Sort

DSA

Why we do sorting?

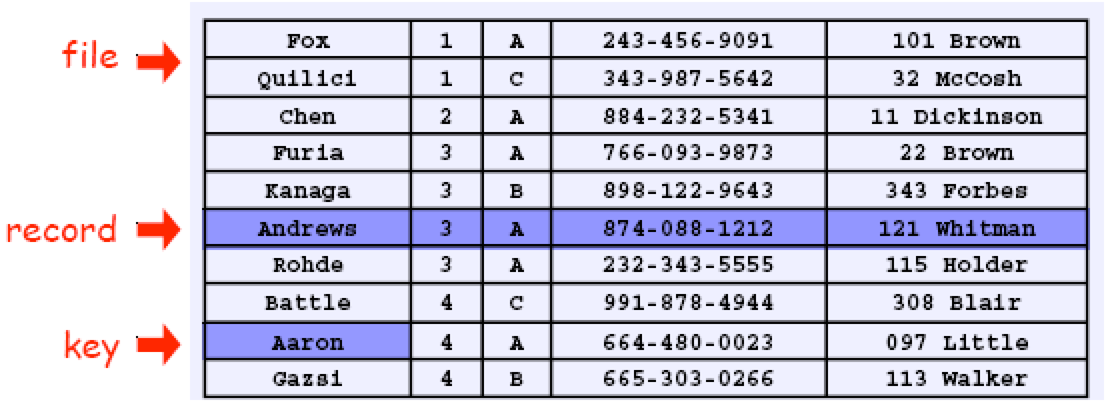
- One of the most common programming tasks in computing.
- Examples of sorting:
  - List containing exam scores sorted from Lowest to Highest or from Highest to Lowest
  - List point pairs of a geometric shape.
  - List of student records and sorted by student number or alphabetically by first or last name.

Some examples

- Sort a list of names.
- Organize an MP3 library.
- Display Google PageRank results.
- List RSS news items in reverse chronological order.
- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.
- Data compression.
- Computer graphics.
- Computational biology.
- Supply chain management.
- Load balancing on a parallel computer.

Why we do sorting?

- Searching for an element in an array will be more efficient. (example: looking for a particular phone number).
- It's always nice to see data in a sorted display. (example: spreadsheet or database application).
- Computers sort things fast - therefore it takes the burden off of the user to search a list.



Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Furia	3	A	766-093-9873	22 Brown
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Aaron	4	A	664-480-0023	097 Little
Gazsi	4	B	665-303-0266	113 Walker

Sort. Rearrange array of N objects into ascending order.

Aaron	4	A	664-480-0023	097 Little
Andrews	3	A	874-088-1212	121 Whitman
Battle	4	C	991-878-4944	308 Blair
Chen	2	A	884-232-5341	11 Dickinson
Fox	1	A	243-456-9091	101 Brown
Furia	3	A	766-093-9873	22 Brown
Gazsi	4	B	665-303-0266	113 Walker
Kanaga	3	B	898-122-9643	343 Forbes
Rohde	3	A	232-343-5555	115 Holder
Quilici	1	C	343-987-5642	32 McCosh

History of Sorting

- Sorting is one of the most important operations performed by computers. In the days of magnetic tape storage before modern databases, database updating was done by sorting transactions and merging them with a master file.

History of Sorting

- It's still important for presentation of data extracted from databases: most people prefer to get reports sorted into some relevant order before flipping through pages of data!

- Insertion
- ShellSort
- MergeSort
- QuickSort
- BubbleSort
- SelectionSort

Insertion Sort

- One of the simplest methods
- Consists of N-1 passes

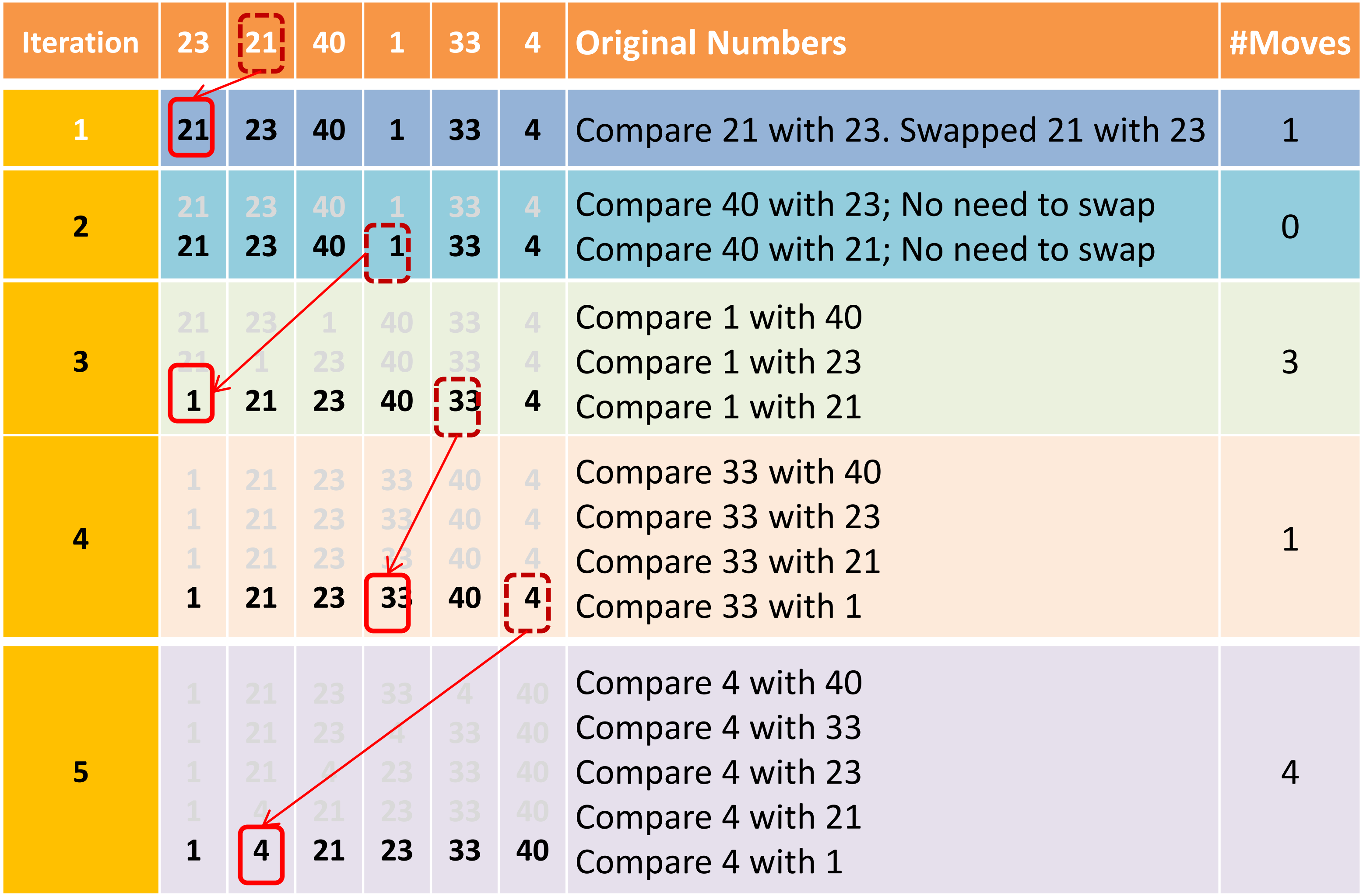
Algorithm

- sorting takes place by inserting a particular element at the appropriate position
  - that’s why the name - insertion sort
- N-1 Iterations
  - In the First iteration, second element A[1] is compared with the first element A[0]
  - In the second iteration third element is compared with first and second element
  - In general, within each iteration :
    - an element is compared with all the elements before it.
    - while comparing if it is found that the element can be inserted at a suitable position, then space is created for it by shifting the other elements one position up and inserts the desired element at the suitable position.
    - procedure is repeated for all the elements in the list.

Insertion Sort demo

Iteration	23	21	40	1	33	4	Original Numbers	#Moves
First	21	23	40	1	33	4	Compare 21 with 23. Swapped 21 with 23	1
Second	21 21	23 23	40 40	1 1	33 33	4 4	Compare 40 with 23; No need to swap Compare 40 with 21; No need to swap	0
Third	21 21 1	23 1 21	1 23 23	40 40 40	33 33 33	4 4 4	Compare 1 with 40, Swap Compare 1 with 23, Swap Compare 1 with 21, Swap	3
Fourth	1 1 1 1	21 21 21 21	23 23 23 23	33 33 33 33	40 40 40 40	4 4 4 4	Compare 33 with 40, Swap Compare 33 with 23 Compare 33 with 21 Compare 33 with 1	1
Fifth	1 1 1 1 1	21 21 21 4 4	23 23 4 21 21	33 4 23 23 23	4 33 33 33 33	40 40 40 40 40	Compare 4 with 40, Swap Compare 4 with 33, Swap Compare 4 with 23, Swap Compare 4 with 21, Swap Compare 4 with 1	4

Insertion Sort demo



Iteration	23	21	40	1	33	4	Original Numbers	#Moves
1	21	23	40	1	33	4	Compare 21 with 23. Swapped 21 with 23	1
2	21 21	23 23	40 40	1 1	33 33	4 4	Compare 40 with 23; No need to swap Compare 40 with 21; No need to swap	0
3	21 21 1	23 1 21	1 23 23	40 40 40	33 33 33	4 4 4	Compare 1 with 40 Compare 1 with 23 Compare 1 with 21	3
4	1 1 1 1	21 21 21 21	23 23 23 23	33 33 33 33	40 40 40 40	4 4 4 4	Compare 33 with 40 Compare 33 with 23 Compare 33 with 21 Compare 33 with 1	1
5	1 1 1 1 1	21 21 21 4 4	23 23 4 21 21	33 4 23 23 23	4 33 33 33 33	40 40 40 40 40	Compare 4 with 40 Compare 4 with 33 Compare 4 with 23 Compare 4 with 21 Compare 4 with 1	4

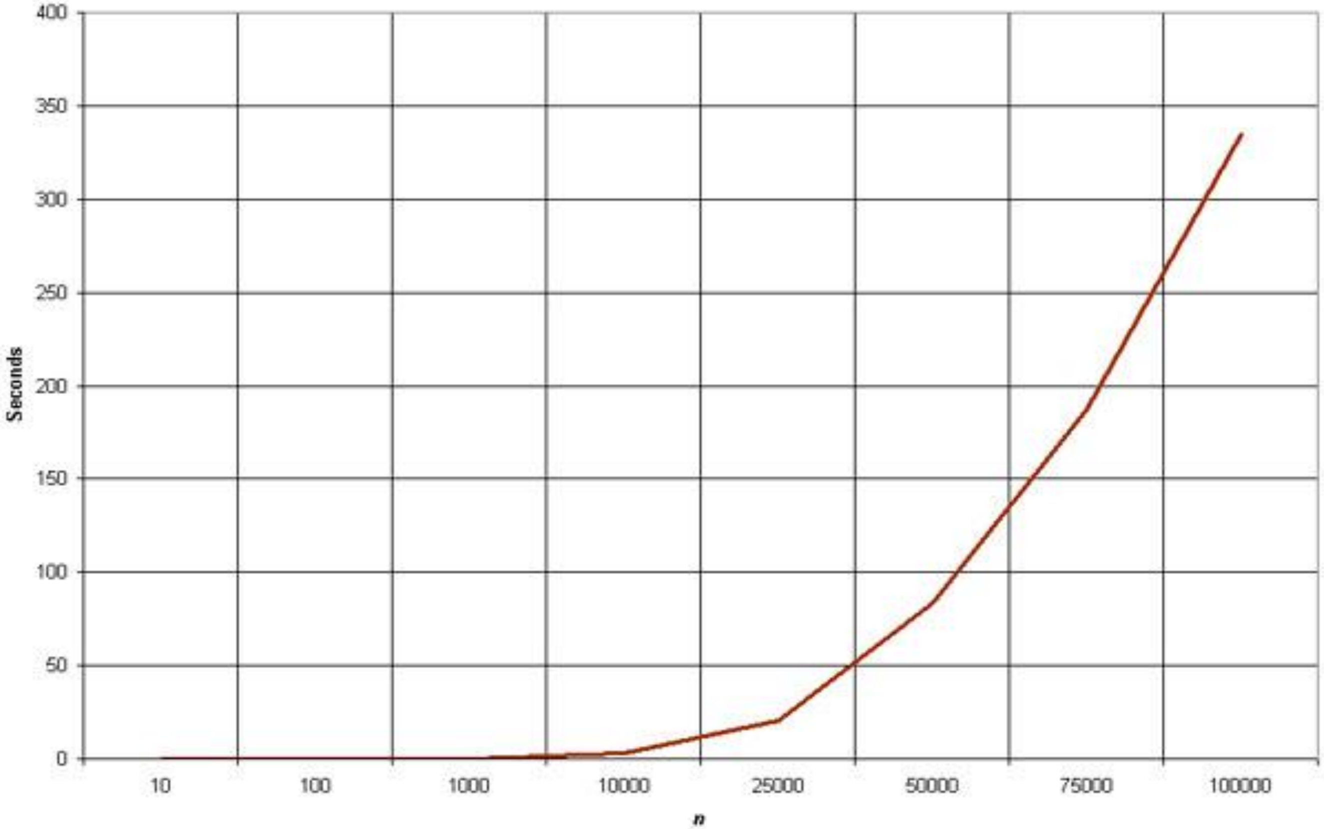
Insertion sort in daily life

- Card sorting during a game of cards
  - To sort the cards in your hand you extract a card, shift the remaining cards, and then insert the extracted card in the correct place. This process is repeated until all the cards are in the correct sequence

Insertion Sort runtimes

- Best case: O(n) It occurs when the data is in sorted order. After making one pass through the data and making no insertions, insertion sort exits.
- Average case: Θ(n^2) since there is a wide variation with the running time.
- Worst case: O(n^2) if the numbers were sorted in reverse order.

Empirical Analysis of Insertion Sort



The graph demonstrates the *n*^2 complexity of the insertion sort.

Source: http://linux.wku.edu/~lamonml/algor/sort/insertion.html

Insertion Sort

- The insertion sort is a good choice for sorting lists of a few thousand items or less.
- Advantage of Insertion Sort is that it is relatively simple and easy to implement.
- Disadvantage of Insertion Sort is that it is not efficient to operate with a large list or input size

MERGE SORT

Merge Sort

- Divide the array at its midpoint
- Recursively apply Merge to sort both halves
- nLOG(n)

Algo

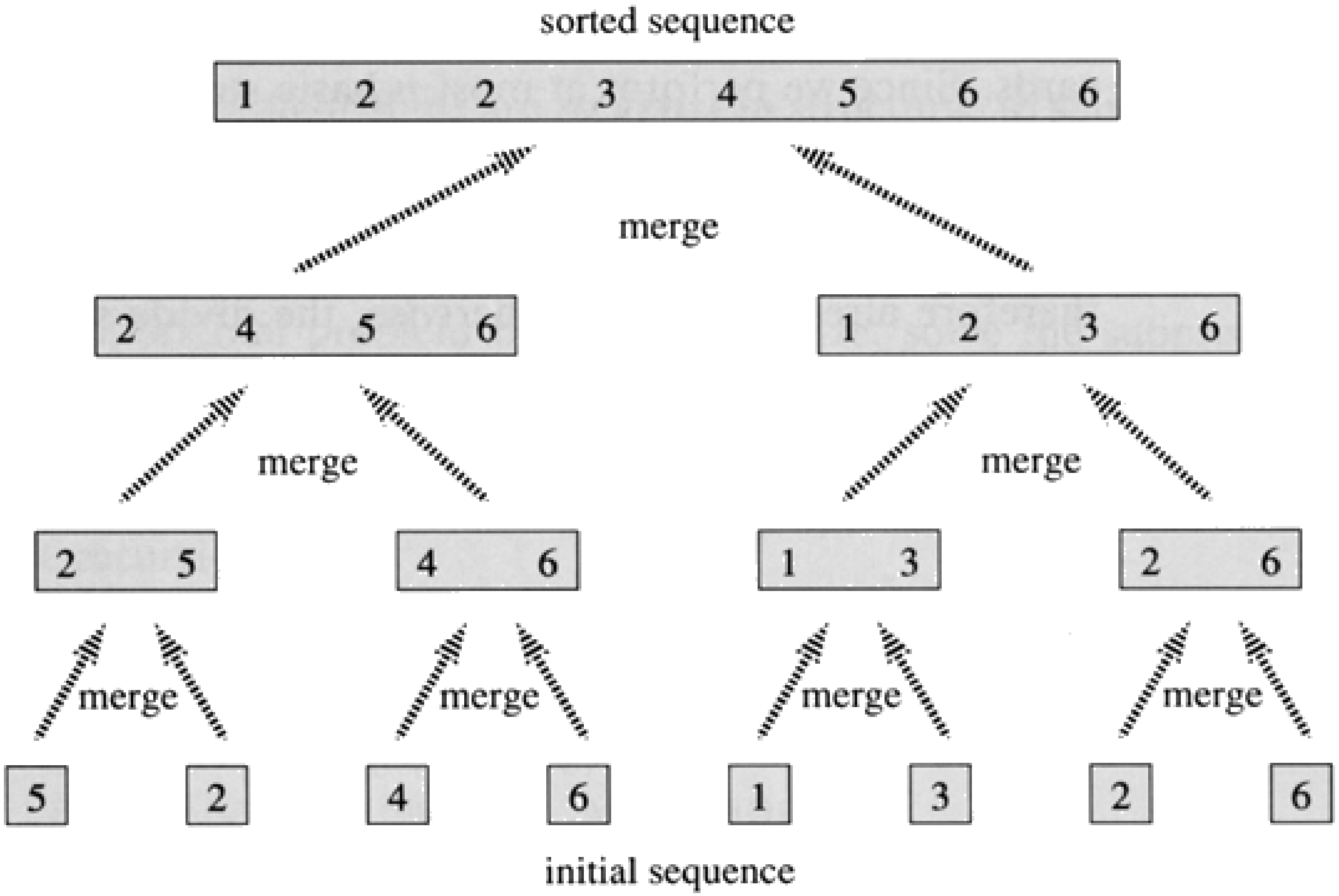
```
void mergesort(int lo, int hi)
{
   if (lo<hi)
   {
      int m=(lo+hi)/2;
      mergesort(lo, m);
      mergesort(m+1, hi);
      merge(lo, m, hi);
   }
}
```

- index ***m*** in the middle between ***lo*** and ***hi*** is determined.
- Then the first part of the sequence (from ***lo*** to ***m***) and the second part (from ***m*+1** to ***hi***) are sorted by recursive calls of *mergesort*.
- Then the two sorted halves are merged by procedure *merge*. Recursion ends when ***lo*** = ***hi***,
  - i.e. when a subsequence consists of only one element.
- The main work of the Mergesort algorithm is performed by function *merge*.
- Different possibilities to implement this function

Merge (Straight)

```
// Straightforward variant
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++];
}
```

- Copy Array (a) into (b)
- Repeat while (I < m and J <= hi)
  - Copy min(b[i], b[j]) to a[k]
- Copy back the remaining (if any) to (b)


sorted sequence
1 2 2 3 4 5 6 6
merge
2 4 5 6
1 2 3 6
merge
merge
2 5
4 6
1 3
2 6
merge
merge
merge
merge
5
2
4
6
1
3
2
6
initial sequence

based on the **divide-and-conquer**

- Its worst-case running time has a lower order of growth than insertion sort.
- Since we are dealing with subproblems, we state each subproblem as sorting a subarray *A*[*p* .. *r*]. Initially, *p* = 1 and *r* = *n*, but these values change as we recurse through subproblems.
- To sort *A*[*p* .. *r*]:
- 1. **Divide Step**
  - If a given array *A* has zero or one element, simply return; it is already sorted. Otherwise, split *A*[*p* .. *r*] into two subarrays *A*[*p* .. *q*] and *A*[*q* + 1 .. *r*], each containing about half of the elements of *A*[*p* .. *r*]. That is, *q* is the halfway point of *A*[*p* .. *r*].
- 2. **Conquer Step**
  - Conquer by recursively sorting the two subarrays *A*[*p* .. *q*] and *A*[*q* + 1 .. *r*].
- 3. **Combine Step**
  - Combine the elements back in *A*[*p* .. *r*] by merging the two sorted subarrays *A*[*p* .. *q*] and *A*[*q* + 1 .. *r*] into a sorted sequence. To accomplish this step, we will define a procedure MERGE (*A*, *p*, *q*, *r*).
- Note that the recursion bottoms out when the subarray has just one element, so that it is trivially sorted.

Idea Behind Linear Time Merging

- Think of two piles of cards, Each pile is sorted and placed face-up on a table with the smallest cards on top. We will merge these into a single sorted pile, face-down on the table.
- Each Basic step (should take constant time, since we check just the two top cards)
  - Choose the smaller of the two top cards.
  - Remove it from its pile, thereby exposing a new top card.
  - Place the chosen card face-down onto the output pile.
  - Repeatedly perform basic steps until one input pile is empty.
  - Once one input pile empties, just take the remaining input pile and place it face-down onto the output pile.

