotValue
            swap array[i] and array[storeIndex]
            storeIndex := storeIndex + 1  // only increment storeIndex on a swap
    swap array[storeIndex] and array[right]  // Move pivot to its final place
    return storeIndex
```

Invoke: partition(x, 0, 6, 5)

pivotValue = 33

storeIndex = 0

Quick Sort: In place partitioning

0	1	2	3	4	5	6
20	56	89	12	90	16	33
	i					

```
function partition(array, left, right, pivotIndex)
   pivotValue := array[pivotIndex]
   swap array[pivotIndex] and array[right]  // Move pivot to end
   storeIndex := left
   for i from left to right - 1  // left ≤ i < right
       if array[i] ≤ pivotValue
           swap array[i] and array[storeIndex]
           storeIndex := storeIndex + 1  // only increment storeIndex on a swap
   swap array[storeIndex] and array[right]  // Move pivot to its final place
   return storeIndex
```

Invoke: partition(x, 0, 6, 5)

pivotValue = 33

storeIndex = 1

Quick Sort: In place partitioning

0	1	2	3	4	5	6
20	56	89	12	90	16	33

i

```
function partition(array, left, right, pivotIndex)
   pivotValue := array[pivotIndex]
   swap array[pivotIndex] and array[right]  // Move pivot to end
   storeIndex := left
   for i from left to right - 1  // left ≤ i < right
       if array[i] ≤ pivotValue
           swap array[i] and array[storeIndex]
           storeIndex := storeIndex + 1  // only increment storeIndex on a swap
   swap array[storeIndex] and array[right]  // Move pivot to its final place
   return storeIndex
```

Invoke: partition(x, 0, 6, 5)

pivotValue = 33

storeIndex = 1

Quick Sort: In place partitioning

0	1	2	3	4	5	6
20	56	89	12	90	16	33

i

```