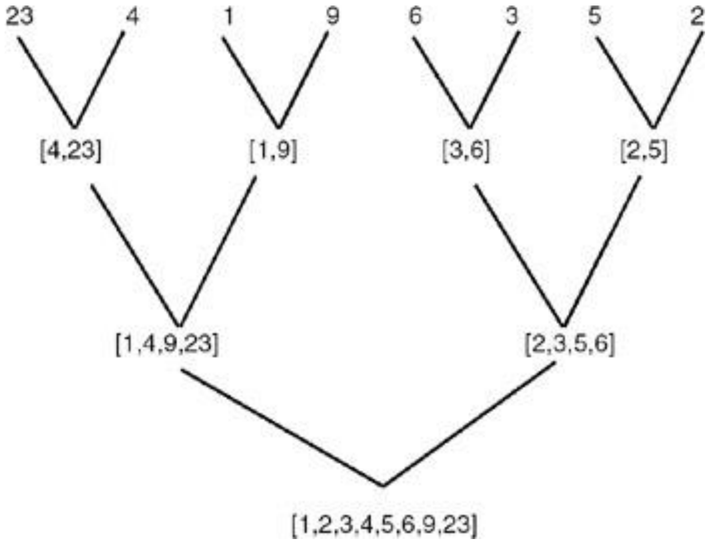
MERGE (*A*, *p*, *q*, *r*)

```
$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
Create arrays $L[1 \, . \, . \, n_1 + 1]$ and $R[1 \, . \, . \, n_2 + 1]$
FOR $i \leftarrow 1$ TO $n_1$
    DO $L[i] \leftarrow A[p + i - 1]$
FOR $j \leftarrow 1$ TO $n_2$
    DO $R[j] \leftarrow A[q + j]$
$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
FOR $k \leftarrow p$ TO $r$
    DO IF $L[i] \leq R[j]$
        THEN $A[k] \leftarrow L[i]$
            $i \leftarrow i + 1$
        ELSE $A[k] \leftarrow R[j]$
            $j \leftarrow j + 1$
```

```
#define MAX 10
void merge(int list[],int list1[],int k,int m,int n) {
    int i,j;
   i=k;
   j = m+1;
   while( i <= m && j <= n)   {
      if(list[i] <= list[j])   {
             list1[k] = list[i];
             i++;      k++;
      }    else     {
            list1[k] = list[j];
         j++;      k++;
      }
   }
   while(i <= m)   {
         list1[k] = list[i];
      i++;    k++;
   }
   while (i <= n )   {
         list1[k] = list[j];
      j++;    k++;
   }
}
```

```
void mpass( int list[],int list1[],int l,int n) {
   int i;
   i = 0;
   while( i <= (n-2*l+1))   {
      merge(list,list1,i,(i+l-1),(i+2*l-1));
      i = i + 2*l;
   }
   if((i+l-1) < n)
       merge(list,list1,i,(i+l-1),n);
   else
       while (i <= n )  {
              list1[i] = list[i];
              i++;
       }
   }
void readlist(int list[],int n)    {
      int i;
      printf("Enter the elements\n");
      for(i=0;i<n;i++)
          scanf("%d",&list[i]);
   }
```

```
void printlist(int list[],int n)
  {
    int i;
    printf("The elements of the list are: \n");
    for(i=0;i<n;i++)
       printf("%d\t",list[i]);
  }

void main()
{
  int list[MAX], n;
  printf("Enter the number of elements in the list max = 10\n");
  scanf("%d",&n);
  readlist(list,n);
  printf("The list before sorting is:\n");
  printlist(list,n);
  msort(list,n-1);
  printf("The list after sorting is:\n");
  printlist(list,n);
}
```


23
4
1
9
6
3
5
2
[4,23]
[1,9]
[3,6]
[2,5]
[1,4,9,23]
[2,3,5,6]
[1,2,3,4,5,6,9,23]

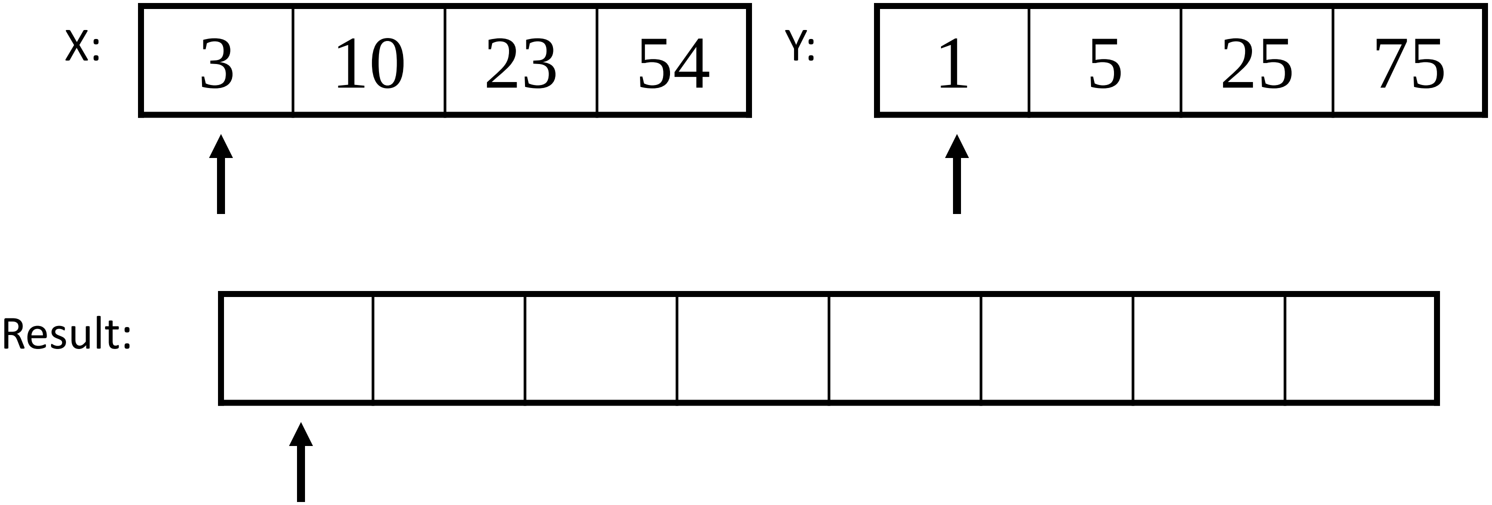
Merging

- The key to Merge Sort is merging two sorted lists into one, such that if you have two lists X $(x_1 \leq x_2 \leq \cdots \leq x_m)$ and Y$(y_1 \leq y_2 \leq \cdots \leq y_n)$ the resulting list is Z$(z_1 \leq z_2 \leq \cdots \leq z_{m+n})$
- Example:

L_1 = { 3 8 9 } L_2 = { 1 5 7 }

merge(L_1, L_2) = { 1 3 5 7 8 9 }

Merging (cont.)



X:

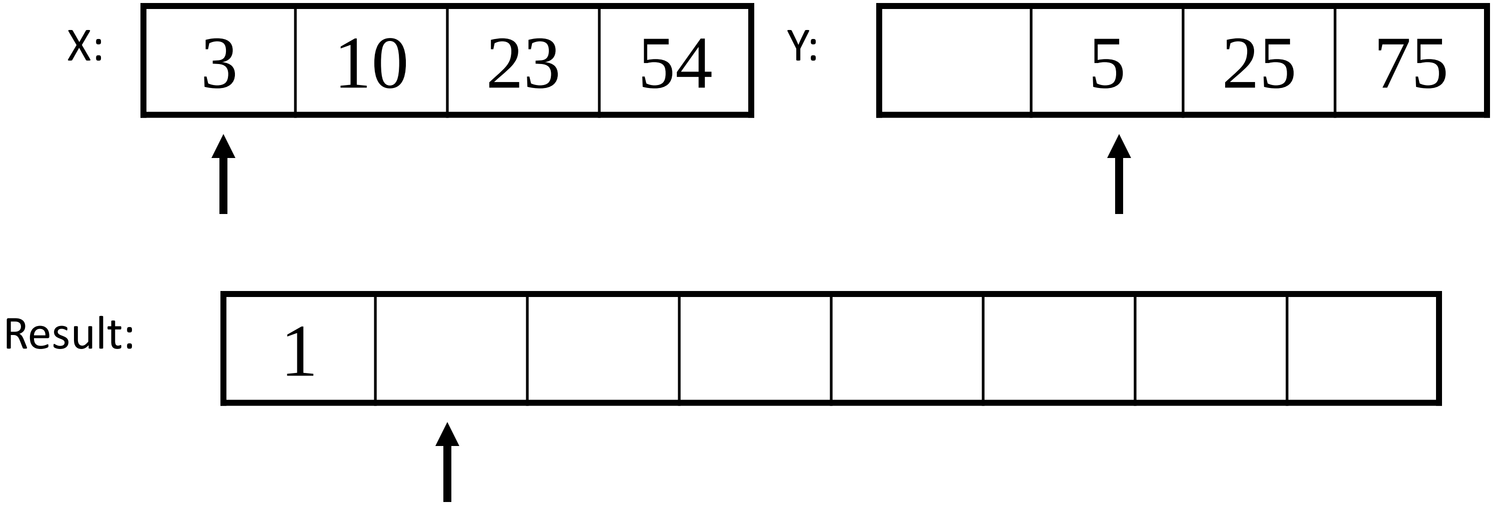
3	10	23	54

Y:

1	5	25	75

Result:

Merging (cont.)



X:

3	10	23	54

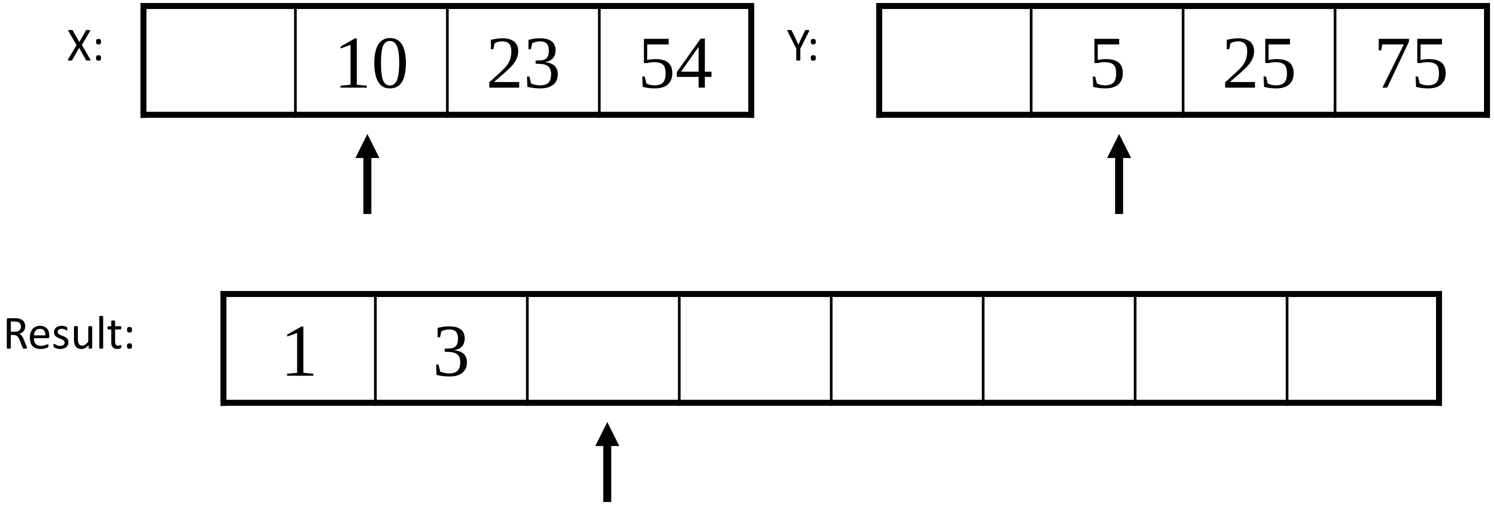
Y:

	5	25	75

Result:

1							

Merging (cont.)



X:

	10	23	54

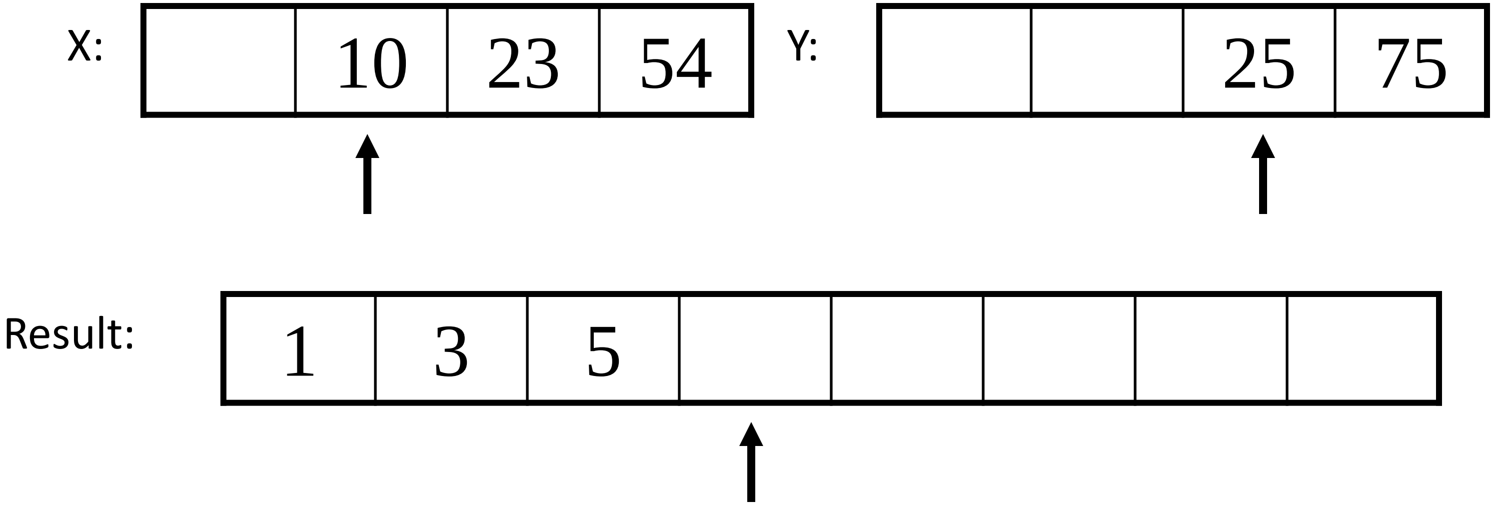
Y:

	5	25	75

Result:

1	3						

Merging (cont.)



X:

	10	23	54

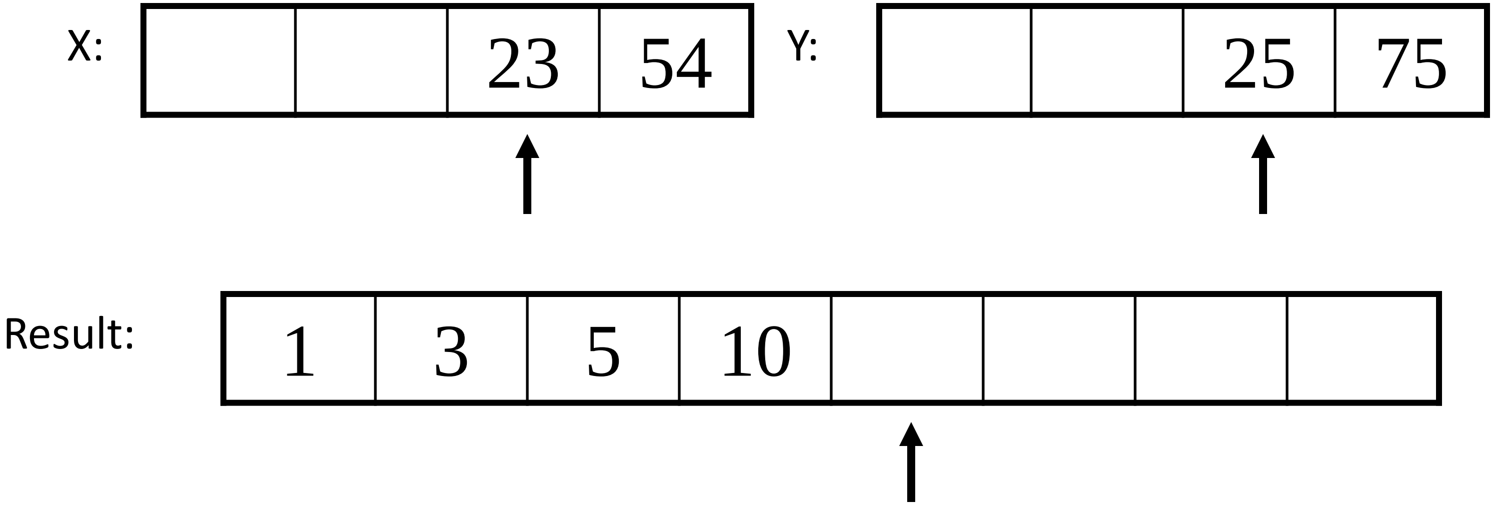
Y:

		25	75

Result:

1	3	5					

Merging (cont.)



X:

		23	54

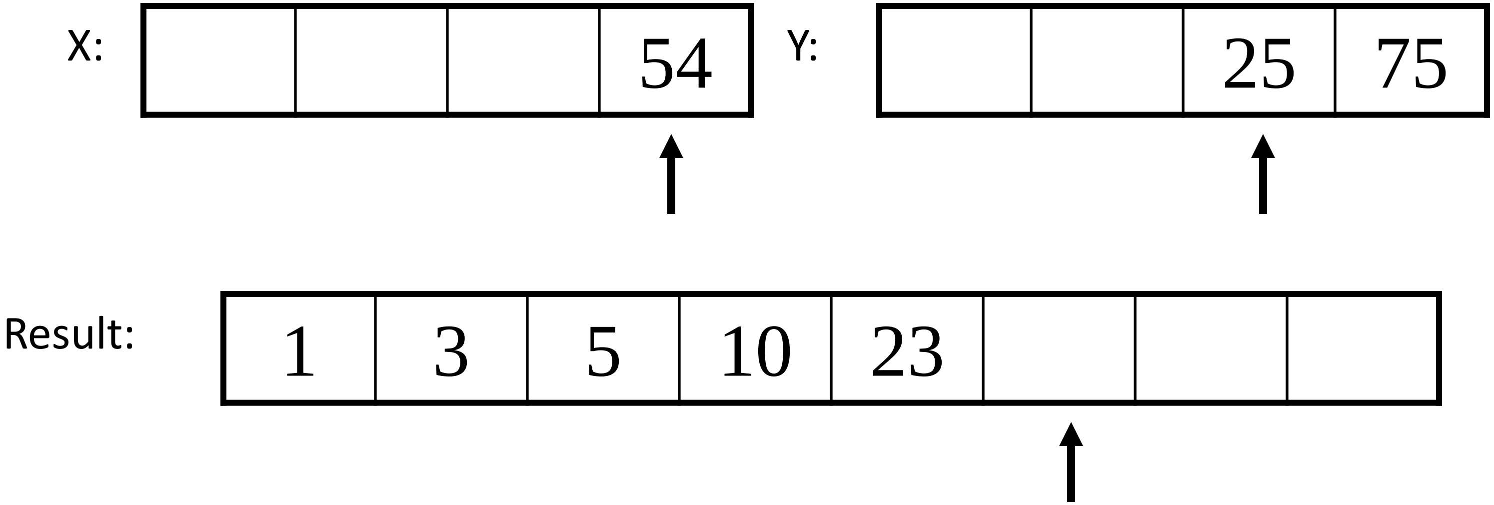
Y:

		25	75

Result:

1	3	5	10				

Merging (cont.)



X:

			54

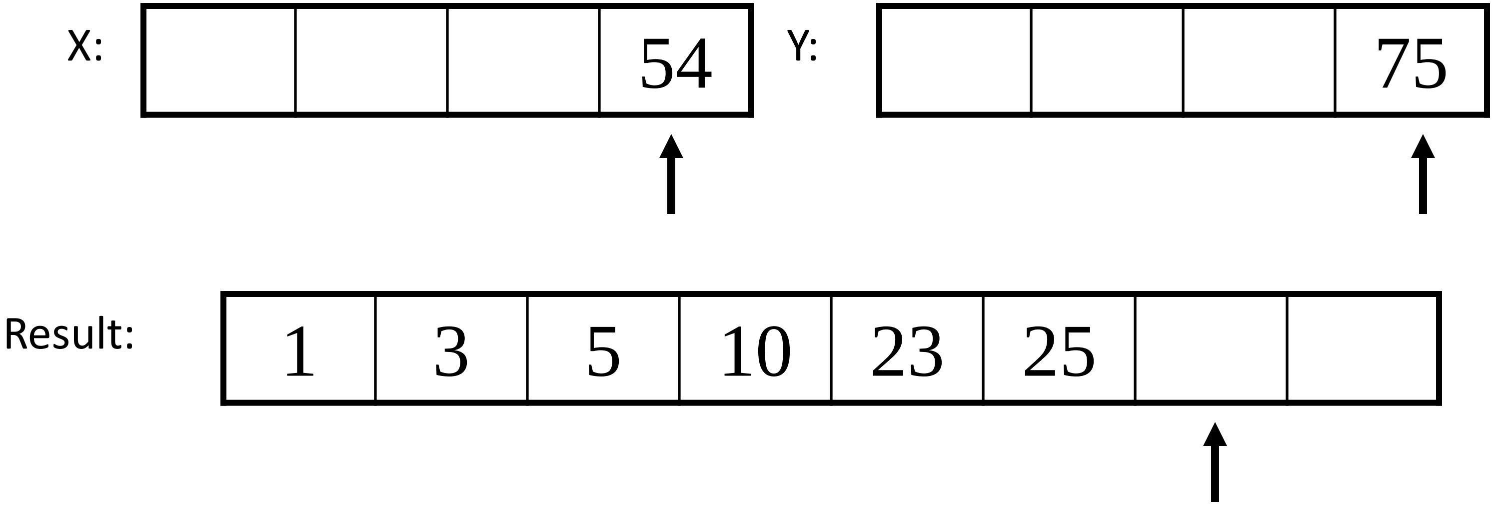
Y:

		25	75

Result:

1	3	5	10	23			

Merging (cont.)