function partition(array, left, right, pivotIndex)
   pivotValue := array[pivotIndex]
   swap array[pivotIndex] and array[right]  // Move pivot to end
   storeIndex := left
   for i from left to right - 1  // left ≤ i < right
       if array[i] ≤ pivotValue
           swap array[i] and array[storeIndex]
           storeIndex := storeIndex + 1  // only increment storeIndex on a swap
   swap array[storeIndex] and array[right]  // Move pivot to its final place
   return storeIndex
```

Invoke: partition(x, 0, 6, 5)

pivotValue = 33

storeIndex = 1

Quick Sort: In place partitioning

0	1	2	3	4	5	6
20	12	89	56	90	16	33
			↑ i			

```
function partition(array, left, right, pivotIndex)
   pivotValue := array[pivotIndex]
   swap array[pivotIndex] and array[right]  // Move pivot to end
   storeIndex := left
   for i from left to right - 1  // left ≤ i < right
       if array[i] ≤ pivotValue
           swap array[i] and array[storeIndex]
           storeIndex := storeIndex + 1  // only increment storeIndex on a swap
   swap array[storeIndex] and array[right]  // Move pivot to its final place
   return storeIndex
```

Invoke: partition(x, 0, 6, 5)

pivotValue = 33

storeIndex = 1

Quick Sort: In place partitioning

0	1	2	3	4	5	6
20	12	89	56	90	16	33
			i			

```
function partition(array, left, right, pivotIndex)
   pivotValue := array[pivotIndex]
   swap array[pivotIndex] and array[right]  // Move pivot to end
   storeIndex := left
   for i from left to right - 1  // left ≤ i < right
       if array[i] ≤ pivotValue
           swap array[i] and array[storeIndex]
           storeIndex := storeIndex + 1  // only increment storeIndex on a swap