X:

			54

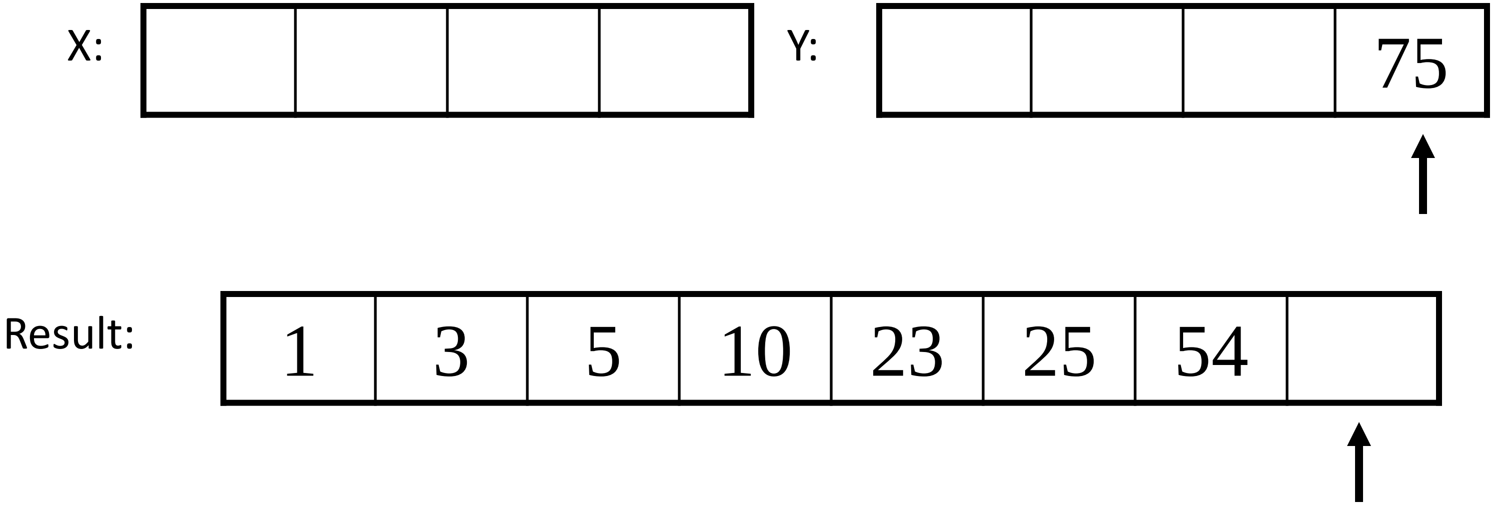
Y:

			75

Result:

1	3	5	10	23	25		

Merging (cont.)



X:

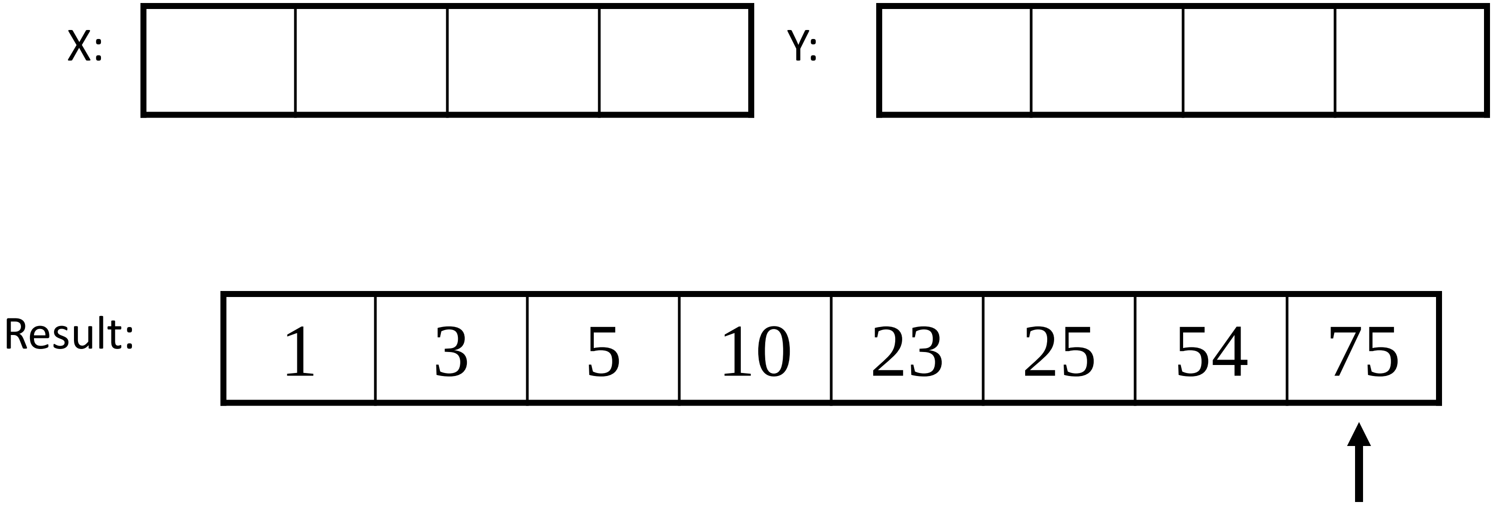
Y:

			75

Result:

1	3	5	10	23	25	54	

Merging (cont.)



X:

Y:

Result:

1	3	5	10	23	25	54	75

Divide And Conquer

- Merging a two lists of one element each is the same as sorting them.
- Merge sort divides up an unsorted list until the above condition is met and then sorts the divided parts back together in pairs.
- Specifically this can be done by recursively dividing the unsorted list in half, merge sorting the right side then the left side and then merging the right and left back together.

Merge Sort Algorithm

Given a list L with a length k:

- If k == 1 → the list is sorted
- Else:
    - Merge Sort the left side (0 thru k/2)
    - Merge Sort the right side (k/2+1 thru k)
    - Merge the right side with the left side

Merge Sort Example

99	6	86	15	58	35	86	4	0

Merge Sort Example

99	6	86	15	58	35	86	4	0

99	6	86	15

58	35	86	4	0

Merge Sort Example

99	6	86	15	58	35	86	4	0

99	6	86	15

58	35	86	4	0

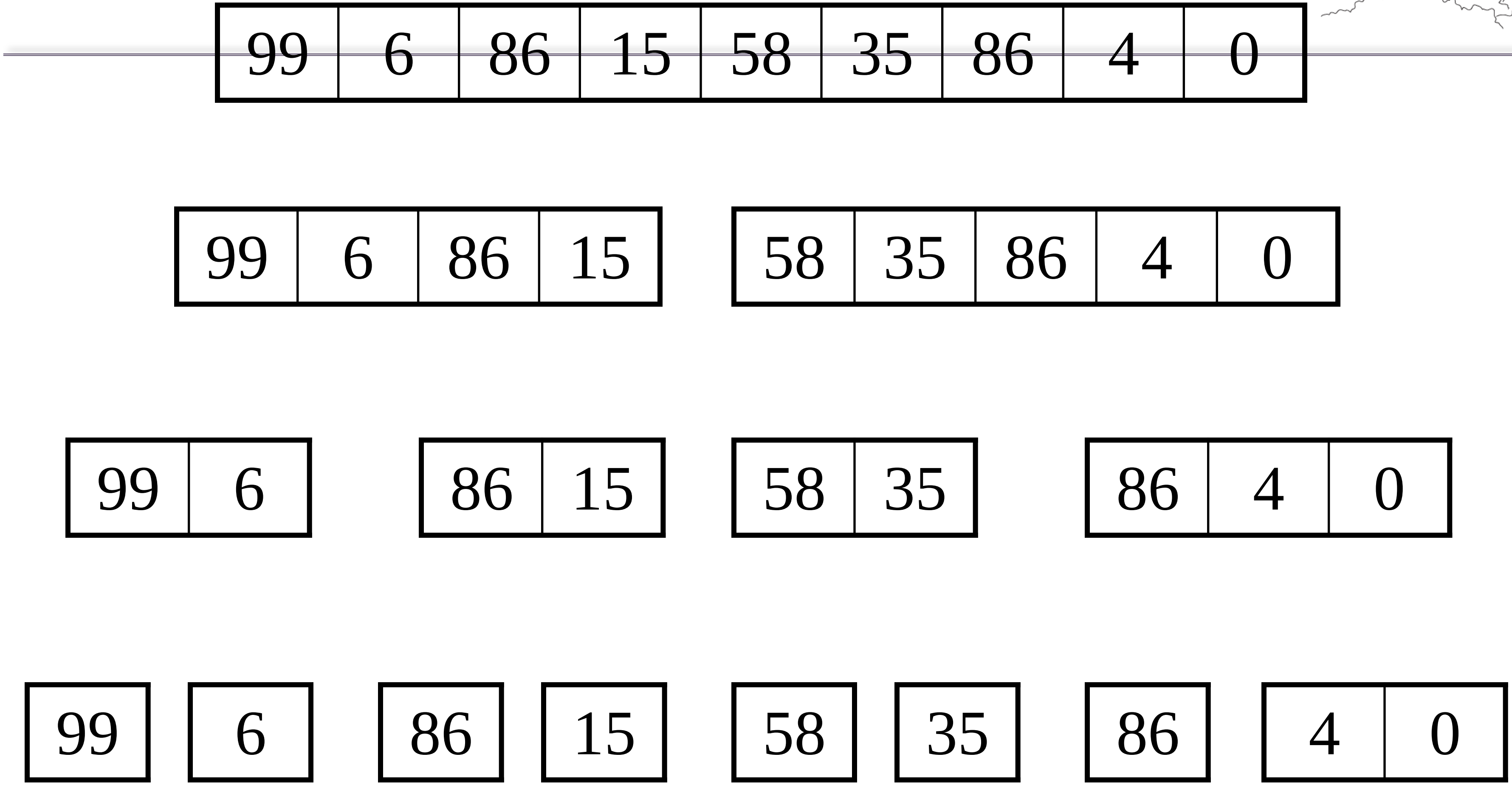
99	6

86	15

58	35

86	4	0

Merge Sort Example



Merge Sort Example

99	6	86	15	58	35	86	4	0

99	6	86	15

58	35	86	4	0

99	6

86	15

58	35

86	4	0

99

6

86

15

58

35

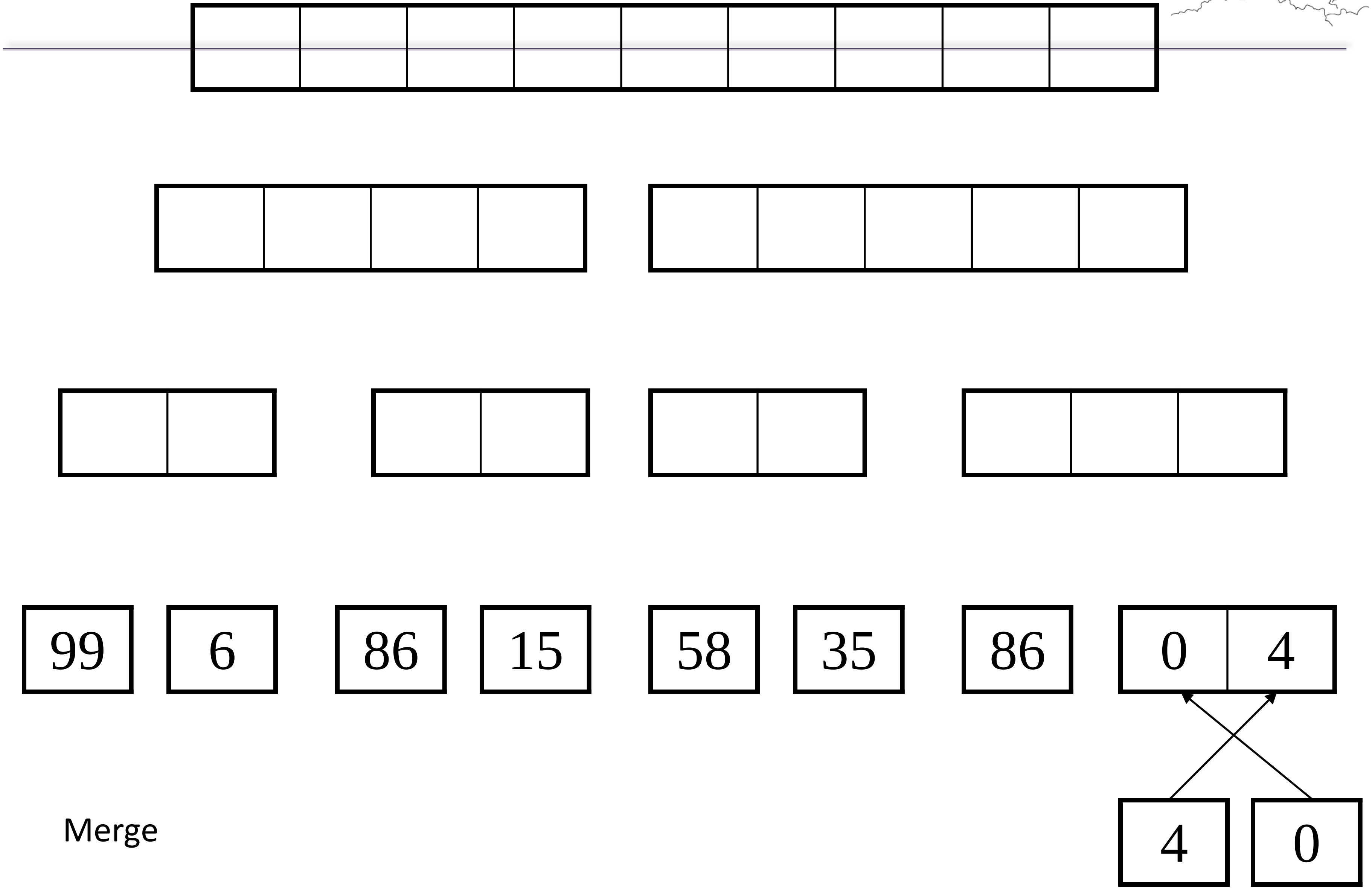
86

4	0

4

0

Merge Sort Example

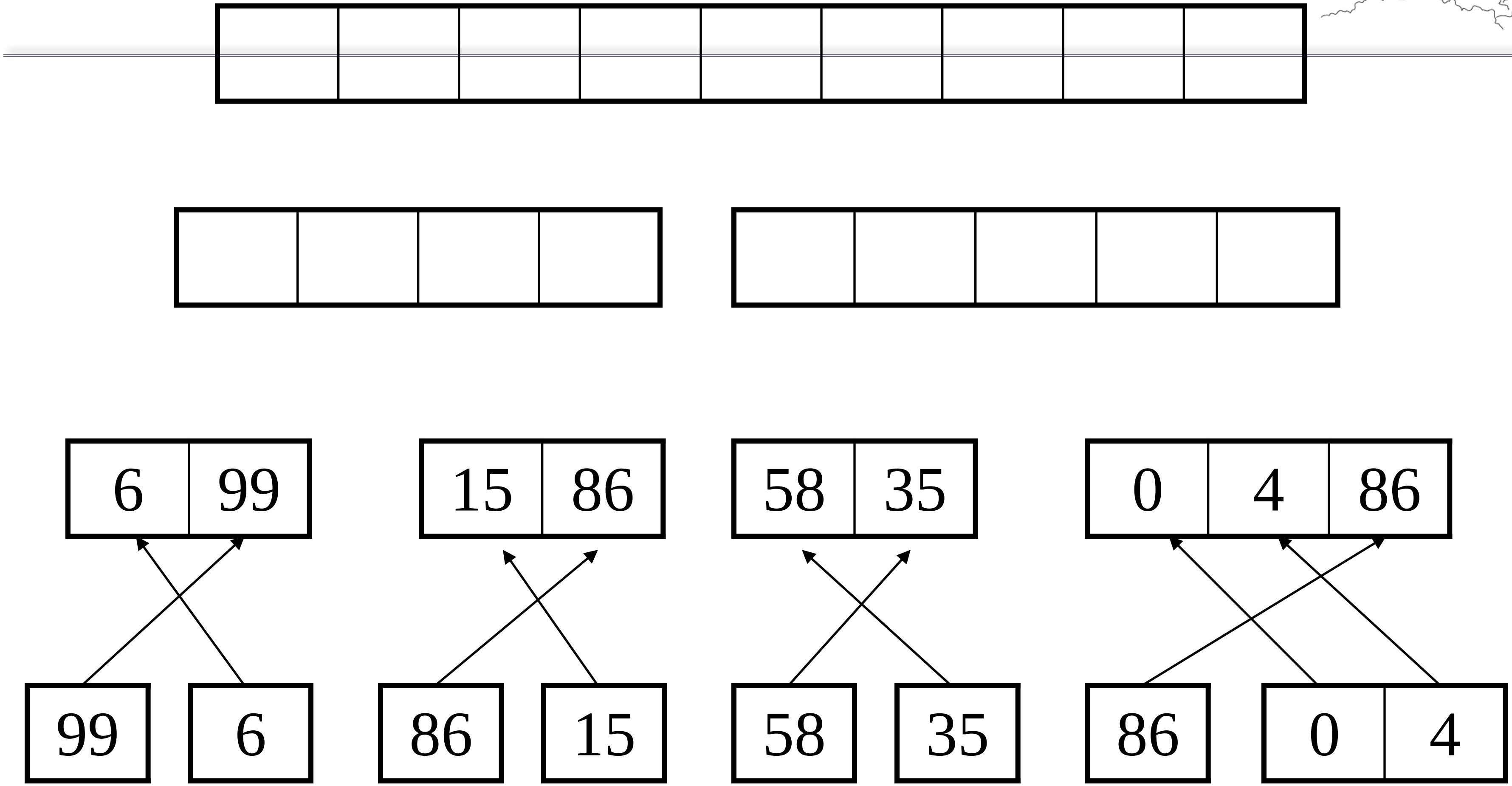


99 6 86 15 58 35 86 0 4

4 0

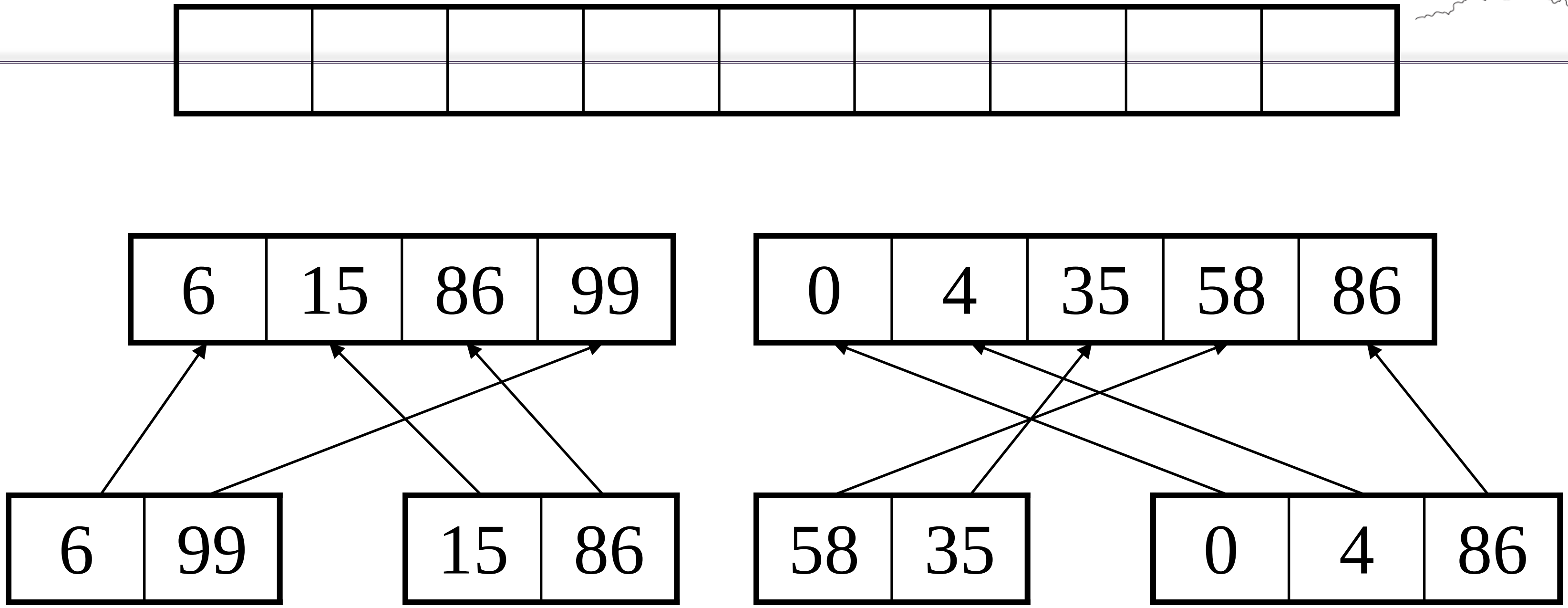
Merge

Merge Sort Example



Merge

Merge Sort Example



6 15 86 99

0 4 35 58 86

6 99

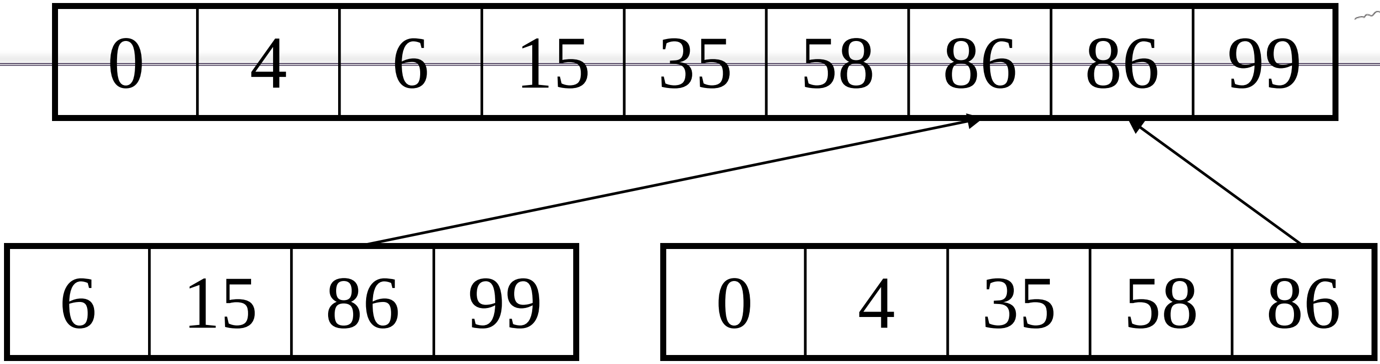
15 86

58 35

0 4 86

Merge

Merge Sort Example



0	4	6	15	35	58	86	86	99

6	15	86	99

0	4	35	58	86

Merge

Merge Sort Example

0	4	6	15	35	58	86	86	99

Implementing Merge Sort

- There are two basic ways to implement merge sort:
  - In Place: Merging is done with only the input array
    - Pro: Requires only the space needed to hold the array
    - Con: Takes longer to merge because if the next element is in the right side then all of the elements must be moved down.
  - Double Storage: Merging is done with a temporary array of the same size as the input array.
    - Pro: Faster than In Place since the temp array holds the resulting array until both left and right sides are merged into the temp array, then the temp array is appended over the input array.
    - Con: The memory requirement is doubled

Merge Sort Analysis

The Double Memory Merge Sort runs O (N log N) for all cases, because of its Divide and Conquer approach.

$$T(N) = 2T(N/2) + N = O(N \log N)$$

- (i.e.) Best Case, Average Case, and Worst Case = O(N logN)

SHELL SORT

Shell Sort

- Invented by Donald Shell in 1959.
- 1st algorithm to break the quadratic time barrier but few years later, a sub quadratic time bound was proven
- Shellsort works by comparing elements that are distant rather than adjacent elements in an array.
- Shellsort uses a sequence h_1, h_2, …, h_t called the ***increment sequence***. Any increment sequence is fine as long as $h_1 = 1$ and some other choices are better than others
- Shellsort improves on the efficiency of insertion sort by *quickly* shifting values to their destination

Shell Sort

- Improves on insertion sort.
- Starts by comparing elements far apart, then elements less far apart, and finally comparing adjacent elements (effectively an insertion sort).