   swap array[storeIndex] and array[right]  // Move pivot to its final place
   return storeIndex
```

Invoke: partition(x, 0, 6, 5)

pivotValue = 33

storeIndex = 2

Quick Sort: In place partitioning

0	1	2	3	4	5	6
20	12	89	56	90	16	33

i

```
function partition(array, left, right, pivotIndex)
   pivotValue := array[pivotIndex]
   swap array[pivotIndex] and array[right]  // Move pivot to end
   storeIndex := left
   for i from left to right - 1  // left ≤ i < right
       if array[i] ≤ pivotValue
           swap array[i] and array[storeIndex]
           storeIndex := storeIndex + 1  // only increment storeIndex on a swap
   swap array[storeIndex] and array[right]  // Move pivot to its final place
   return storeIndex
```

Invoke: partition(x, 0, 6, 5)

pivotValue = 33

storeIndex = 2

Quick Sort: In place partitioning

0	1	2	3	4	5	6
20	12	89	56	90	16	33

i

```
function partition(array, left, right, pivotIndex)
   pivotValue := array[pivotIndex]
   swap array[pivotIndex] and array[right]  // Move pivot to end
   storeIndex := left
   for i from left to right - 1  // left ≤ i < right
       if array[i] ≤ pivotValue
           swap array[i] and array[storeIndex]
           storeIndex := storeIndex + 1  // only increment storeIndex on a swap
   swap array[storeIndex] and array[right]  // Move pivot to its final place
   return storeIndex
```

Invoke: partition(x, 0, 6, 5)

pivotValue = 33

storeIndex = 2

Quick Sort: In place partitioning

0	1	2	3	4	5	6
20	12	16	56	90	89	33

i

```
function partition(array, left, right, pivotIndex)
   pivotValue := array[pivotIndex]
   swap array[pivotIndex] and array[right]  // Move pivot to end
   storeIndex := left