- By this stage the elements are sufficiently sorted that the running time of the final stage is much closer to $O(N)$ than $O(N^2)$.

Principle

- arrange the data sequence in a two-dimensional array
- sort the columns of the array
  - The effect is that the data sequence is partially sorted. The process above is repeated, but each time with a narrower array, i.e. with a smaller number of columns. In the last step, the array consists of only one column

Shell Sort

- Step 1: divide the original list into smaller lists.
- Step 2: sort individual sub lists using any known sorting algorithm (like bubble sort, insertion sort, selection sort, etc).

Shell Sort - Parameters

- how should I divide the list? Which sorting algorithm to use? How many times I will have to execute steps 1 and 2? And the most puzzling question if I am anyway using bubble, insertion or selection sort then how I can achieve improvement in efficiency?
- For dividing the original list into smaller lists, we choose a value K, which is known as increment. Based on the value of K, we split the list into K sub lists.
- For example, if our original list is x[0], x[1], x[2], x[3], x[4]….x[99] and we choose 5 as the value for increment, K then we get the following sub lists.
  - first_list = x[0], x[5], x[10], x[15]…….x[95]
  - second_list =x[1], x[6], x[11], x[16]…….x[96]
  - third_list =x[2], x[7], x[12], x[17]…….x[97]
  - forth_list =x[3], x[8], x[13], x[18]…….x[98]
  - fifth_list =x[4], x[9], x[14], x[19] …….x[99]
- So the ith sub list will contain every Kth element of the original list starting from index i-1.

Shell Sort

- In-place sort
- O(nLog(n))
- Spacing dependent
  - Start with large and converge into small ultimately 1
- Formal analysis is difficult
- Arriving at Optimal spacing values – difficult to analyze

- Since for every iteration we are decreasing the value of the increment (K) the algorithm is also known as “diminishing increment sort”.

Shellsort Example

- Sort: 18 32 12 5 38 33 16 2

8 Numbers to be sorted, Shell’s increment will be floor(n/2)

*** floor(8/2) → floor(4) = 4**

increment 4: **1 2 3 4**

18 32 12 5 38 33 16 2

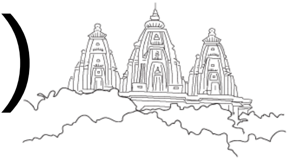
Step **1**) Only look at **18** and **38** and sort in order ;
18 and **38** stays at its current position because they are in order.

Step **2**) Only look at **32** and **33** and sort in order ;
32 and **33** stays at its current position because they are in order.

Step **3**) Only look at **12** and **16** and sort in order ;
12 and **16** stays at its current position because they are in order.

Step **4**) Only look at **5** and **2** and sort in order ;
2 and **5** need to be switched to be in order.

Shellsort Example (con't)



- Sort: 18 32 12 5 38 33 16 2

Resulting numbers after increment 4 pass:

18 32 12 *2* 38 33 16 *5*

*** floor(4/2) ➔ floor(2) = 2**

increment 2: **1 2**

18 32 12 2 38 33 16 5

Step **1**) Look at **18**, **12**, **38**, **16** and sort them in their appropriate location:

12 32 16 2 18 33 38 5

Step **2**) Look at **32**, **2**, **33**, **5** and sort them in their appropriate location:

12 2 16 5 18 32 38 33

Shellsort Example (con’t)

- Sort: 18 32 12 5 38 33 16 2

*** floor(2/2) ➔ floor(1) = 1**

increment 1: 1

12	**2**	**16**	**5**	**18**	**32**	**38**	**33**
2	**5**	**12**	**16**	**18**	**32**	**33**	**38**

The last increment or phase of Shellsort is basically an Insertion Sort algorithm.

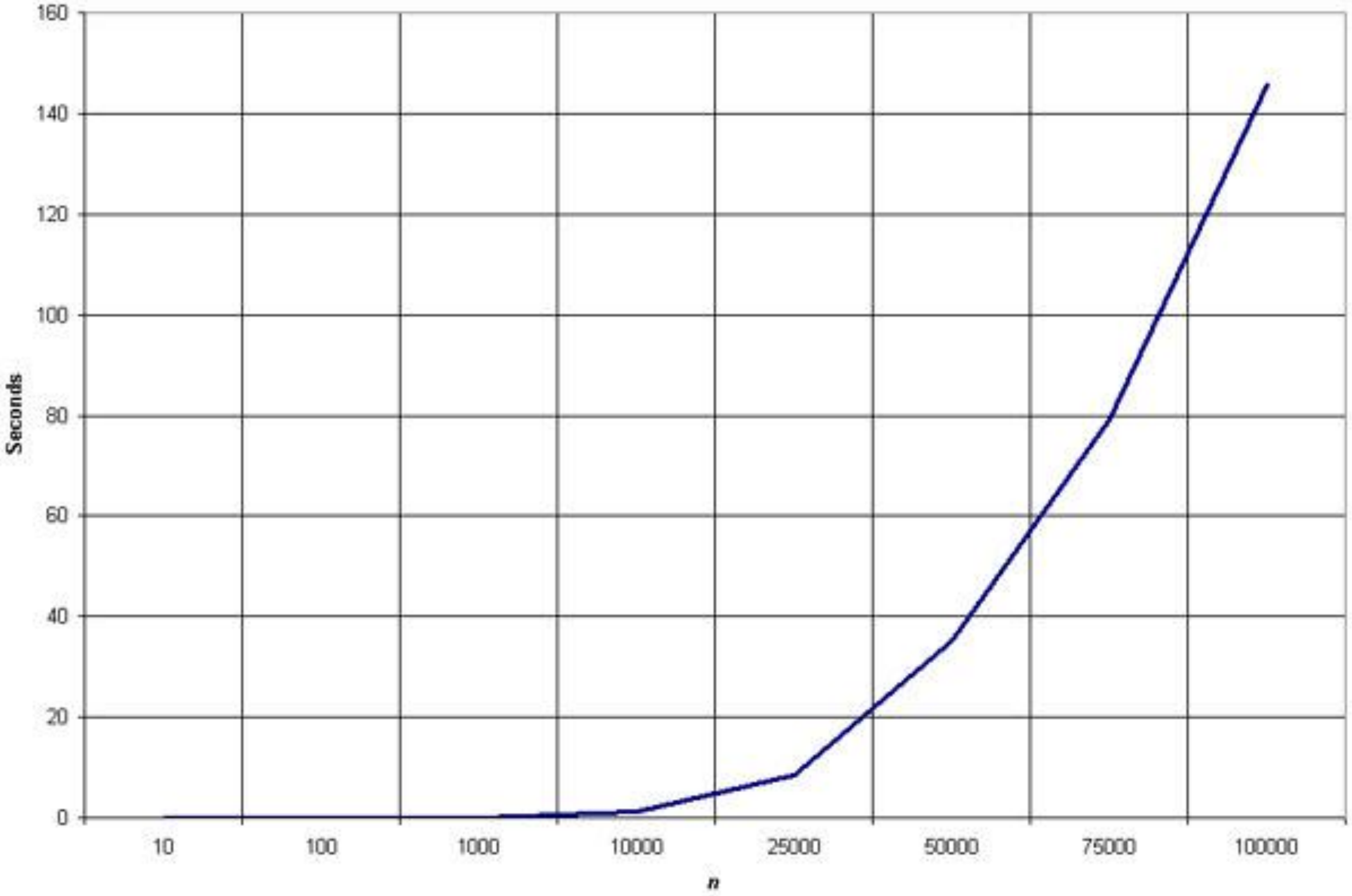
Ad

- Advantages
  - efficient for medium size lists.
    - For bigger lists, the algorithm is not the best choice. Fastest of all O(N^2) sorting algorithms.
  - 5 times faster than the bubble sort and a little over twice as fast as the insertion sort, its closest competitor.
- Disadvantages
  - it is a complex algorithm and its not nearly as efficient as the merge, heap, and quick sorts.
  - The shell sort is still significantly slower than the merge, heap, and quick sorts, but its relatively simple algorithm makes it a good choice for sorting lists of less than 5000 items unless speed important. It's also an excellent choice for repetitive sorting of smaller lists.

Best Case

- Shell Sort best case is when the array is already sorted in the right order and number are elements are large. The number of comparisons is less

Empirical Analysis of Shellsort



Example

2	23	1	4	5	6	8	10	12	18

2	23	1	4
5	6	8	10
12	18		

2	23	1	4
5	6	8	10
12	18		

2	6	1
4	5	18
8	10	12
23		

2	5	1
4	6	12
8	10	18
23		

2	5
1	4
6	12
8	10
18	23

1	4
2	5
6	10
8	12
18	23

- Increment sequence (h1, h2, … hk)

QUICK SORT

Quicksort

- Efficient sorting algorithm
  - Discovered by C.A.R. Hoare
- Example of Divide and Conquer algorithm
- Two phases
  - Partition phase
    - Divides the work into half
  - Sort phase
    - Conquers the halves!

Quick Sort

- Quicksort sorts a list based on the divide and conquer strategy.
- Divide the list into two sub-lists, sort these sub-lists and recursively until the list is sorted; The basic steps of quicksort algorithm are as follows:
    - Choose a key element in the list which is called a pivot.
    - Reorder the list with the rule that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it. After the partitioning, the pivot is in its final position.
    - Recursively reorder two sub-lists: the sub-list of lesser elements and the sub-list of greater elements.

Algo

- QuickSort (Set) {
  - IF Set contains 1 element, return (Set)
  - Choose a Pivot from Set
  - Let S1, S2, S3 be the sequences of elements in Set such that they are LESS THAN, EQUAL TO and MORE THAN the Pivot, respectively
  - Return QuickSort(S1), followed by S2 followed by QuickSort(S3)
- }

Example

We are given array of n integers to sort:

40	20	10	80	60	50	7	30	100

Pick Pivot Element

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

40	20	10	80	60	50	7	30	100

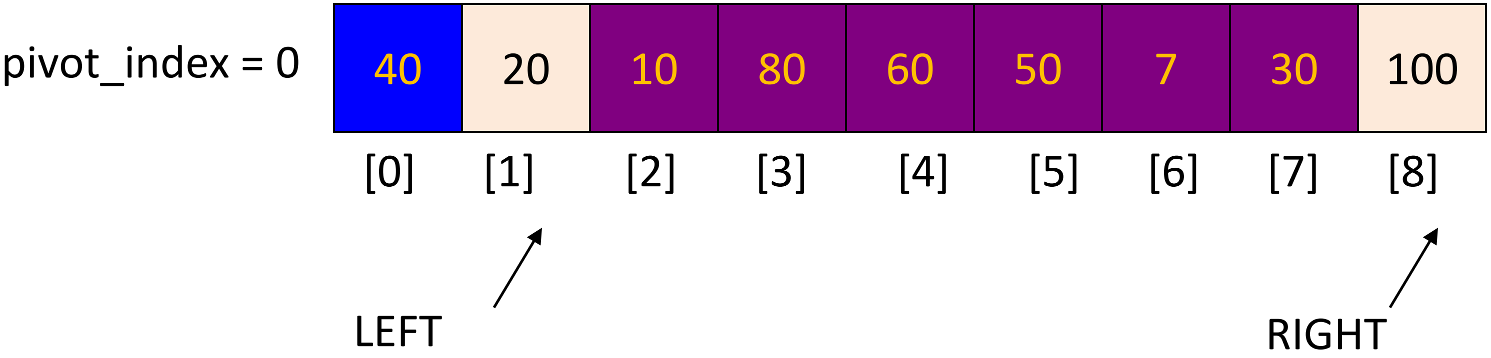
Partitioning Array

Given a pivot, partition the elements of the array such that the resulting array consists of:

1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

The sub-arrays are stored in the original data array.

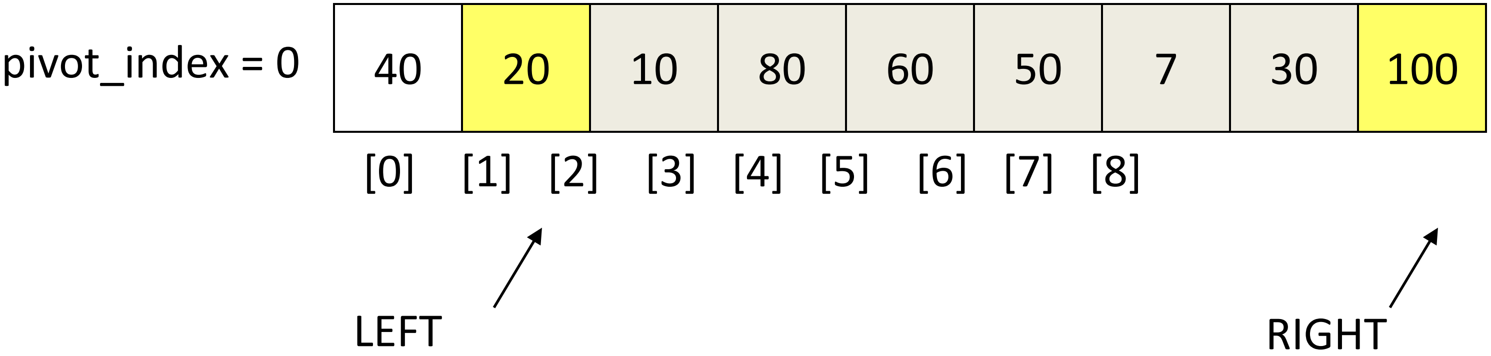
Partitioning loops through, swapping elements below/above pivot.



pivot_index = 0

40	20	10	80	60	50	7	30	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

```
1.  While data[LEFT] <= data[pivot]
        ++LEFT
```



pivot_index = 0

40	20	10	80	60	50	7	30	100

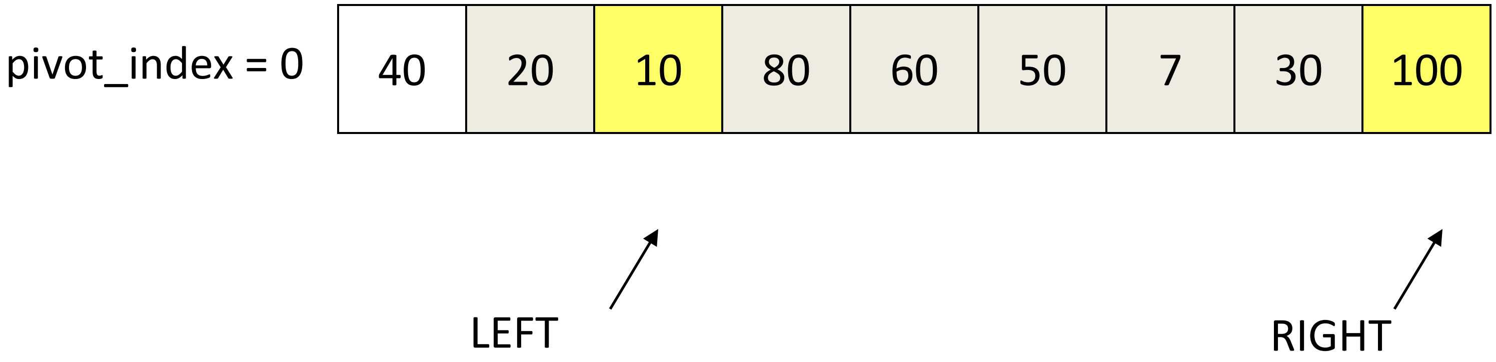
[0] [1] [2] [3] [4] [5] [6] [7] [8]

LEFT

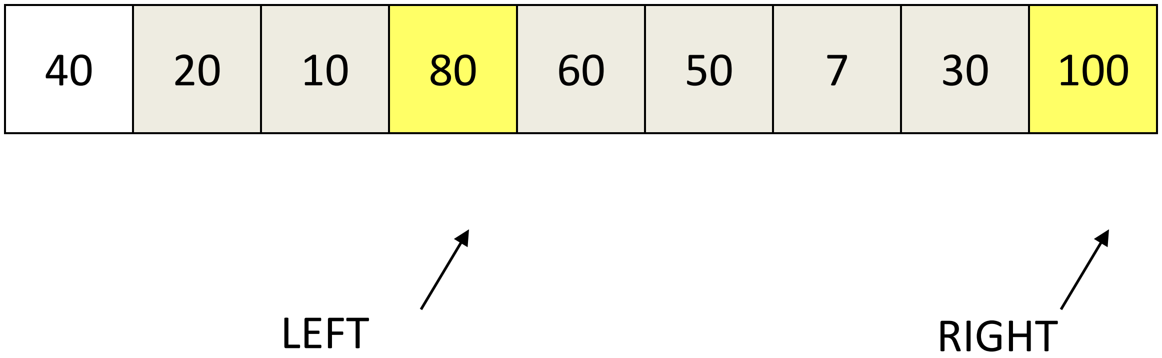
RIGHT

1. While data[LEFT] <= data[pivot]

   ++LEFT



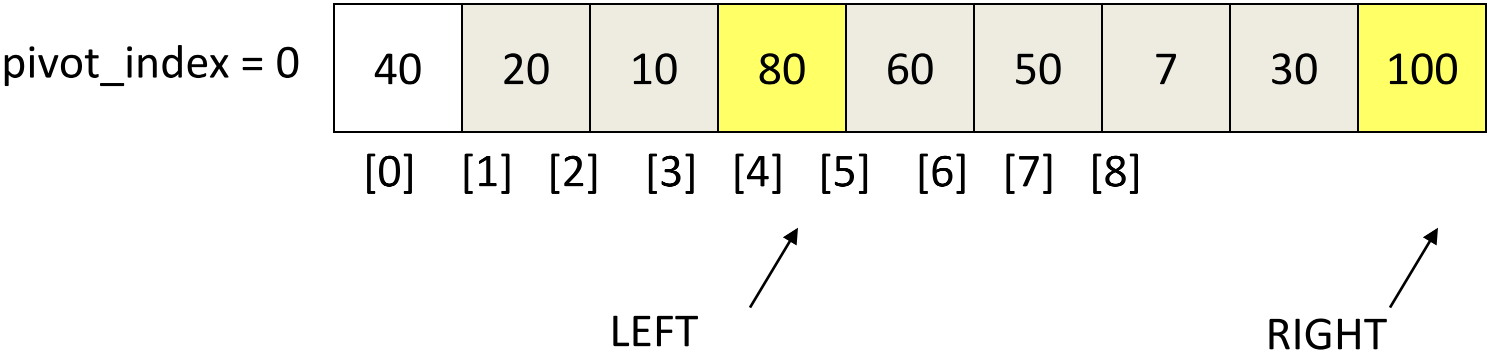
```
1.   While data[LEFT] <= data[pivot]
         ++LEFT
```



pivot_index = 0

40	20	10	80	60	50	7	30	100

```
1. While data[LEFT] <= data[pivot]
       ++LEFT
2. While data[RIGHT] > data[pivot]
       --RIGHT
```



pivot_index = 0

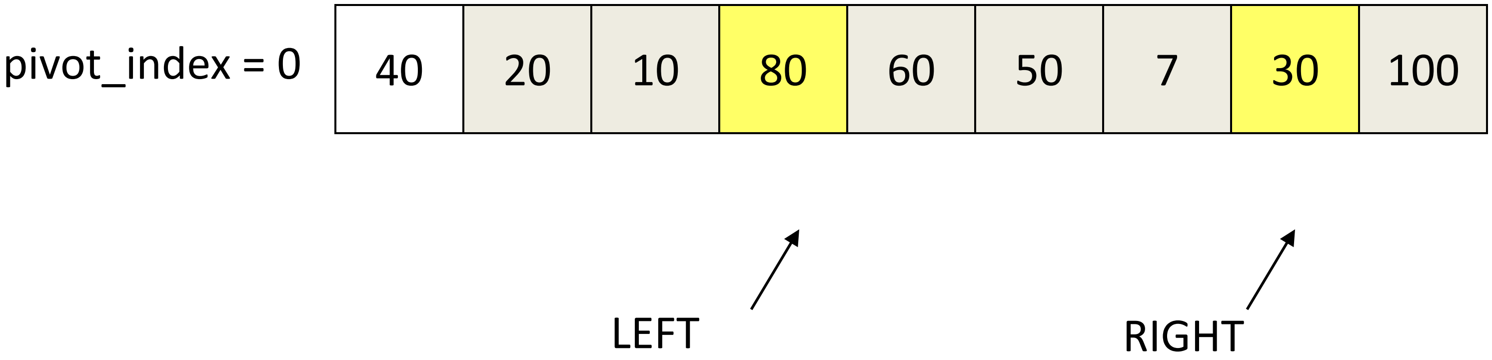
40	20	10	80	60	50	7	30	100

[0] [1] [2] [3] [4] [5] [6] [7] [8]