   for i from left to right - 1  // left ≤ i < right
      if array[i] ≤ pivotValue
         swap array[i] and array[storeIndex]
         storeIndex := storeIndex + 1  // only increment storeIndex on a swap
   swap array[storeIndex] and array[right]  // Move pivot to its final place
   return storeIndex
```

Invoke: partition(x, 0, 6, 5)

pivotValue = 33

storeIndex = 2

Quick Sort: In place partitioning

0	1	2	3	4	5	6
20	12	16	56	90	89	33
					i	

```
function partition(array, left, right, pivotIndex)
   pivotValue := array[pivotIndex]
   swap array[pivotIndex] and array[right]  // Move pivot to end
   storeIndex := left
   for i from left to right - 1  // left ≤ i < right
       if array[i] ≤ pivotValue
           swap array[i] and array[storeIndex]
           storeIndex := storeIndex + 1  // only increment storeIndex on a swap
   swap array[storeIndex] and array[right]  // Move pivot to its final place
   return storeIndex
```

Invoke: partition(x, 0, 6, 5)

pivotValue = 33

storeIndex = 3

Quick Sort: In place partitioning

0	1	2	3	4	5	6
20	12	16	33	90	89	56

```
function partition(array, left, right, pivotIndex)
   pivotValue := array[pivotIndex]
   swap array[pivotIndex] and array[right]  // Move pivot to end
   storeIndex := left
   for i from left to right - 1  // left ≤ i < right
       if array[i] ≤ pivotValue
           swap array[i] and array[storeIndex]
           storeIndex := storeIndex + 1  // only increment storeIndex on a swap
   swap array[storeIndex] and array[right]  // Move pivot to its final place
   return storeIndex
```

Invoke: partition(x, 0, 6, 5)

pivotValue = 33

storeIndex = 3

Quick Sort: In place partitioning