LEFT

RIGHT

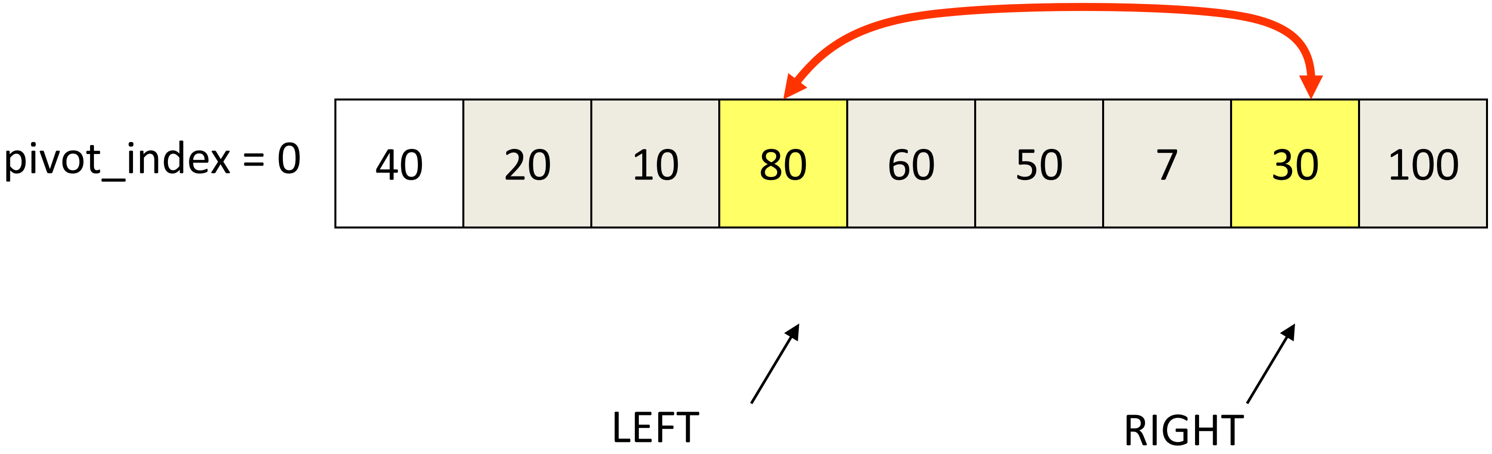
```
1. While data[LEFT] <= data[pivot]
       ++LEFT
2. While data[RIGHT] > data[pivot]
       --RIGHT
```



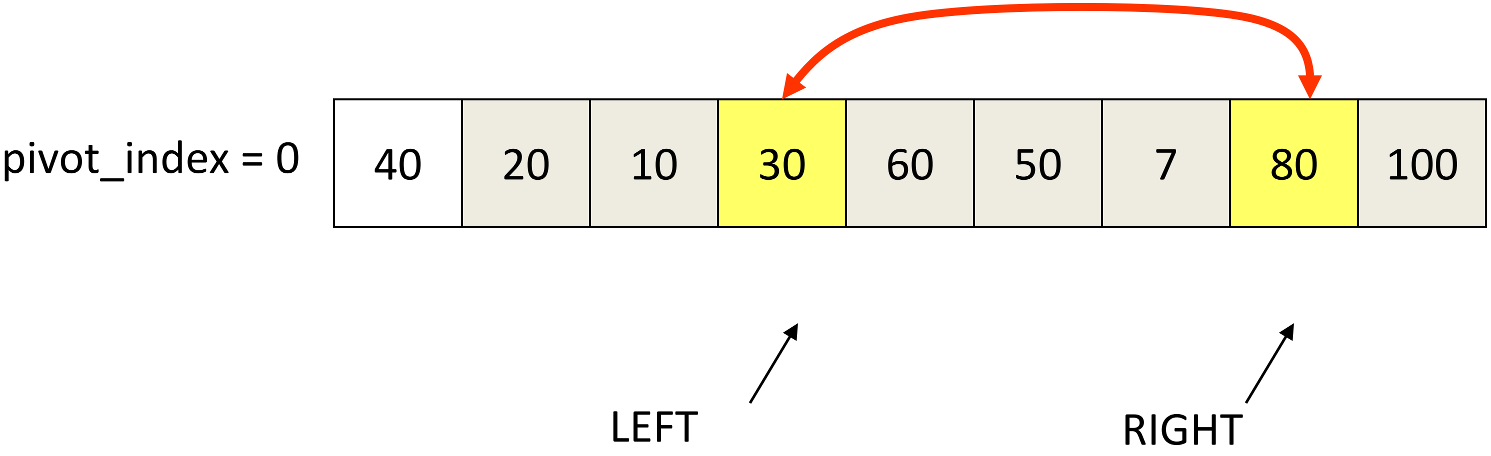
pivot_index = 0

40	20	10	80	60	50	7	30	100

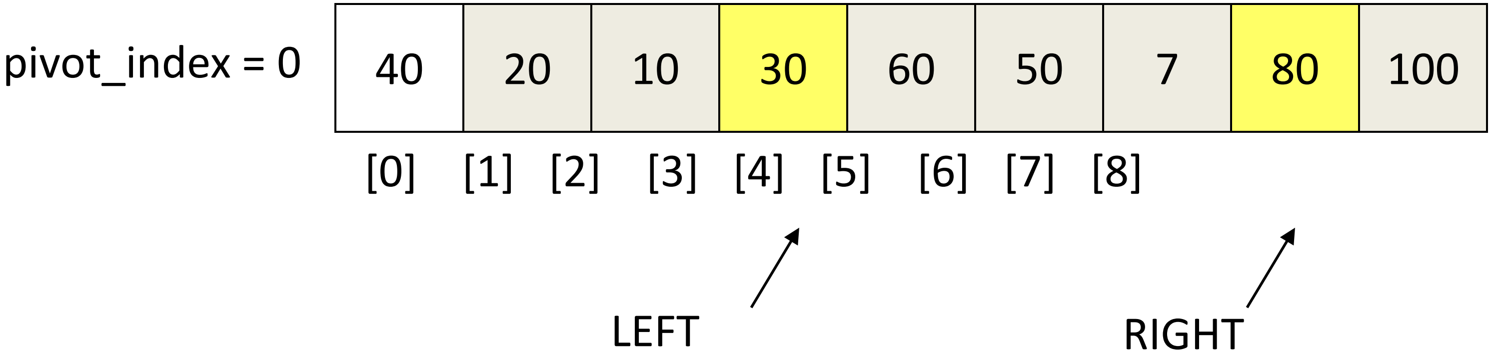
```
1.  While data[LEFT] <= data[pivot]
        ++LEFT
2.  While data[RIGHT] > data[pivot]
        --RIGHT
3.  If LEFT < RIGHT
        swap data[LEFT] and data[RIGHT]
```



1. While data[LEFT] <= data[pivot]
    ++LEFT
2. While data[RIGHT] > data[pivot]
    --RIGHT
3. If LEFT < RIGHT
    swap data[LEFT] and data[RIGHT]



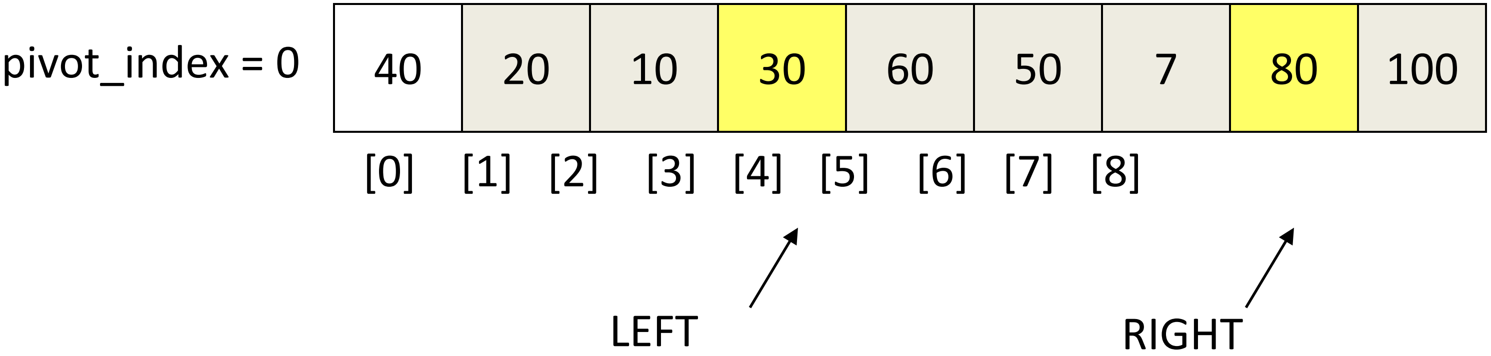
```
1.  While data[LEFT] <= data[pivot]
        ++LEFT
2.  While data[RIGHT] > data[pivot]
        --RIGHT
3.  If LEFT < RIGHT
        swap data[LEFT] and data[RIGHT]
4.  While RIGHT > LEFT, go to 1.
```



pivot_index = 0

40	20	10	30	60	50	7	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

```
1.  While data[LEFT] <= data[pivot]
            ++LEFT
2.  While data[RIGHT] > data[pivot]
            --RIGHT
3.  If LEFT < RIGHT
            swap data[LEFT] and data[RIGHT]
4.  While RIGHT > LEFT, go to 1.
```



pivot_index = 0

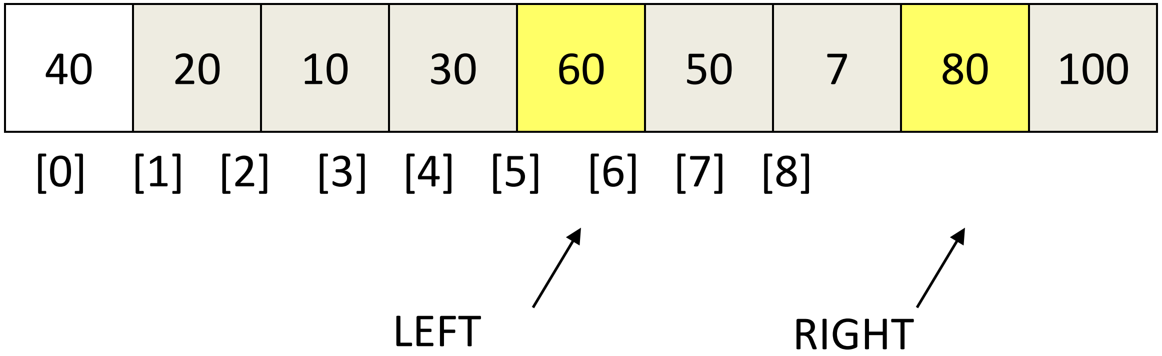
40	20	10	30	60	50	7	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

LEFT

RIGHT

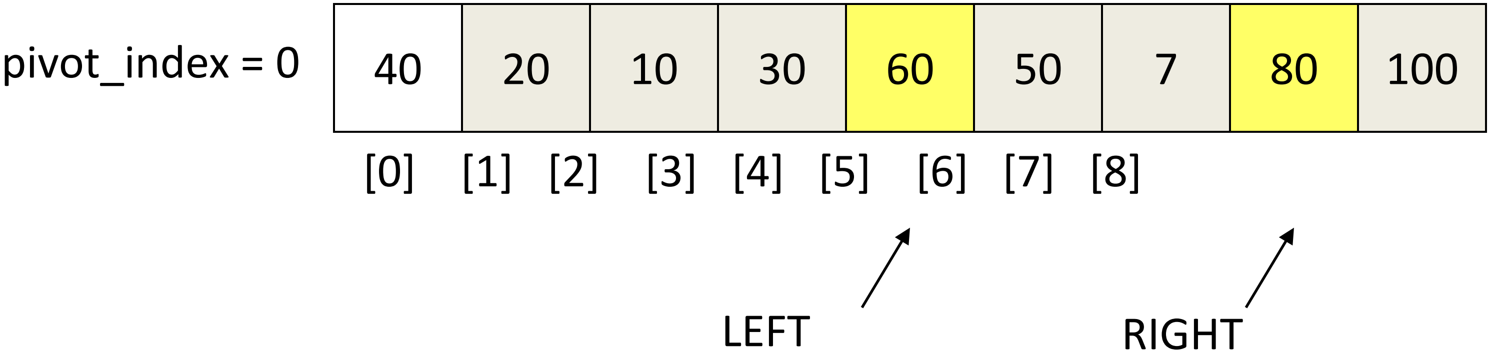
```
→ 1.   While data[LEFT] <= data[pivot]
               ++LEFT
  2.   While data[RIGHT] > data[pivot]
               --RIGHT
  3.   If LEFT < RIGHT
               swap data[LEFT] and data[RIGHT]
  4.   While RIGHT > LEFT, go to 1.
```

pivot_index = 0



40	20	10	30	60	50	7	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

```
   1. While data[LEFT] <= data[pivot]
          ++LEFT
→  2. While data[RIGHT] > data[pivot]
          --RIGHT
   3. If LEFT < RIGHT
          swap data[LEFT] and data[RIGHT]
   4. While RIGHT > LEFT, go to 1.
```

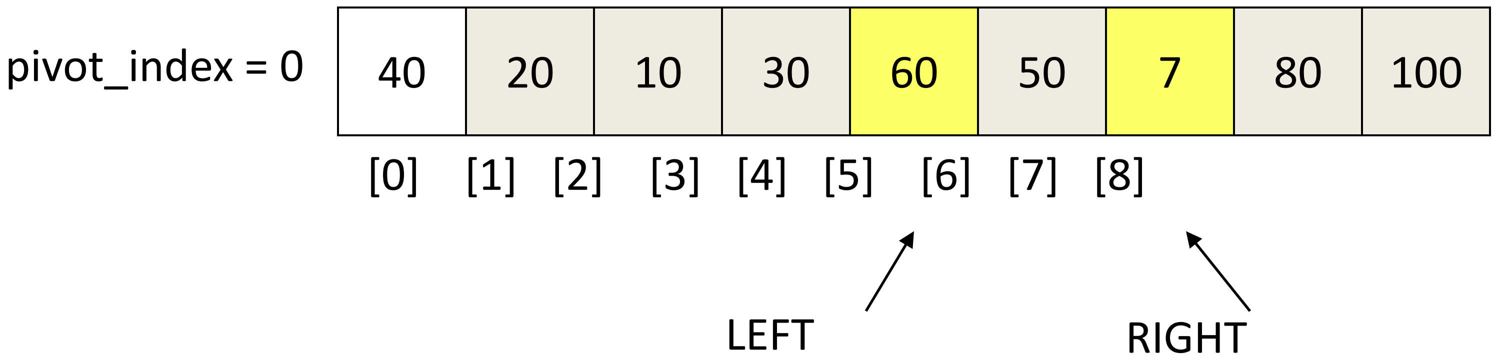


pivot_index = 0

40	20	10	30	60	50	7	80	100

[0] [1] [2] [3] [4] [5] [6] [7] [8]

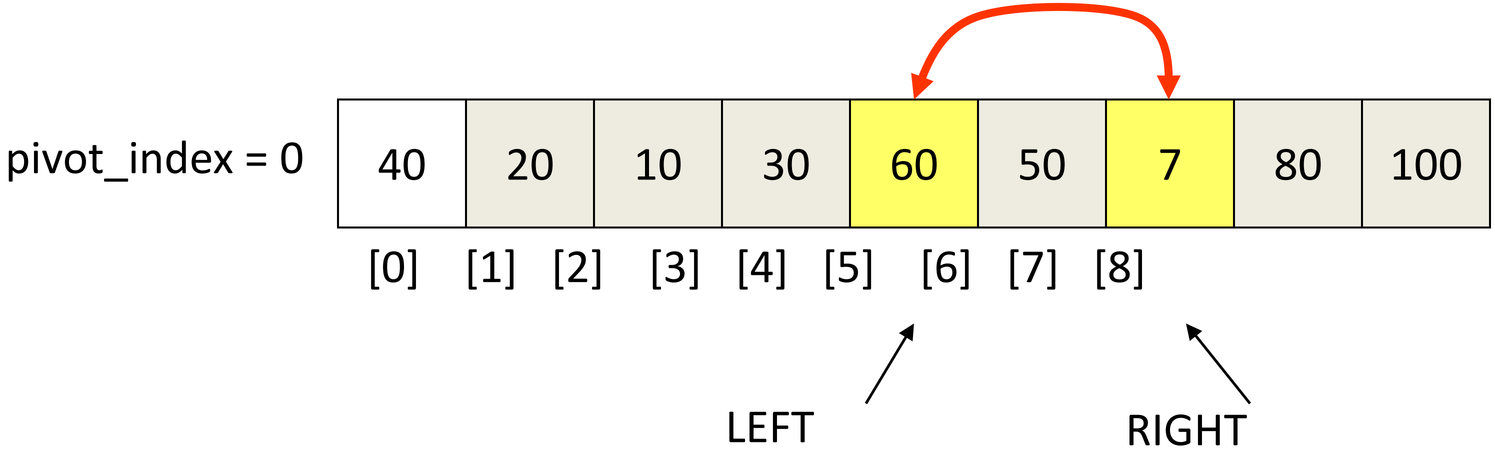
```
   1. While data[LEFT] <= data[pivot]
          ++LEFT
→  2. While data[RIGHT] > data[pivot]
          --RIGHT
   3. If LEFT < RIGHT
          swap data[LEFT] and data[RIGHT]
   4. While RIGHT > LEFT, go to 1.
```



pivot_index = 0

40	20	10	30	60	50	7	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

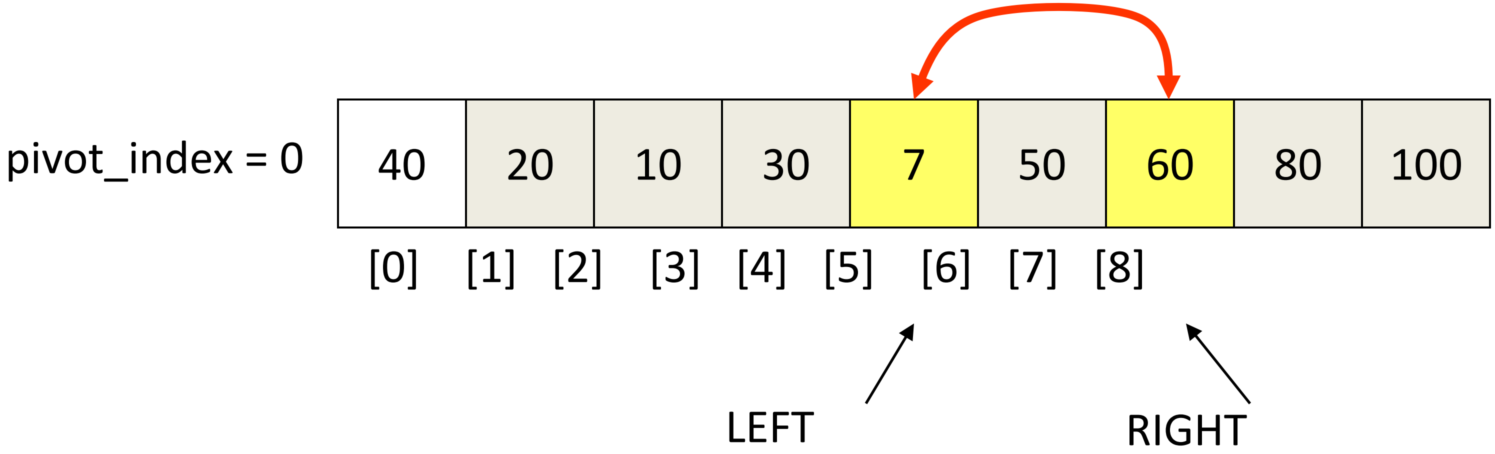
```
1. While data[LEFT] <= data[pivot]
        ++LEFT
2. While data[RIGHT] > data[pivot]
        --RIGHT
→ 3. If LEFT < RIGHT
        swap data[LEFT] and data[RIGHT]
4. While RIGHT > LEFT, go to 1.
```



pivot_index = 0

40	20	10	30	60	50	7	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

```
  1. While data[LEFT] <= data[pivot]
         ++LEFT
  2. While data[RIGHT] > data[pivot]
         --RIGHT
→ 3. If LEFT < RIGHT
         swap data[LEFT] and data[RIGHT]
  4. While RIGHT > LEFT, go to 1.
```

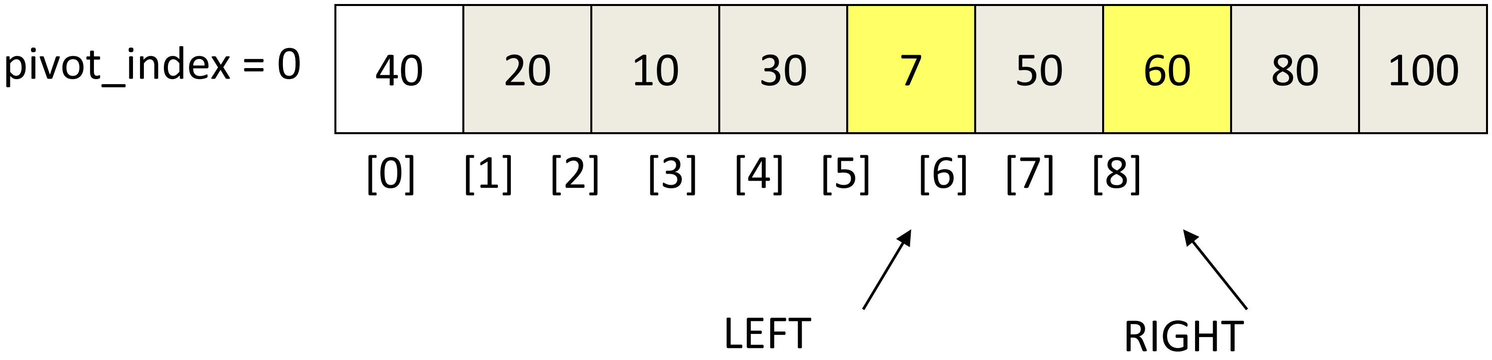


pivot_index = 0

40	20	10	30	7	50	60	80	100

[0] [1] [2] [3] [4] [5] [6] [7] [8]

```
1. While data[LEFT] <= data[pivot]
        ++LEFT
2. While data[RIGHT] > data[pivot]
        --RIGHT
3. If LEFT < RIGHT
        swap data[LEFT] and data[RIGHT]
4. While RIGHT > LEFT, go to 1.
```