0	1	2	3	4	5	6
20	12	16	33	90	89	56

```
function partition(array, left, right, pivotIndex)
   pivotValue := array[pivotIndex]
   swap array[pivotIndex] and array[right]  // Move pivot to end
   storeIndex := left
   for i from left to right - 1  // left ≤ i < right
       if array[i] ≤ pivotValue
           swap array[i] and array[storeIndex]
           storeIndex := storeIndex + 1  // only increment storeIndex on a swap
   swap array[storeIndex] and array[right]  // Move pivot to its final place
   return storeIndex
```

Quicksort: An example

quicksort(A, 1, 12)
12 9 80 77 8 92 45 <u>19</u> 78 15 3 43
12 8 9 3 15 <u>19</u> 80 92 43 77 45 78

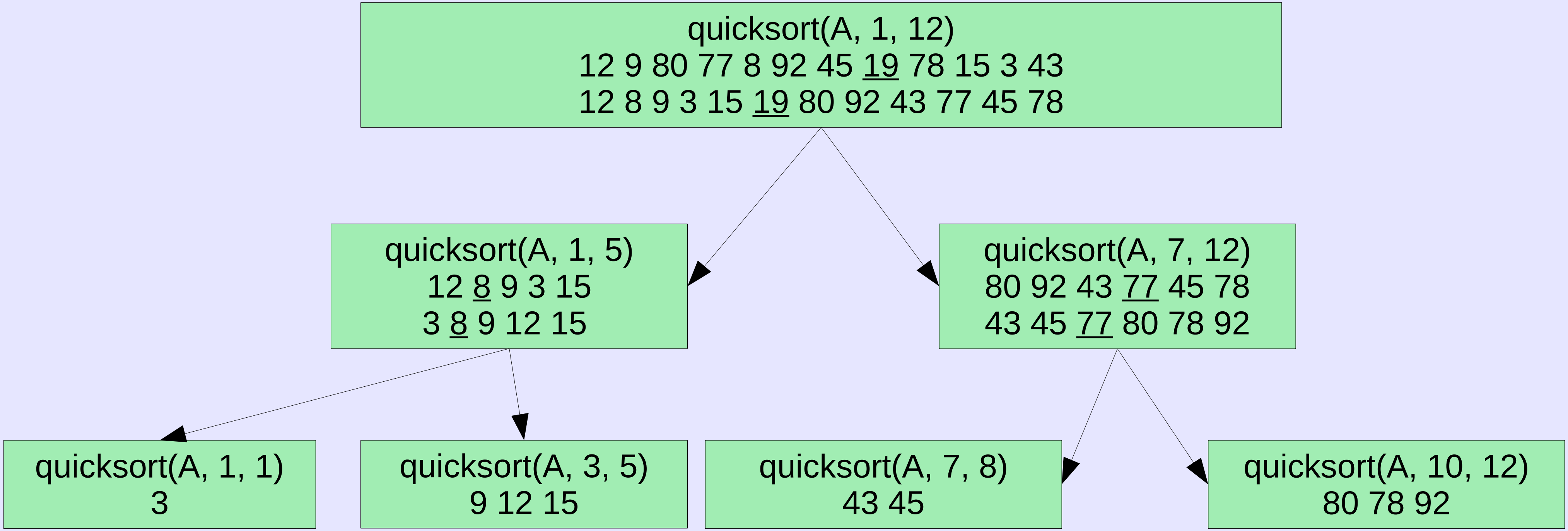
Quicksort: An example

quicksort(A, 1, 12)
12 9 80 77 8 92 45 19 78 15 3 43
12 8 9 3 15 19 80 92 43 77 45 78

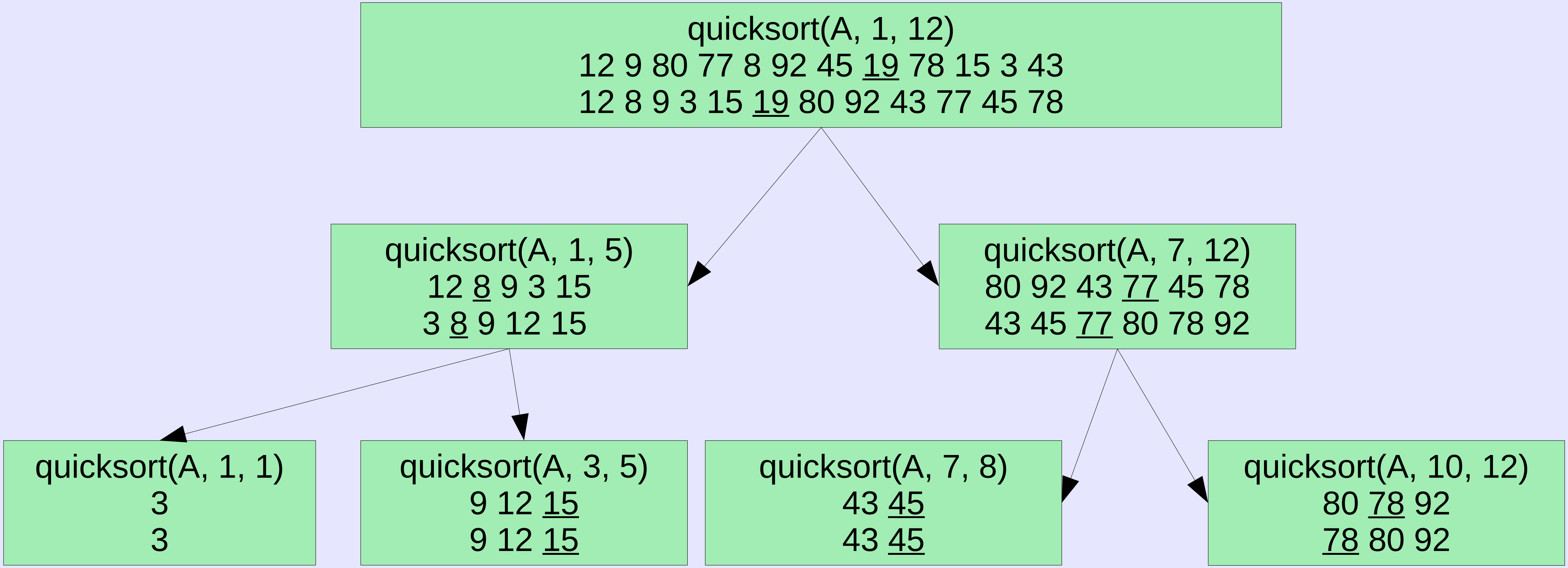
quicksort(A, 1, 5)
12 8 9 3 15

quicksort(A, 7, 12)
80 92 43 77 45 78

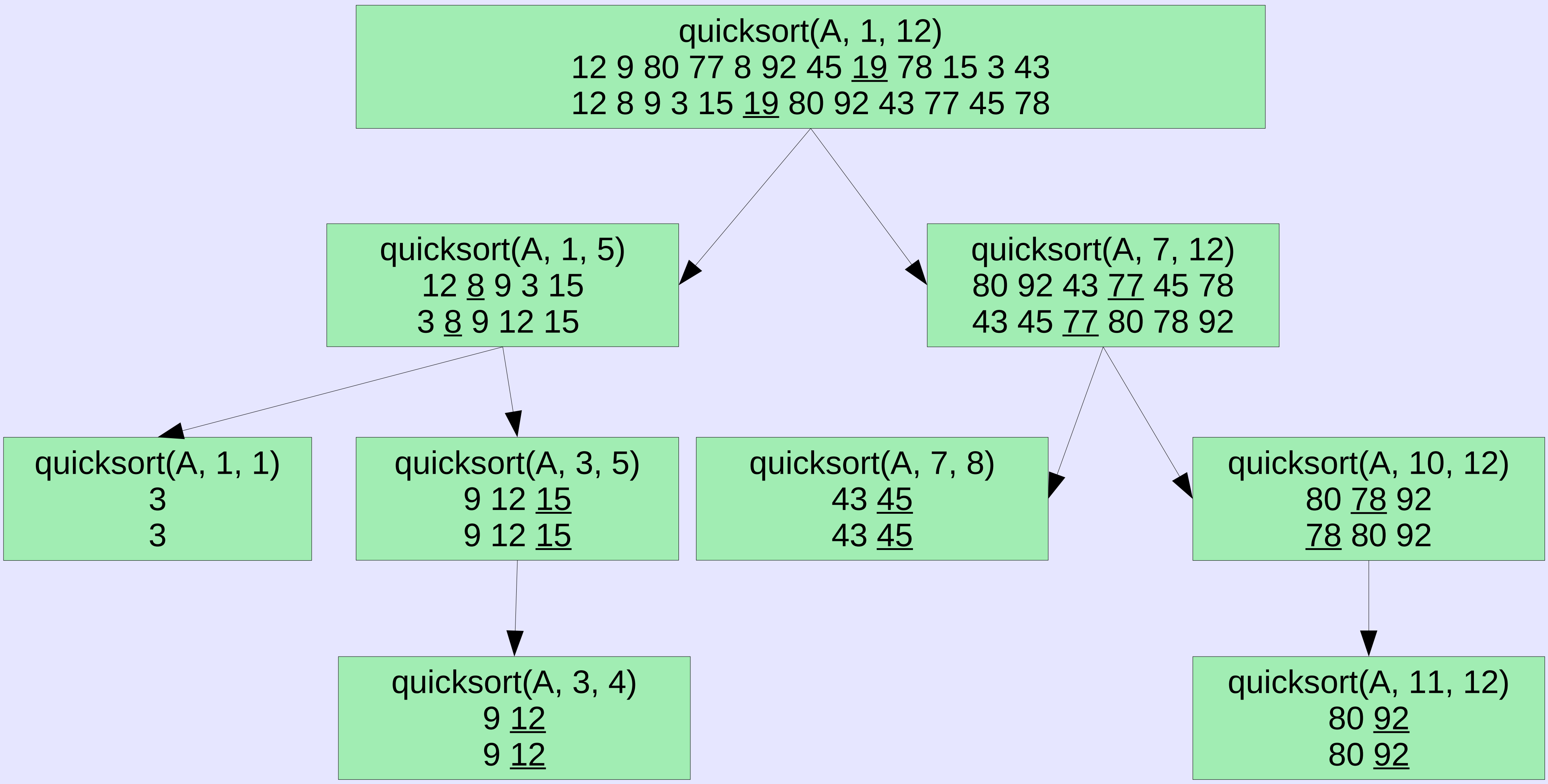
Quicksort: An example



Quicksort: An example

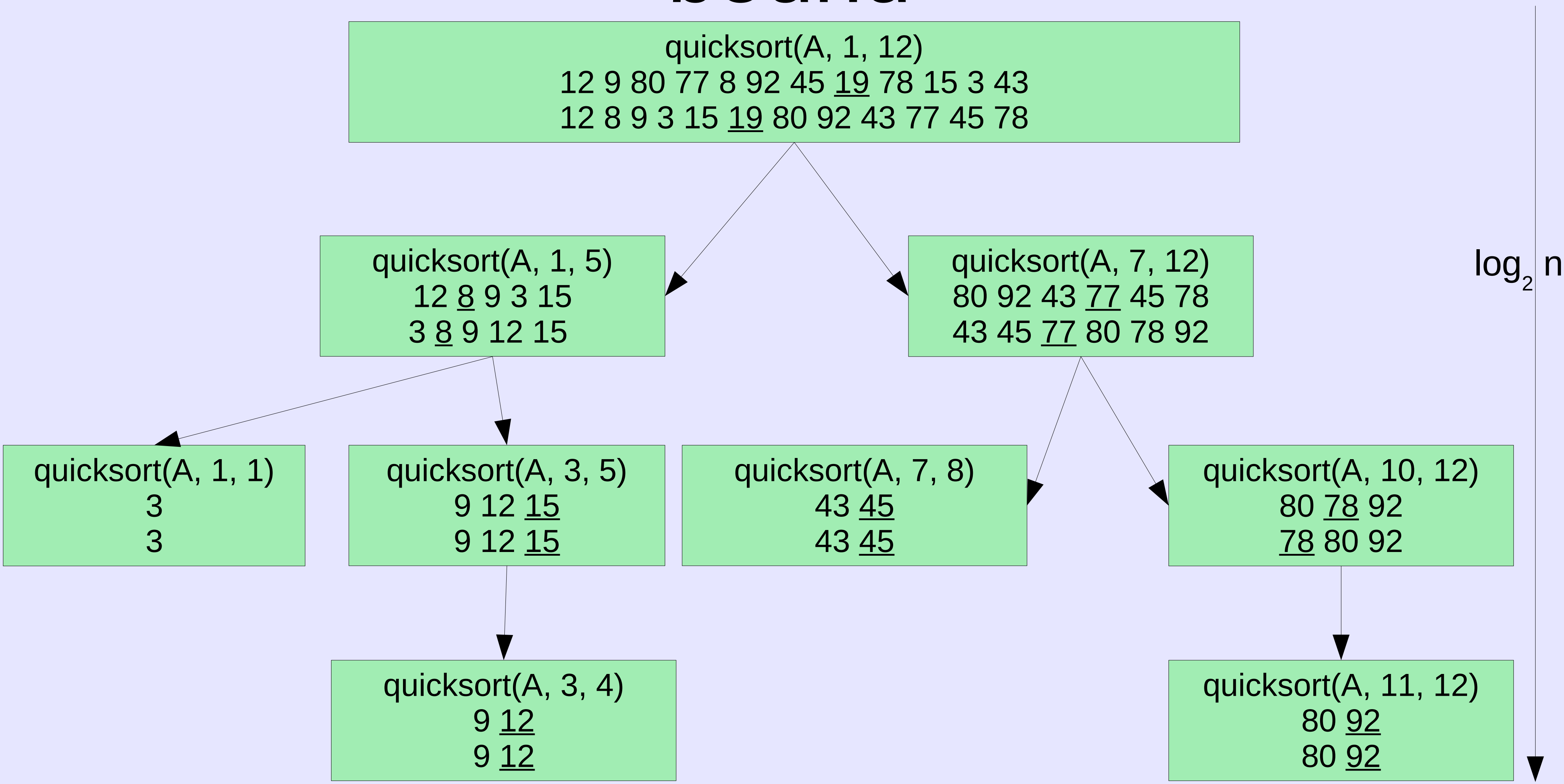


Quicksort: An example



3 8 9 12 15 19 43 45 77 78 80 92

Quicksort: An intuitive complexity bound



3	8	9	12	15	19	43	45	77	78	80	92

n comparisons

$\boxed{O(n \log n)}$

Partition and choice of pivot

- Clearly, partitioning is important to Quicksort.
- Also clearly, choosing the right pivot to partition on is important.
  - Want to be left with two (approximately) equal halves.
    - See CLRS on how even with a 9-to-1 split, runtime remains *O(n log n)*.
  - If this is not the case, then Quicksort degenerates to $O(n^2)$ for worst case.
    - Worst case occurs when one subarray has *n-1* elements and the other subarray has 0.