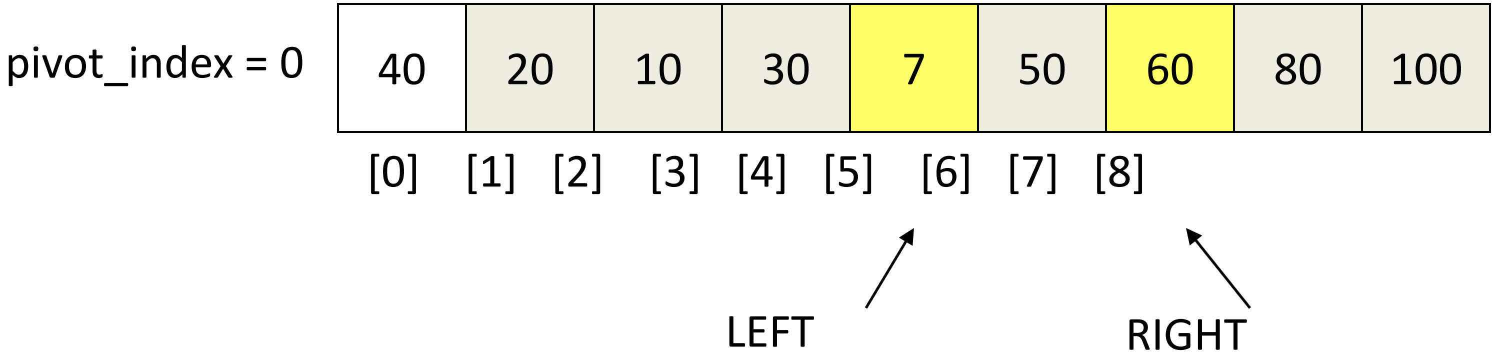
pivot_index = 0

40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

LEFT

RIGHT

```
→ 1. While data[LEFT] <= data[pivot]
        ++LEFT
  2. While data[RIGHT] > data[pivot]
        --RIGHT
  3. If LEFT < RIGHT
        swap data[LEFT] and data[RIGHT]
  4. While RIGHT > LEFT, go to 1.
```



pivot_index = 0

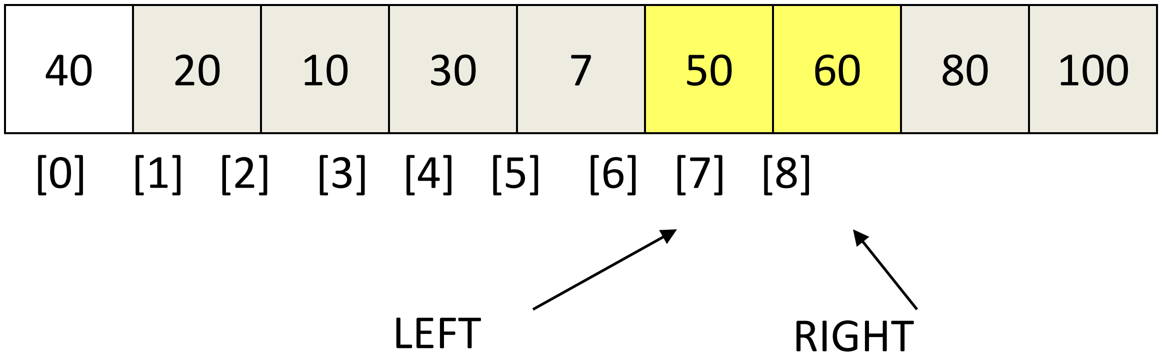
40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

LEFT

RIGHT

```
→ 1. While data[LEFT] <= data[pivot]
             ++LEFT
  2. While data[RIGHT] > data[pivot]
             --RIGHT
  3. If LEFT < RIGHT
             swap data[LEFT] and data[RIGHT]
  4. While RIGHT > LEFT, go to 1.
```

pivot_index = 0



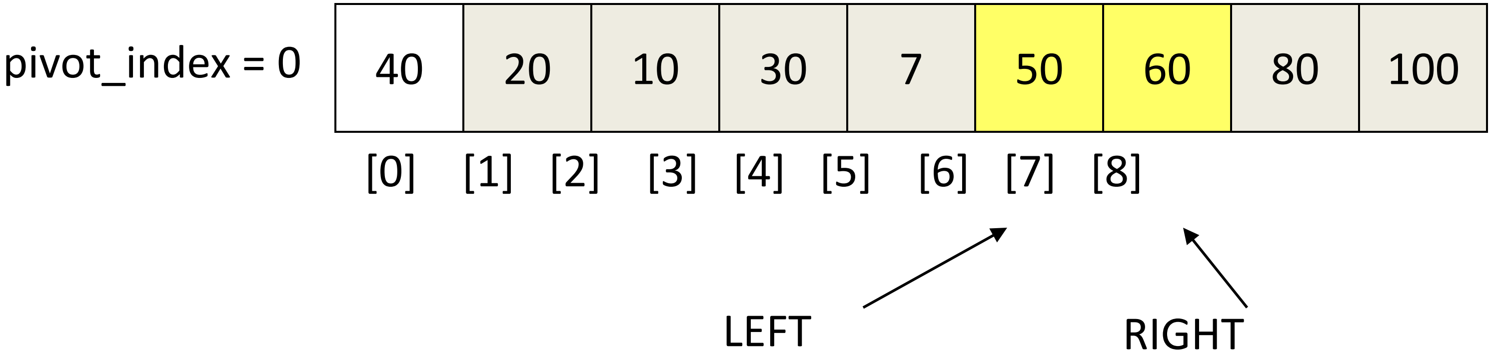
40	20	10	30	7	50	60	80	100

[0] [1] [2] [3] [4] [5] [6] [7] [8]

LEFT

RIGHT

```
   1. While data[LEFT] <= data[pivot]
          ++LEFT
→  2. While data[RIGHT] > data[pivot]
          --RIGHT
   3. If LEFT < RIGHT
          swap data[LEFT] and data[RIGHT]
   4. While RIGHT > LEFT, go to 1.
```



pivot_index = 0

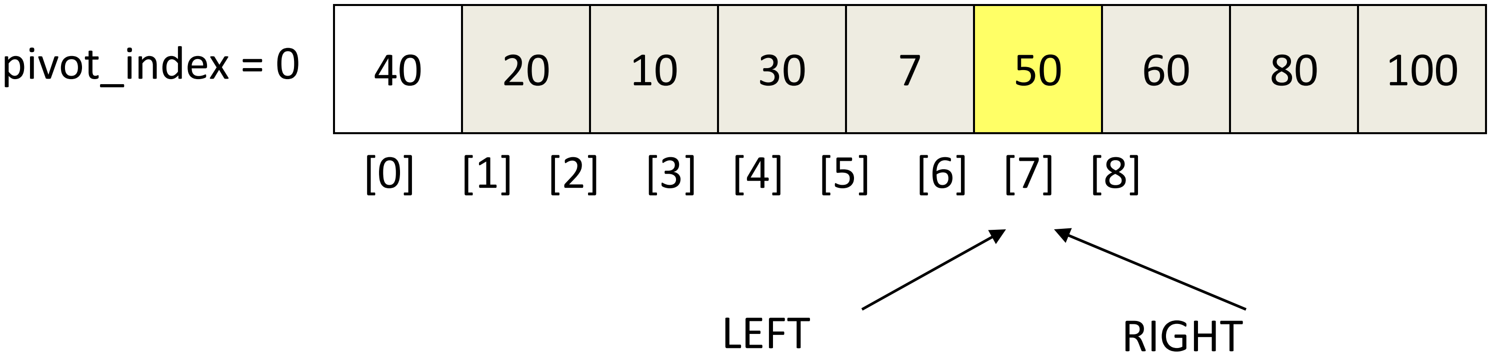
40	20	10	30	7	50	60	80	100

[0] [1] [2] [3] [4] [5] [6] [7] [8]

LEFT

RIGHT

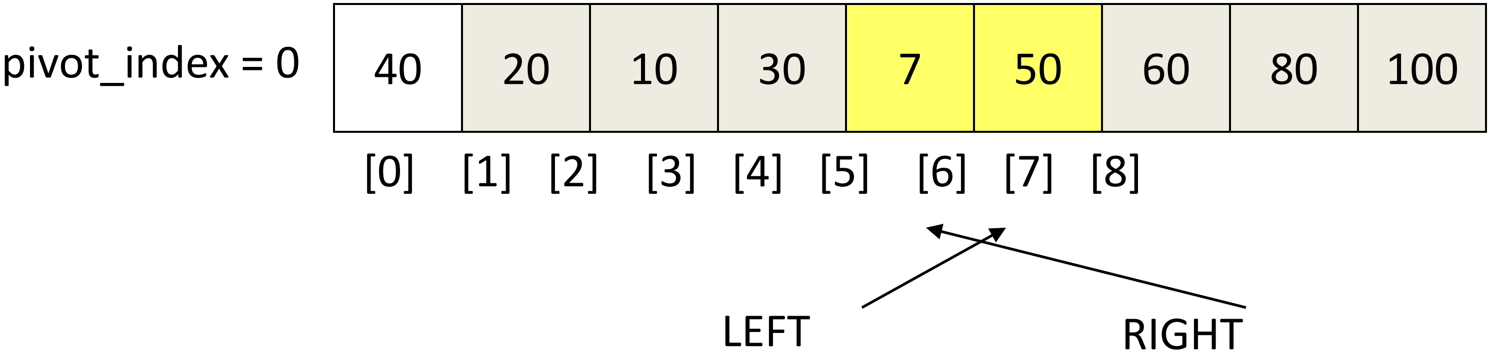
```
1. While data[LEFT] <= data[pivot]
        ++LEFT
→ 2. While data[RIGHT] > data[pivot]
        --RIGHT
3. If LEFT < RIGHT
        swap data[LEFT] and data[RIGHT]
4. While RIGHT > LEFT, go to 1.
```



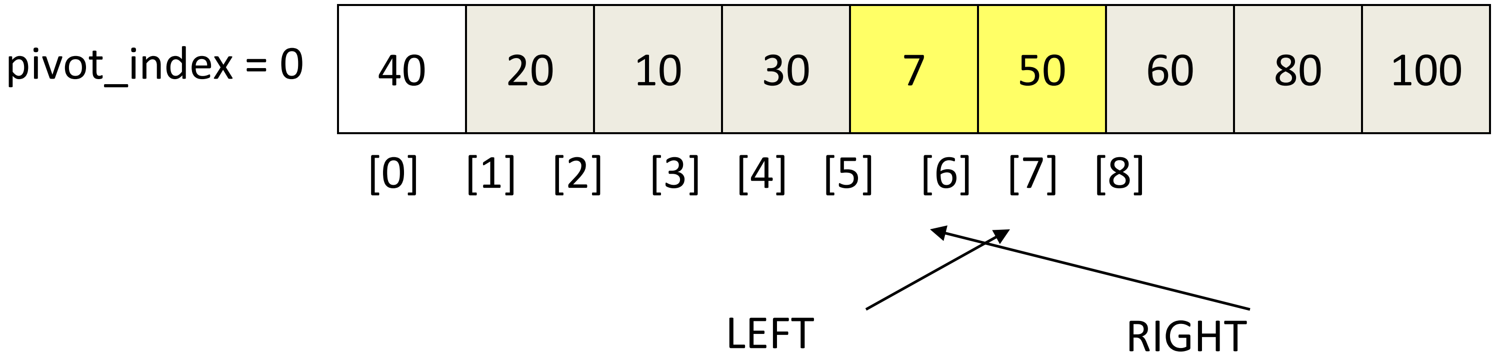
pivot_index = 0

40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

```
   1. While data[LEFT] <= data[pivot]
          ++LEFT
→  2. While data[RIGHT] > data[pivot]
          --RIGHT
   3. If LEFT < RIGHT
          swap data[LEFT] and data[RIGHT]
   4. While RIGHT > LEFT, go to 1.
```



```
   1. While data[LEFT] <= data[pivot]
          ++LEFT
   2. While data[RIGHT] > data[pivot]
          --RIGHT
→  3. If LEFT < RIGHT
          swap data[LEFT] and data[RIGHT]
   4. While RIGHT > LEFT, go to 1.
```

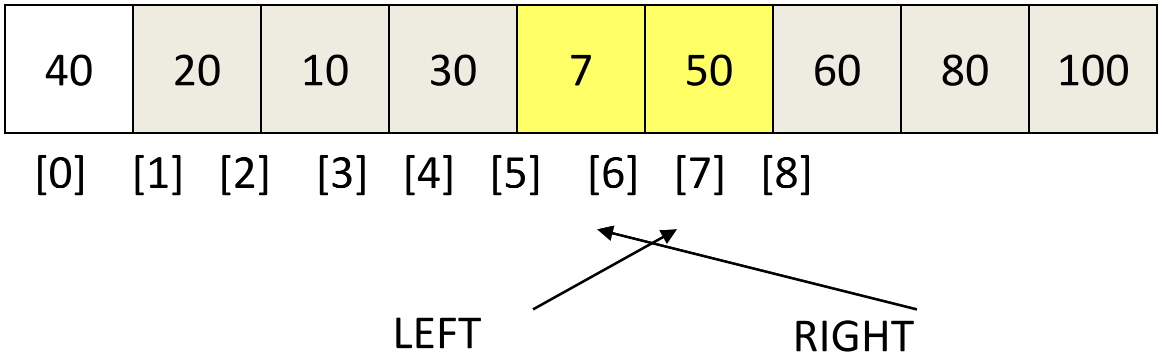


pivot_index = 0

40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

```
   1. While data[LEFT] <= data[pivot]
          ++LEFT
   2. While data[RIGHT] > data[pivot]
          --RIGHT
   3. If LEFT < RIGHT
          swap data[LEFT] and data[RIGHT]
→  4. While RIGHT > LEFT, go to 1.
```

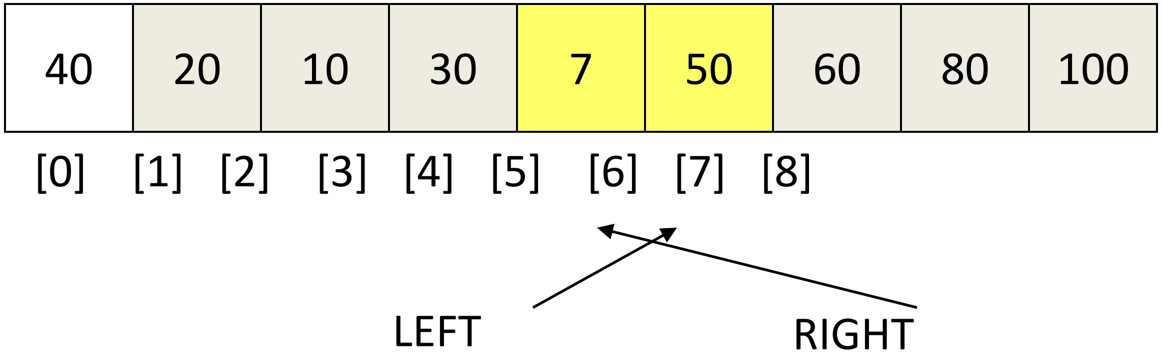
pivot_index = 0



40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

```
   1. While data[LEFT] <= data[pivot]
          ++LEFT
   2. While data[RIGHT] > data[pivot]
          --RIGHT
   3. If LEFT < RIGHT
          swap data[LEFT] and data[RIGHT]
   4. While RIGHT > LEFT, go to 1.
→  5. Swap data[RIGHT] and data[pivot_index]
```

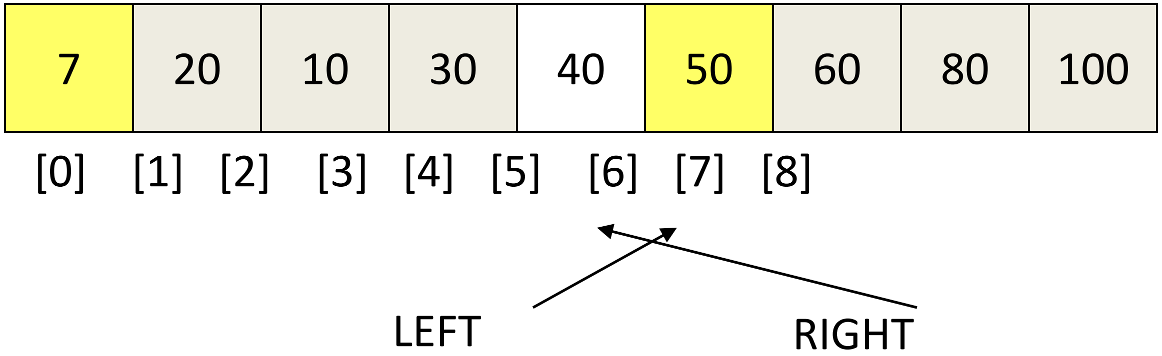
pivot_index = 0



40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

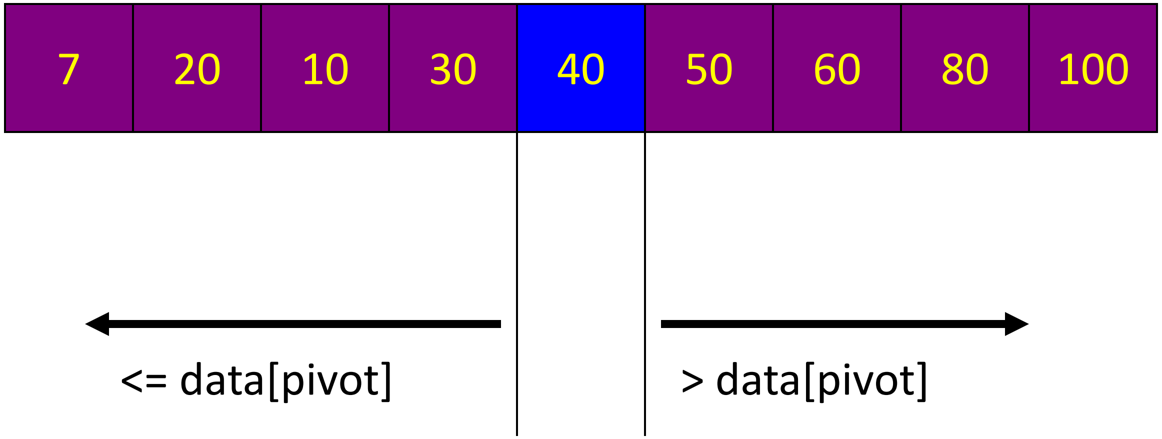
```
1. While data[LEFT] <= data[pivot]
       ++LEFT
2. While data[RIGHT] > data[pivot]
       --RIGHT
3. If LEFT < RIGHT
       swap data[LEFT] and data[RIGHT]
4. While RIGHT > LEFT, go to 1.
5. Swap data[RIGHT] and data[pivot_index]
```

pivot_index = 4



7	20	10	30	40	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

Partition Result

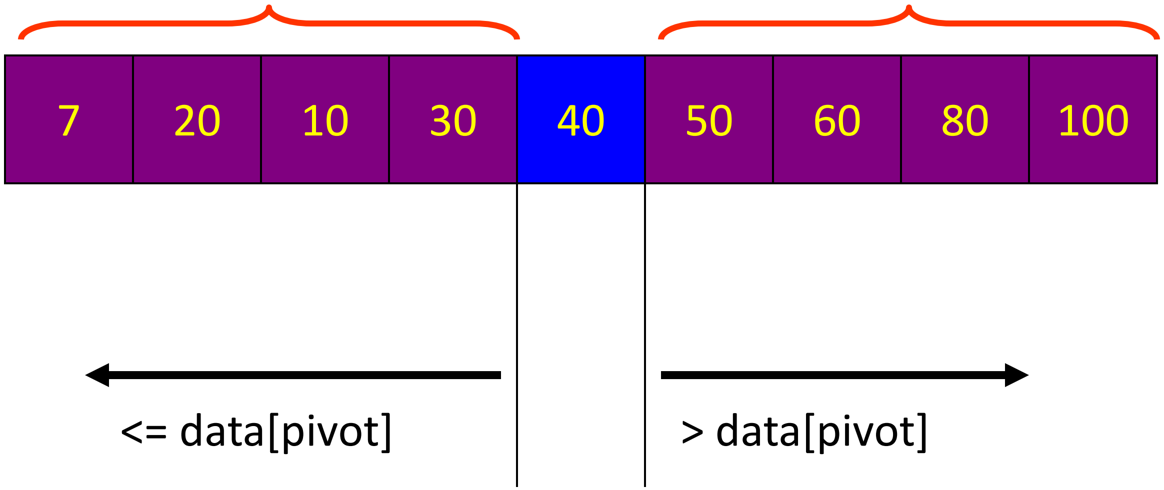


7	20	10	30	40	50	60	80	100

<= data[pivot]

> data[pivot]

Recursion: Quicksort Sub-arrays



7	20	10	30	40	50	60	80	100

<= data[pivot]

> data[pivot]