Partition and choice of pivot

- Therefore, the choice of pivot is the most important attribute in Quicksort.
- So, how do we pick the pivot?

Partition and choice of pivot

- Therefore, the choice of pivot is the most important attribute in Quicksort.
- So, how do we pick the pivot?
- Various ways:
  - First, last, middle element.
  - Random element.
  - <u>Median of three</u>
    - Take three random elements
    - Median of first, last, middle element.

Partition and choice of pivot

pivot = last element

quicksort(A, 1, 7)
1 6 19 18 23 40 <u>63</u>
1 6 19 18 23 40 <u>63</u>

Partition and choice of pivot

quicksort(A, 1, 7)
1 6 19 18 23 40 63
1 6 19 18 23 40 63

pivot = last element

quicksort(A, 1, 6)
1 6 19 18 23 40
1 6 19 18 23 40

Partition and choice of pivot

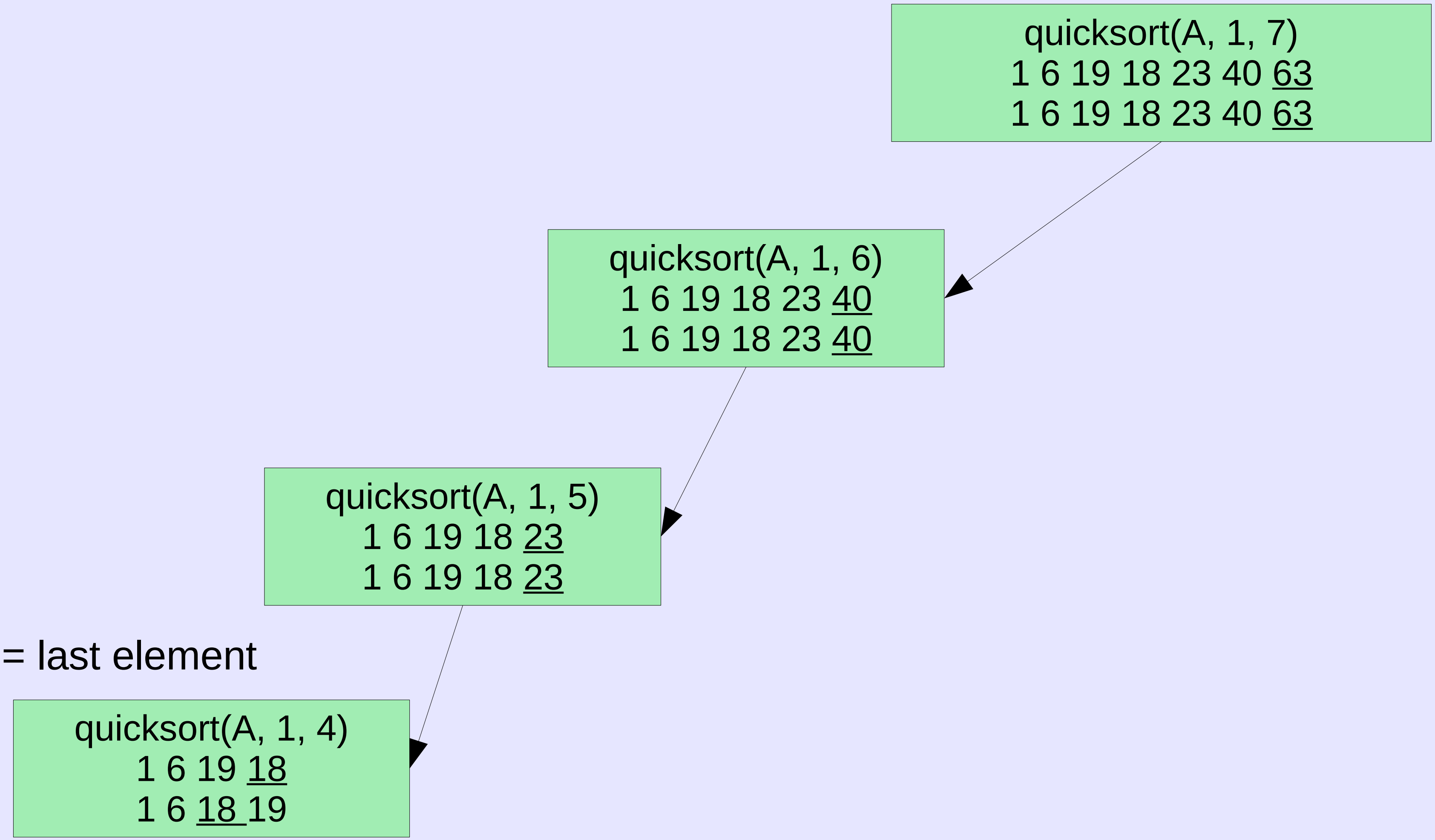
quicksort(A, 1, 7)
1 6 19 18 23 40 63
1 6 19 18 23 40 63

quicksort(A, 1, 6)
1 6 19 18 23 40
1 6 19 18 23 40

pivot = last element

quicksort(A, 1, 5)
1 6 19 18 23
1 6 19 18 23

Partition and choice of pivot



pivot = last element

Partition and choice of pivot

quicksort(A, 1, 7)
1 6 19 18 23 40 63
1 6 19 18 23 40 63

quicksort(A, 1, 6)
1 6 19 18 23 40
1 6 19 18 23 40

quicksort(A, 1, 5)
1 6 19 18 23
1 6 19 18 23

quicksort(A, 1, 4)
1 6 19 18
1 6 18 19

⋮

Question: What does this remind you of?

Partition and choice of pivot

quicksort(A, 1, 7)
1 6 19 18 23 40 63
1 6 19 18 23 40 63

quicksort(A, 1, 6)
1 6 19 18 23 40
1 6 19 18 23 40

quicksort(A, 1, 5)
1 6 19 18 23
1 6 19 18 23

quicksort(A, 1, 4)
1 6 19 18
1 6 18 19

⋮

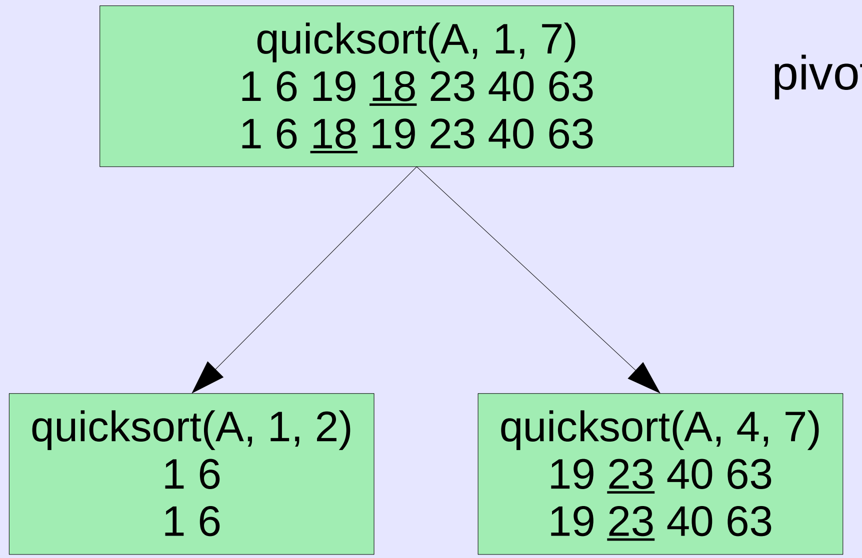
Question: What does this remind you of?
Skewed trees! *O(n)*
Here, it is $O(n^2)$ because of n levels and n comparisons.

Partition and choice of pivot

quicksort(A, 1, 7)
1 6 19 18 23 40 63
1 6 18 19 23 40 63

pivot = median(1, 18, 63) = 18

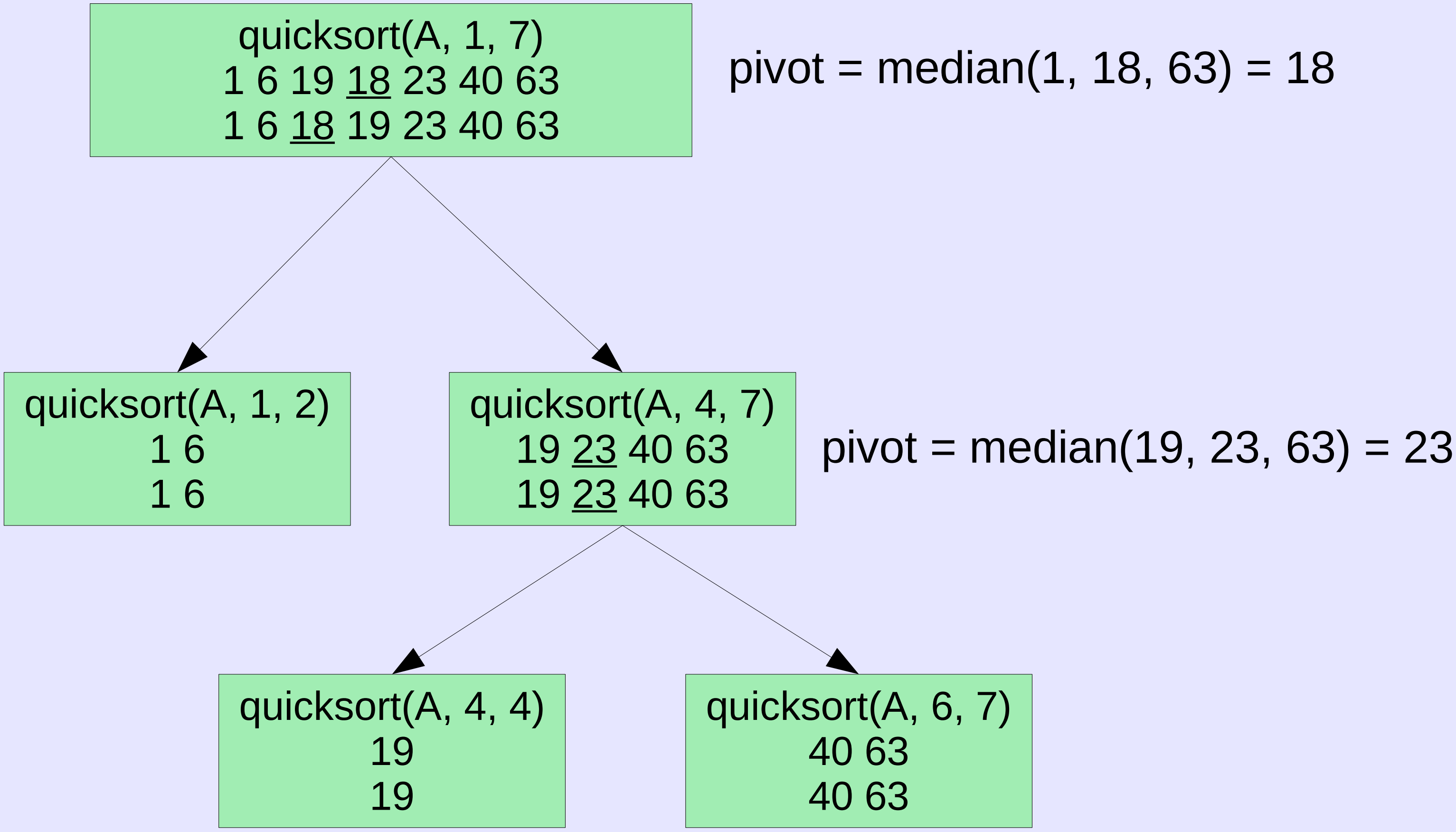
Partition and choice of pivot



pivot = median(1, 18, 63) = 18

pivot = median(19, 23, 63) = 23

Partition and choice of pivot



Divide-and-conquer sort comparison

Name	Average	Worst	Stable	Space
Quick sort	O(n log n)	$O(n^2)$	No	O(log n)[1]
Merge sort	O(n log n)	O(n log n)	Yes	O(n)[2]
Heap sort	O(n log n)	O(n log n)	No	O(1)

1. Using in-place partitioning; O(log n) additional space for recursion.
2. For auxiliary array.

The Master Theorem

- Many recursive algorithms
  - Split the problem into pieces (divide).