Two Critical steps

- Picking up the right value for Pivot
- Partitioning

Picking up the Pivot

- Use the first element as pivot
  - if the input is random, ok
  - if the input is presorted (or in reverse order)
    - all the elements go into S2 (or S1)
    - this happens consistently throughout the recursive calls
    - Results in $O(n^2)$ behavior (Analyze this case later)
- Choose the pivot randomly
  - generally safe
  - random number generation can be expensive

A better Pivot

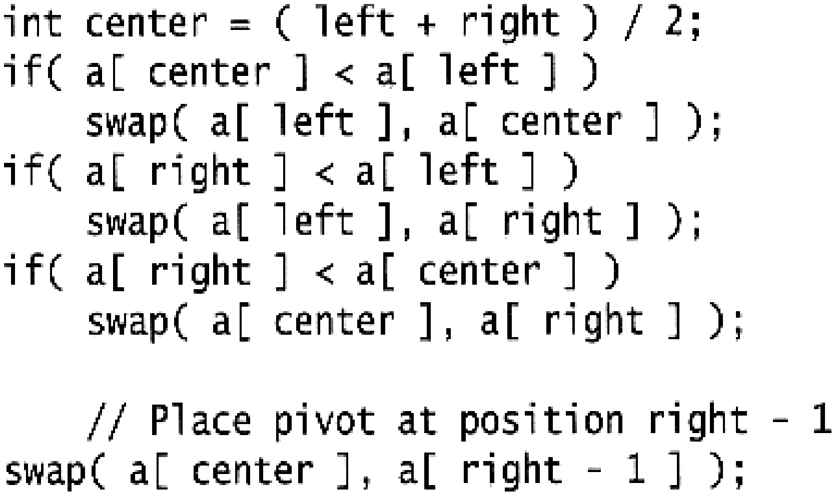
Use the median of the array

- Partitioning always cuts the array into roughly half
- An optimal quicksort (O(N log N))
- However, hard to find the exact median (chicken-egg?)
  - e.g., sort an array to pick the value in the middle
- Approximation to the exact median: …

Median of three

- We will use median of three
  - Compare just three elements: the leftmost, rightmost and center
  - Swap these elements if necessary so that
    - A[left] = Smallest
    - A[right] = Largest
    - A[center] = Median of three
  - Pick A[center] as the pivot
  - Swap A[center] and A[right – 1] so that pivot is at second last position (why?)

median3



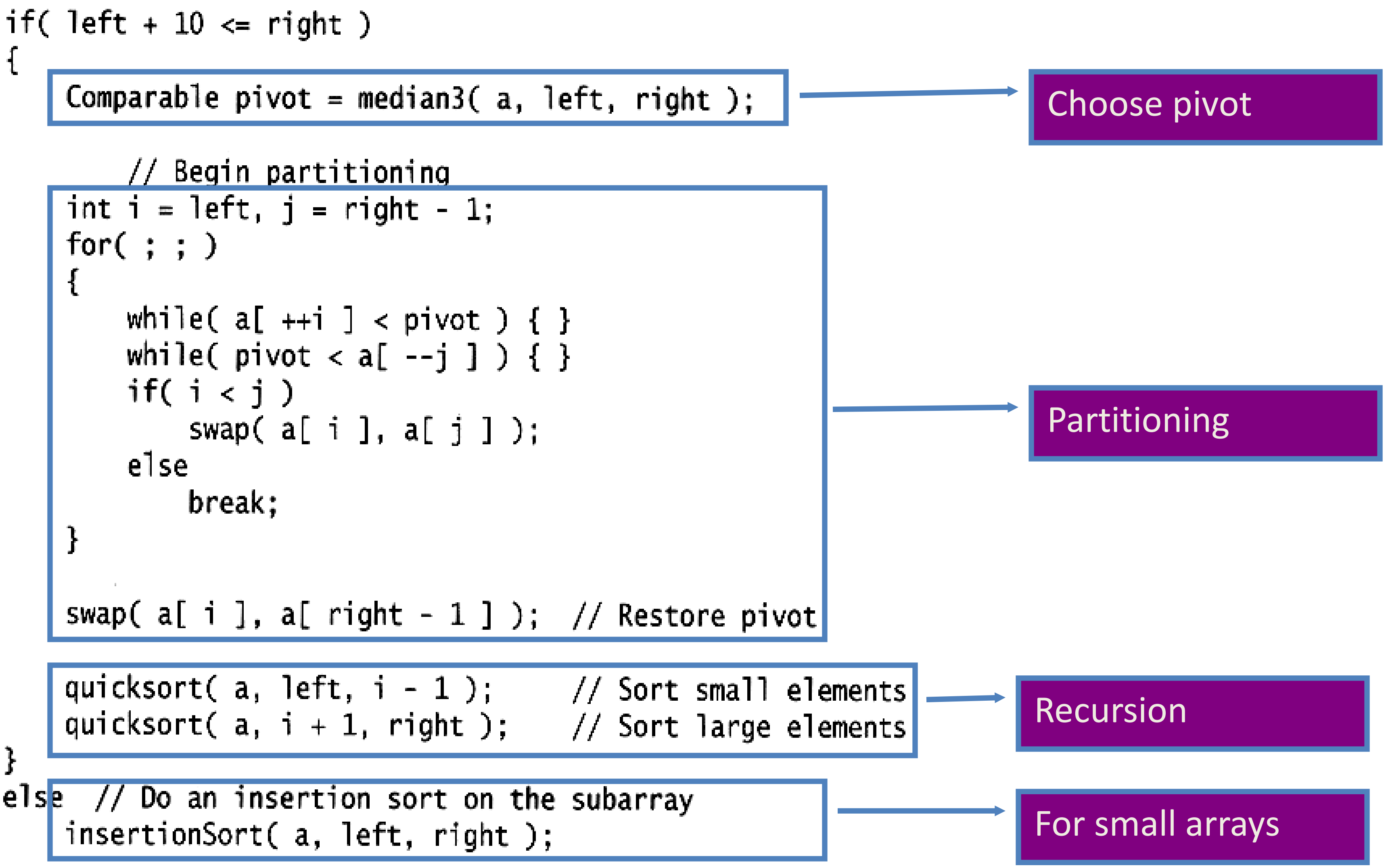
```
int center = ( left + right ) / 2;
if( a[ center ] < a[ left ] )
    swap( a[ left ], a[ center ] );
if( a[ right ] < a[ left ] )
    swap( a[ left ], a[ right ] );
if( a[ right ] < a[ center ] )
    swap( a[ center ], a[ right ] );

    // Place pivot at position right - 1
swap( a[ center ], a[ right - 1 ] );
```

In-Place Partition

- If use additional array (not in-place) like MergeSort
  - Straightforward to code like MergeSort (write it down!)
  - Inefficient!

- Many ways to implement
- Even the slightest deviations may cause surprisingly bad results.
  - Not stable as it does not preserve the ordering of the identical keys.
  - Hard to write correctly ☹

A practical implementation



```
if( left + 10 <= right )
{
    Comparable pivot = median3( a, left, right );

        // Begin partitioning
    int i = left, j = right - 1;
    for( ; ; )
    {
        while( a[ ++i ] < pivot ) { }
        while( pivot < a[ --j ] ) { }
        if( i < j )
            swap( a[ i ], a[ j ] );
        else
            break;
    }

    swap( a[ i ], a[ right - 1 ] );  // Restore pivot

    quicksort( a, left, i - 1 );     // Sort small elements
    quicksort( a, i + 1, right );    // Sort large elements
}
else  // Do an insertion sort on the subarray
    insertionSort( a, left, right );
```

Choose pivot

Partitioning

Recursion

For small arrays

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$!!!
- What can we do to avoid worst case?

Improved Pivot Selection

Pick median value of three elements from data array:

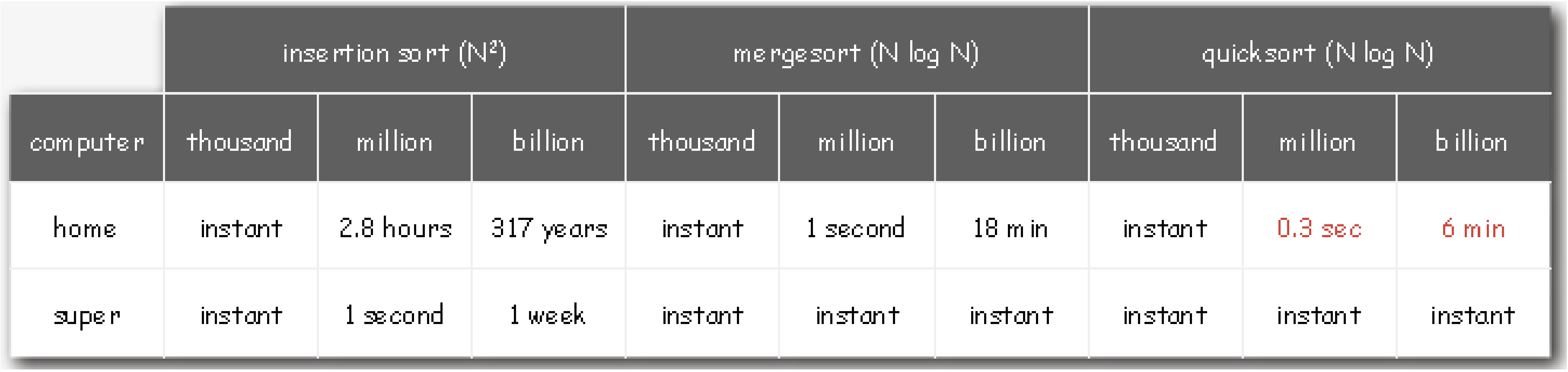
data[0], data[n/2], and data[n-1].

Use this median value as pivot.

Performance

- Quicksort turns out to be the fastest sorting algorithm in practice. It has a time complexity of $\Theta(n \log(n))$ on the average. However, in the (very rare) worst case quicksort is as slow as Bubble-sort, namely in $\Theta(n^2)$. There are sorting algorithms with a time complexity of $O(n \log(n))$ even in the worst case,
  - e.g. Heapsort and Mergesort. But on the average, these algorithms are by a constant factor slower than quicksort.

Performance



	insertion sort (N^2)			mergesort (N log N)			quicksort (N log N)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.3 sec	6 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

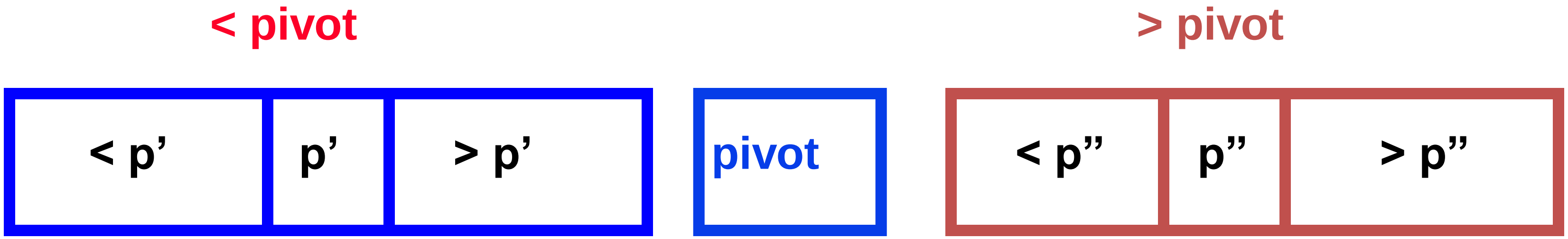
Quicksort – Summary

- Partition
  - Choose a pivot
  - Find the position for the pivot so that
    - all elements to the left are less
    - all elements to the right are greater

< pivot	**pivot**	**> pivot**

Quicksort

- Conquer
  - Apply the same algorithm to each half



Quicksort

Implementation

```
quicksort( void *a, int low, int high )
   {
   int pivot;
   /* Termination condition! */
   if ( high > low )
     {
      pivot = partition( a, low, high );
      quicksort( a, low, pivot-1 );
      quicksort( a, pivot+1, high );
     }
   }
```

Divide

Conquer

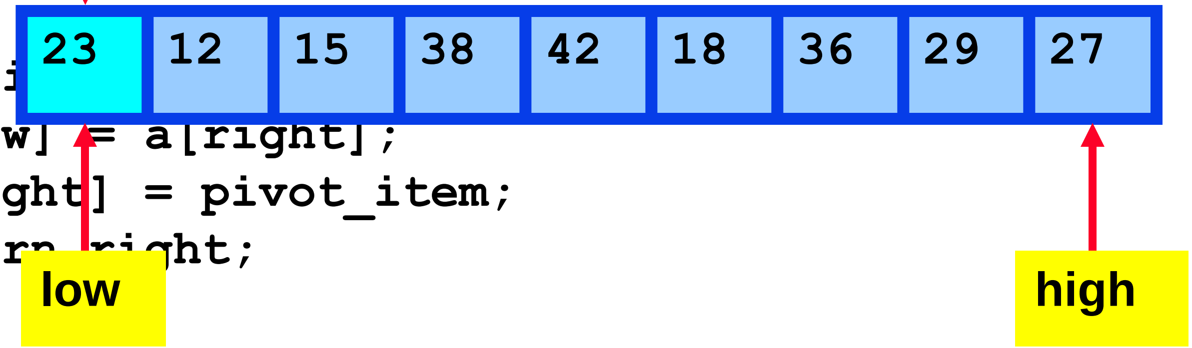
Quicksort - Partition

```
int partition(int *a, int low, int high) {
   int left, right;
   int pivot_item;
   pivot_item = a[low];
   pivot = left = low;
   right = high;
   while ( left < right ) {
     /* Move left while item < pivot */
     while( a[left] <= pivot_item ) left++;
     /* Move right while item > pivot */
     while( a[right] >= pivot_item ) right--;
     if ( left < right ) SWAP(a,left,right);
     }
   /* right is final position for the pivot */
   a[low] = a[right];
   a[right] = pivot_item;
   return right;
   }
```

Quicksort - Partition

```
int partition( int *a, int low, int high ) {
   int left, right;
   int pivot_item;
   pivot_item = a[low];
   pivot = left = low;
   right = high;
   while ( left < right
     /* Move left while
     while( a[left] <= pivot_item ) left++;
     /* Move right while item > pivot */
     while( a[right] >= pivot_item ) right--;
     if ( left < right ) SWAP(a,left,right);
     }
   /* ri
   a[low] = a[right];
   a[right] = pivot_item;
   return right;
   }
```

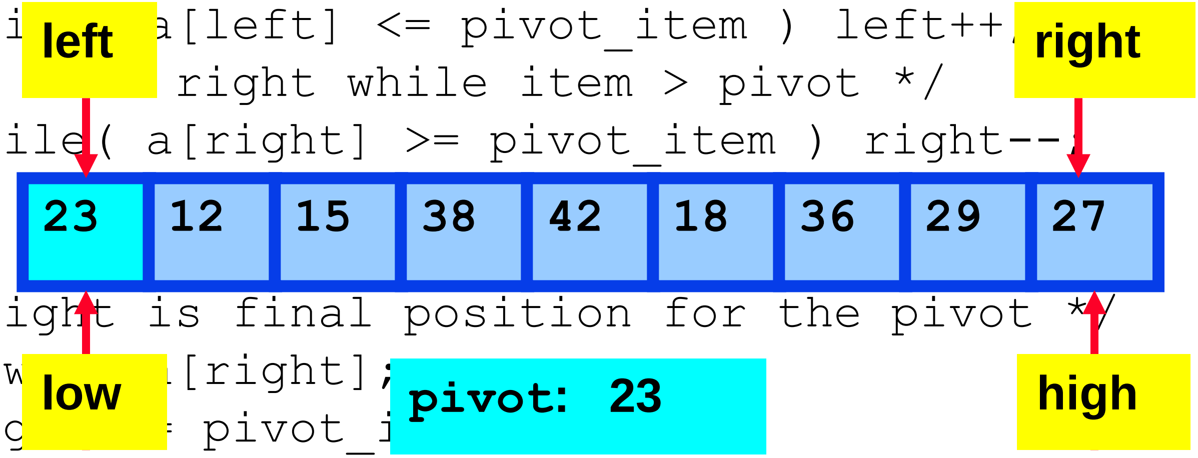
Any item will do as the pivot, choose the leftmost one!



Quicksort - Partition

```
int partition( int *a, int low, int high ) {
    int left, right;
    int pivot_item;
    pivot_item = a[low];
    pivot = left = low;
    right = high;
    while ( left < right ) {
      /* Move left while item < pivot */
      whi      a[left] <= pivot_item ) left++
      /*       right while item > pivot */
      while( a[right] >= pivot_item ) right--
      if
      }
    /* right is final position for the pivot */
    a[low      [right];
    a[rig      = pivot_i
    return right;
    }
```

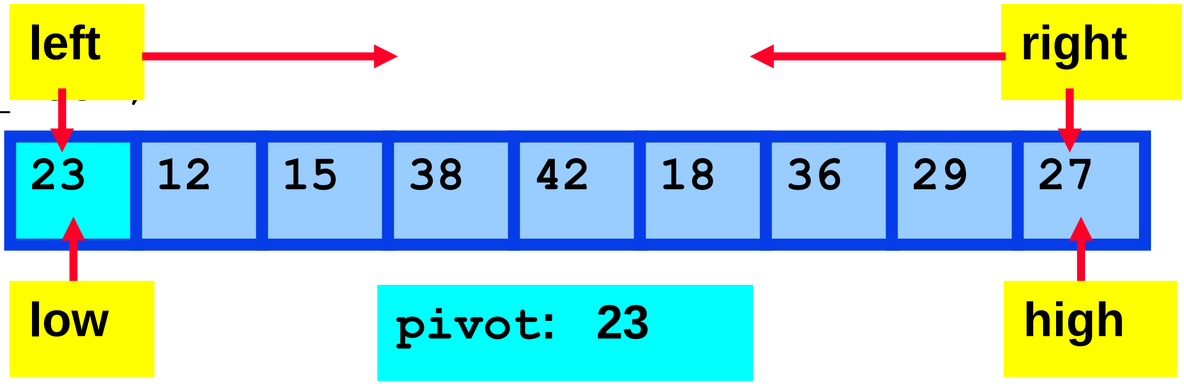
Set left and right markers



Quicksort - Partition

```
int partition( int *a, int low, int high ) {
    int left, right;
    int pivot_item;
    pivot_item = a[low];
    pivot = left = low;
    right = high;
    while ( left < right ) {
      /* Move left while item < pivot */
      while( a[left] <= pivot_item ) left++;
      /* Move right while item > pivot */
      while( a[right] >= pivot_item ) right--;
      if ( left < right ) SWAP(a,left,right);
      }
    /* right is final position for the pivot */
    a[low] = a[right]
    a[right] = pivot_
    return right;
    }
```

Move the markers until they cross over

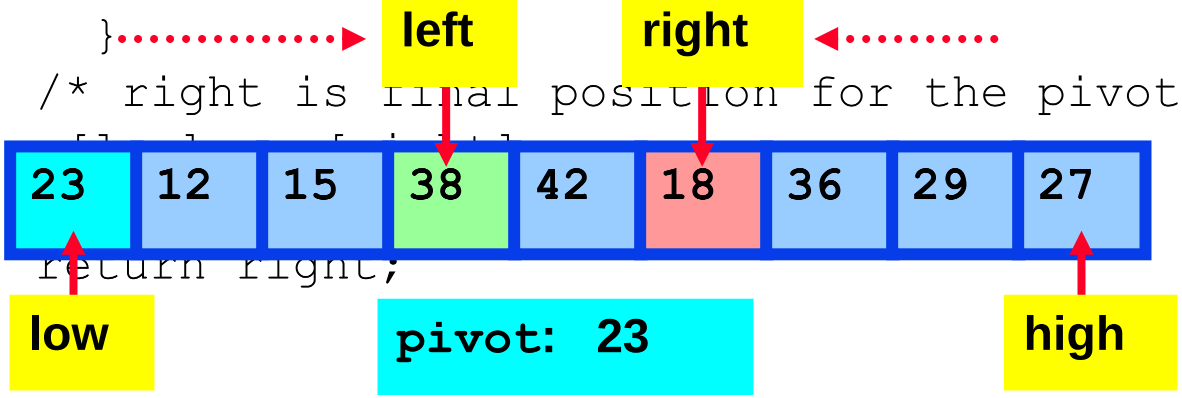


Quicksort - Partition

```
int partition( int *a, int low, int high ) {
    int left, right;
    int pivot_item;
    pivot_item = a[low];
    pivot = left = low;
    right = high;
    while ( left < right ) {
      /* Move left while iten
      while( a[left] <= pivot_item ) left++;
      /* Move right while item > pivot */
      while( a[right] >= pivot_item ) right--;
      if ( left < right ) SWAP(a,left,right);
      }
    /* right is final position for the pivot

    return right;
```

Move the left pointer while it points to items <= pivot

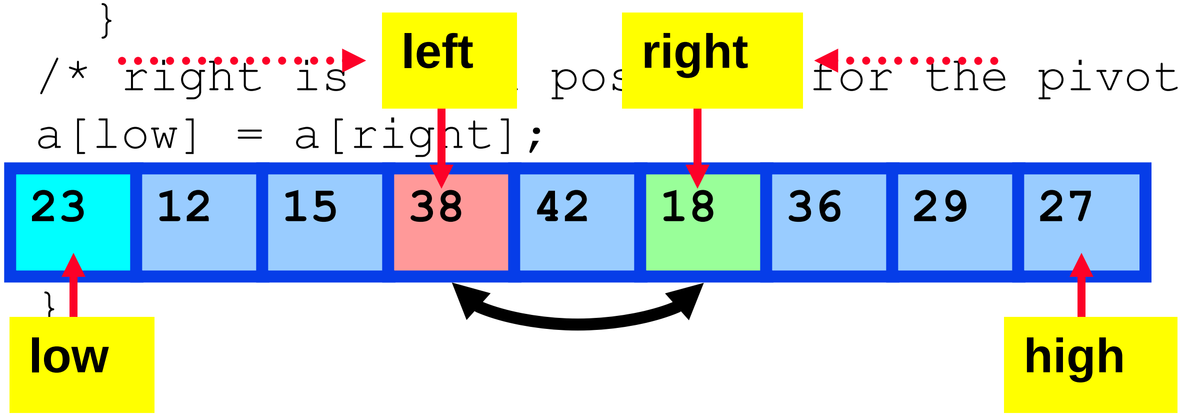
Move right similarly



Quicksort - Partition

```
int partition( int *a, int low, int high ) {
    int left, right;
    int pivot_item;
    pivot_item = a[low];
    pivot = left = low;
    right = high;
    while ( left < right )
      /* Move left while item < pivot */
      while( a[left] <= pivot_item ) left++;
      /* Move right while item > pivot */
      while( a[right] >= pivot_item ) right--;
      if ( left < right ) SWAP(a,left,right);
      }
    /* right is          pos            for the pivot */
    a[low] = a[right];
    }
```