  - Recurse on the pieces to solve the problem (conquer).
- Running time of such algorithms is fundamentally a recurrence relation:
  - $T(n) = T(n/2) + a$, where a some time to operate on the subproblem (bounded by n).
  - E.g.: Mergesort $T(n) = T(n/2) + T(n/2) + n$

The Master Theorem

- Theorem: For an increasing function, *f*, with the recurrence relation:

  $f(n) = a\, f(n/b) + cn^d$,

  with $a \geq 1$, $b > 1$, $c > 0$ and $d \geq 0$

$$f(n) = \begin{cases} O(n^d) & : a < b^d \\ O(n^d \log n) & : a = b^d \\ O(n^{\log_b a}) & : a > b^d \end{cases}$$

Proof of the Master Theorem for all three cases is in CLRS.

The Master Theorem

- Cannot use the Master Theorem if
  - $f(n)$ is not monotone, i.e., $f(n) = \sin n$
  - $f(n)$ is not a polynomial, i.e., $f(n) = 2 f(n/2) + 2^n$
  - b cannot be expressed as a constant, i.e., $f(n) = f(n/n^2)$
- Note that the Master Theorem does **not** solve a recurrence relation, **nor** does it derive one for you.
- You need a recurrence relation to be able to derive the complexity using the Master Theorem as a template.

The Master Theorem

$$f(n) = \begin{cases} O(n^d) & : a < b^d \\ O(n^d \log n) & : a = b^d \\ O(n^{\log_b a}) & : a > b^d \end{cases}$$

$$f(n) = a\, f(n/b) + cn^d$$

For Mergesort and Quicksort, we have a recurrence relation of:
$T(N) = 2\,T(N/2) + N$.
Here, $a = 2$, $b = 2$, $c = 1$, $d = 1$,
so $a = b^d$. Case II applies and Mergesort/Quicksort = $O(n \log n)$.

The Master Theorem

$$f(n) = \begin{cases} O(n^d) & : a < b^d \\ O(n^d \log n) & : a = b^d \\ O(n^{\log_b a}) & : a > b^d \end{cases}$$

$$f(n) = a\, f(n/b) + cn^d$$

For BST search, we have a recurrence relation of:
$T(N) = T(N/2) + 1$.

Here, $a = 1, b = 2, c = 1, d = 0$,
so $a = b^d$. Case II applies and binary search is $O(\log n)$.

The Master Theorem

$$f(n) = \begin{cases} O(n^d) & : a < b^d \\ O(n^d \log n) & : a = b^d \\ O(n^{\log_b a}) & : a > b^d \end{cases}$$

$$f(n) = a\, f(n/b) + cn^d$$

For BST traversal, we have a recurrence relation of:
$T(N) = 2T(N/2) + 1$.

Here, $a = 2$, $b = 2$, $c = 1$, $d = 0$,
so $a > b^d$. Case III applies and binary search is $O(n)$.

The Master Theorem

$$f(n) = \begin{cases} O(n^d) & : a < b^d \\ O(n^d \log n) & : a = b^d \\ O(n^{\log_b a}) & : a > b^d \end{cases}$$

$$f(n) = a\,f(n/b) + cn^d$$

Suppose you come up with an algorithm as follows:

```
function f(n)  {
   if (n <= 1) return
   else   {
      a = f(n/2)
      b = f(n/2)
      combine a and b using n^2 steps
   }
}
```

The running time will be: $f(n) = 2 * f(n/2) + n^2$

From the Master Theorem we have a = 2, b = 2 , c = 1, d = 2, so $a < b^d$.

Case I applies and the running time is $O(n^2)$.

Sorting: run time estimates

Running time estimates:

- Home pc executes 10^8 comparisons/second.
- Supercomputer executes 10^{12} comparisons/second.

Insertion Sort (N^2)

computer	thousand	million	billion
home	instant	2.8 hours	317 years
super	instant	1 second	1.6 weeks

Mergesort (N log N)

thousand	million	billion
instant	1 sec	18 min
instant	instant	instant

Quicksort (N log N)

thousand	million	billion
instant	0.3 sec	6 min
instant	instant	instant

Lesson 1. Good algorithms are better than supercomputers.

Lesson 2. Great algorithms are better than good ones.

Source: Robert Sedgewick and Kevin Wayne, 2007