Swap the two items
on the wrong side of the pivot

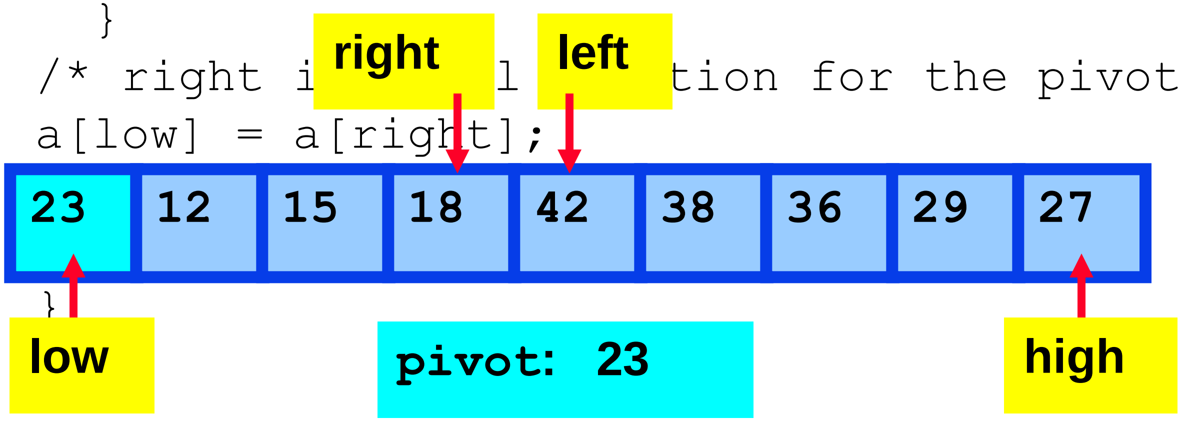


pivot: 23

Quicksort - Partition

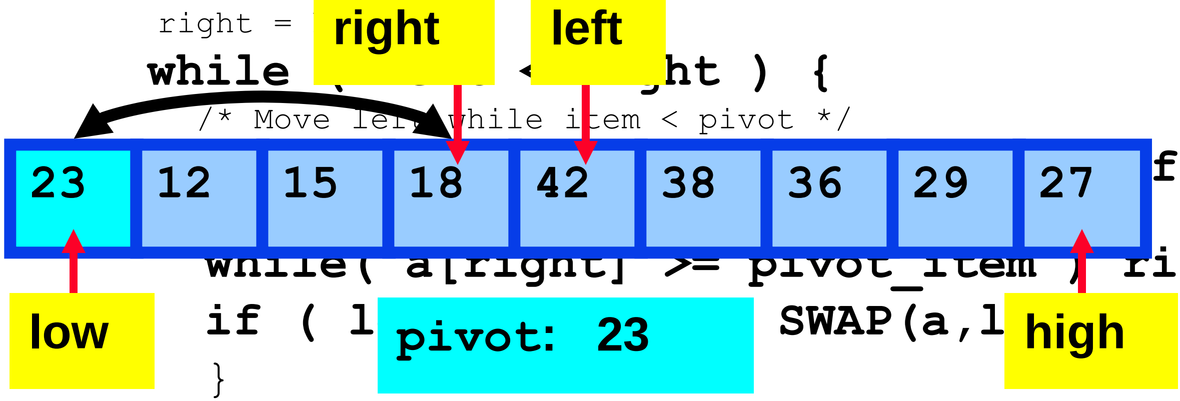
```
int partition( int *a, int low, int high ) {
     int left, right;
      int pivot_item;
      pivot_item = a[low];
      pivot = left = low;
      right = high;
     while ( left < right ) {
        /* Move left while item < pivot */
        while( a[left] <= pivot_item ) left++;
         /* Move right while item > pivot */
        while( a[right] >= pivot_item ) right--;
        if ( left < right ) SWAP(a,left,right);
        }
     /* right i      l      tion for the pivot */
     a[low] = a[right];
     }
```

left and **right** have swapped over, so stop



Quicksort - Partition

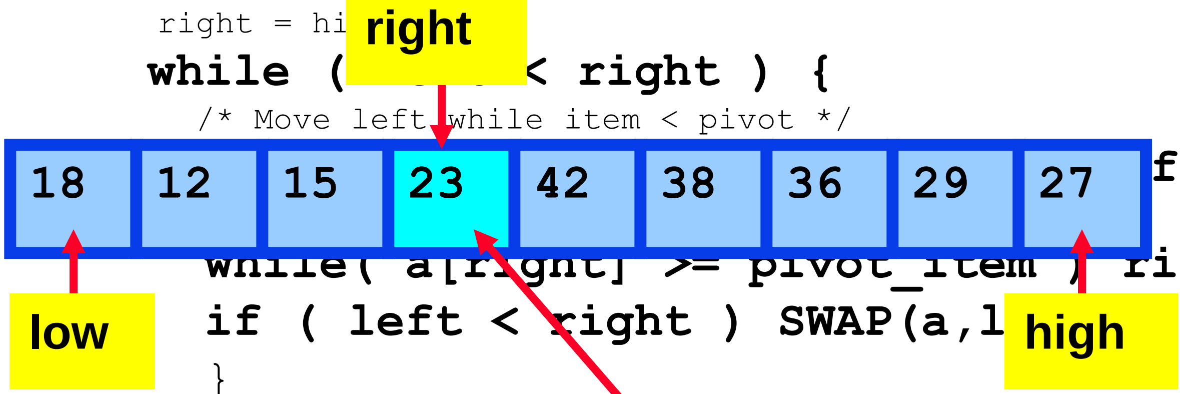
```
int partition( int *a, int low, int high ) {
    int left, right;
    int pivot_item;
    pivot_item = a[low];
    pivot = left = low;
    right =
    while (        <    ght ) {
        /* Move le   while item < pivot */
                                                   ft++;
        while( a[right] >= pivot_item ) right--;
        if ( l                   SWAP(a,l        ght);
        }
    /* right is final position for the pivot */
    a[low] = a[right];
    a[right] = pivot_item;
    return right;
    }
```



Finally, swap the pivot and `right`

Quicksort - Partition

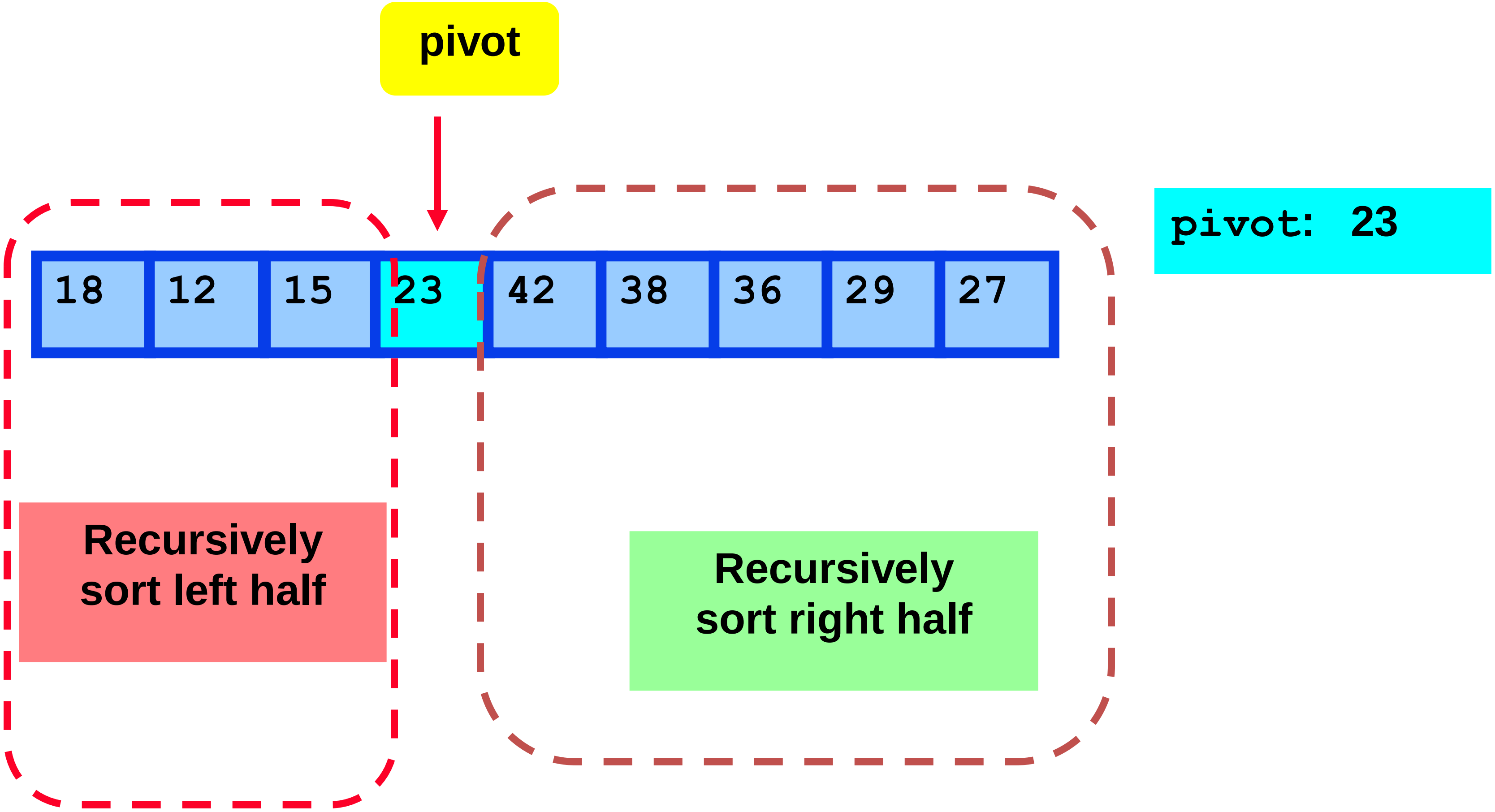
```
int partition( int *a, int low, int high ) {
      int left, right;
       int pivot_item;
       pivot_item = a[low];
       pivot = left = low;
       right = hi
      while (      < right ) {
         /* Move left while item < pivot */
                                            ft++;

         while( a[right] >= pivot_item ) right--;
         if ( left < right ) SWAP(a,l           ght);
         }
      /* right is final position for the pivot */
      a[low] = a[right]
      a[right] = pivot_
      return right;
      }
```



pivot: 23

Return the position of the pivot

Quicksort - Conquer



pivot

18	12	15	23	42	38	36	29	27

`pivot:  23`

Recursively sort left half

Recursively sort right half

Quicksort - Analysis

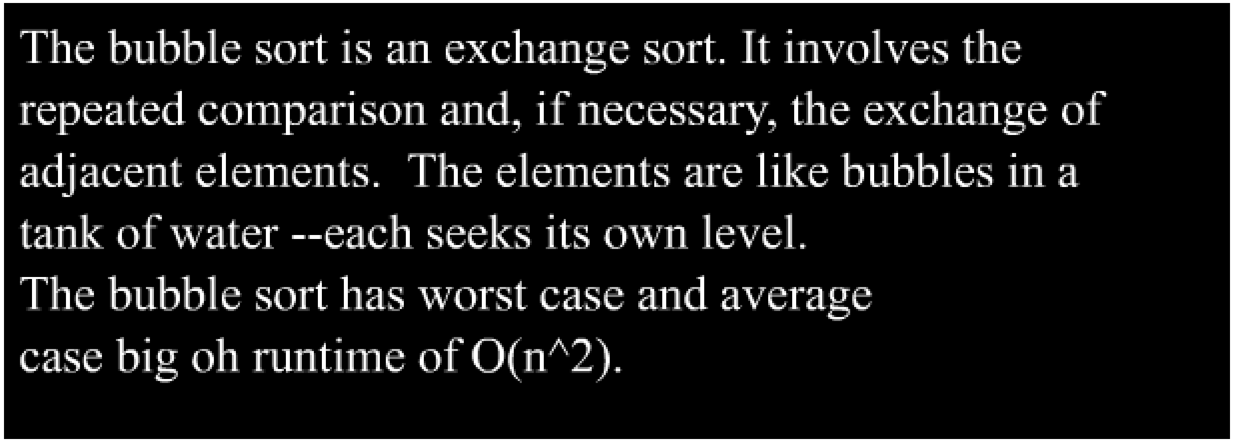
- Partition
  - Check every item once $O(n)$
- Conquer
  - Divide data in half $O(\log_2 n)$
- Total
  - Product $O(n \log n)$
- *Same as Heapsort*
  - quicksort is *generally* faster
    - Fewer comparisons
    - *Details later (and assignment 2!)*
- *But there's a catch …………….*

Quicksort - The truth!

- What happens if we use quicksort on data that's already sorted *(or nearly sorted)*
- *We'd certainly expect it to perform well!*

BUBBLE SORT

Bubble



The bubble sort is an exchange sort. It involves the repeated comparison and, if necessary, the exchange of adjacent elements. The elements are like bubbles in a tank of water --each seeks its own level.
The bubble sort has worst case and average case big oh runtime of O(n^2).

Bubble sort

- Works by repeatedly stepping through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.

Sort: 34 8 64 51 32 21

Pass 1	Pass 2
34 8 64 51 32 21	8 34 51 32 21 64
8 34 64 51 32 21	8 34 51 32 21 64
8 34 51 64 32 21	8 34 51 32 21 64
8 34 51 32 64 21	8 34 32 51 21 64
8 34 51 32 21 64	8 34 32 21 51 64
	8 34 32 21 51 64

Repeat until no swaps are made.

Worst and average - O(n^2)

Not practical for list with large n - except when list is very close to sorted

Algo

- In bubble sort, the larger bubbles (higher values) bubble up displacing the smaller bubbles (lower values)

```
procedure bubbleSort(A : list of sortable items)
  repeat
    swapped = false
    for i = 1 to length(A) - 1 {
      if A[i-1] > A[i] {
        swap(A[i-1], A[i])
        swapped = true
      }
    }
  until not swapped      /*repeat until no element has been swapped*/
end procedure
```

Algo - optimized

- Can be optimized further
  - Final element can be ignored in each iteration
    - Nth pass finds the largest element and puts in place, so need not be compared
  - All elements after the last swap have been sorted

Selection Sort

- works by selecting the smallest (or largest) element of the array and placing it at the head of the array
- Then the process is repeated for the remainder of the array; the next largest element is selected and put into the next slot, and so on down the line

Selection Sort

- Repeatedly searches for the largest value in a section of the data
  - Moves that value into its correct position in a sorted section of the list
- Uses the Find Largest algorithm

Selection Sort

```
1. Get values for n and the n list items
2. Set the marker for the unsorted section at the end of the list
3. While the unsorted section of the list is not empty, do steps 4 through 6
4.      Select the largest number in the unsorted section of the list
5.      Exchange this number with the last number in the unsorted
        section of the list
6.      Move the marker for the unsorted section left one position
7. Stop
```

Selection Sort

- Count comparisons of *largest so far* against other values
- Find Largest, given m values, does m-1 comparisons
- Selection sort calls Find Largest n times,
  - Each time with a smaller list of values
  - Cost = $n-1 + (n-2) + \ldots + 2 + 1 = n(n-1)/2$

Selection Sort

- Time efficiency
  - Comparisons: $n(n-1)/2$
  - Exchanges: n (swapping largest into place)
  - Overall: $\Theta(n^2)$, best and worst cases
- Space efficiency
  - Space for the input sequence, plus a constant number of local variables

Selection Sort

Sort: 34 8 64 51 32 21

- 34 8 64 51 32 21 ^
  - Set marker at 21 and search for largest
- 34 8 21 51 32 ^ 64
  - 34-32 is considered unsorted list
- 34 8 21 32 ^ 51 64
  - **Repeat** this process until unsorted list is empty
- 32 8 21 34 51 64
- 21 8 32 34 51 64
- 8 21 32 34 51 64

Selection Sort

- Quadratic sorting algo
- Searches all elements until it finds the smallest
- Swaps this with the 1st element of the list
- Repeats with the remaining elements swaps with the next element
- O(n2)
- Does not depend on the values of the list

Selection Sort

```
for(int x=0; x<n; x++) {
    int index_of_min = x;
    for(int y=x; y<n; y++) {
        if(array[index_of_min]>array[y]) {
            index_of_min = y;
        }
    }
    int temp = array[x];
    array[x] = array[index_of_min];
    array[index_of_min] = temp;
}
```

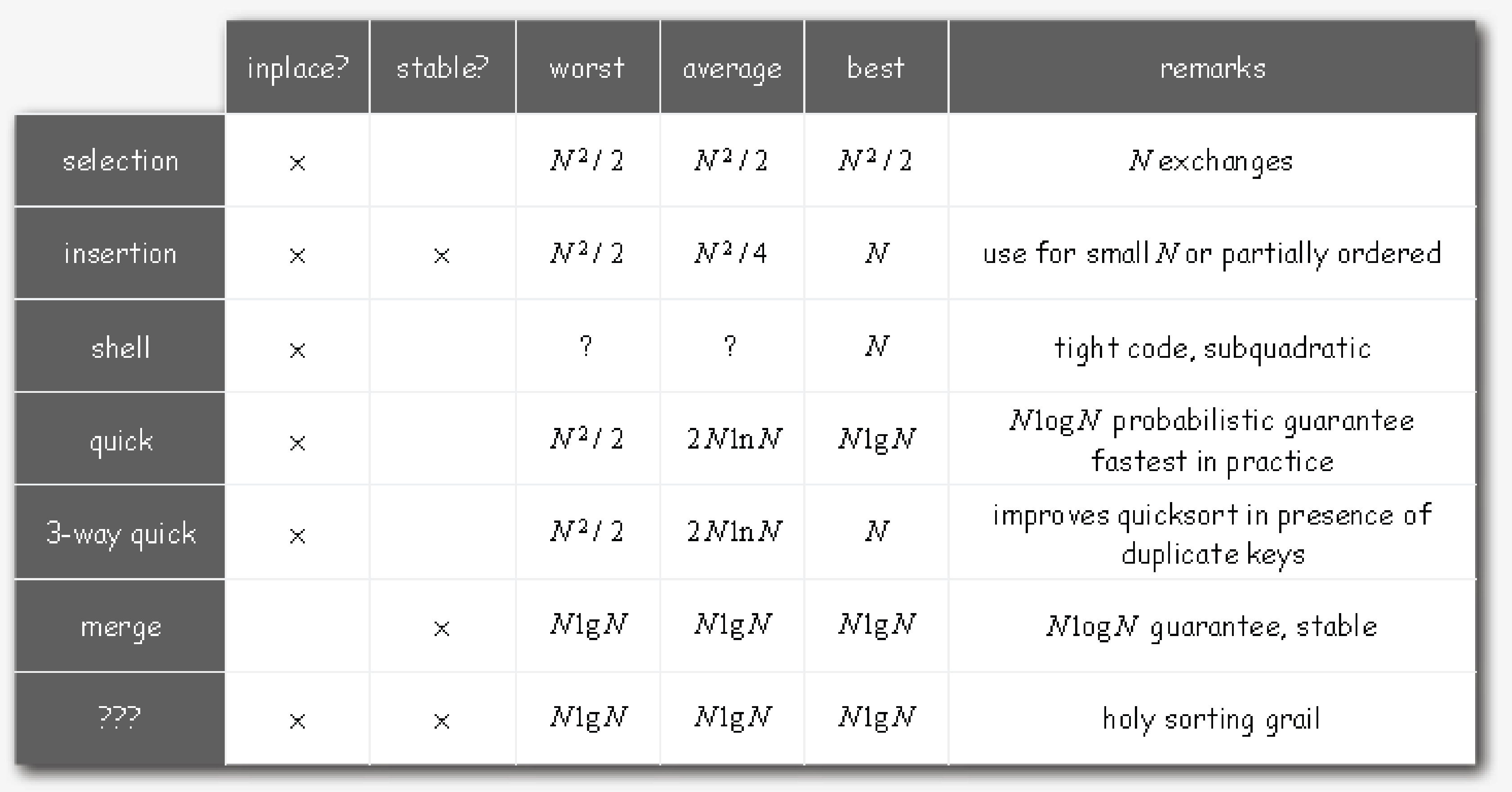
COMPARISON OF ALL SORTS

- Insertion, selection and bubble sort have quadratic worst-case performance
- The faster comparison based algorithm ?

  O(nlogn)

- Mergesort and Quicksort

Summary



	inplace?	stable?	worst	average	best	remarks
selection	×		$N^2/2$	$N^2/2$	$N^2/2$	N exchanges
insertion	×	×	$N^2/2$	$N^2/4$	N	use for small N or partially ordered
shell	×		?	?	N	tight code, subquadratic
quick	×		$N^2/2$	$2N\ln N$	$N\lg N$	$N\log N$ probabilistic guarantee fastest in practice
3-way quick	×		$N^2/2$	$2N\ln N$	N	improves quicksort in presence of duplicate keys
merge		×	$N\lg N$	$N\lg N$	$N\lg N$	$N\log N$ guarantee, stable
???	×	×	$N\lg N$	$N\lg N$	$N\lg N$	holy sorting grail