FPT UNIVERSITY
DREAM OF INNOVATION

6. Sorting

Objectives

- Elementary Sorting Algorithms
  - Selection Sort
  - Insertion Sort
  - Bubble Sort
- Efficient Sorting Algorithms
  - Quick Sort
  - Merge Sort
  - Heap sort
  - Radix Sort
- Sorting in `java.util`

Elementary Sorting Algorithms

Selection Sort

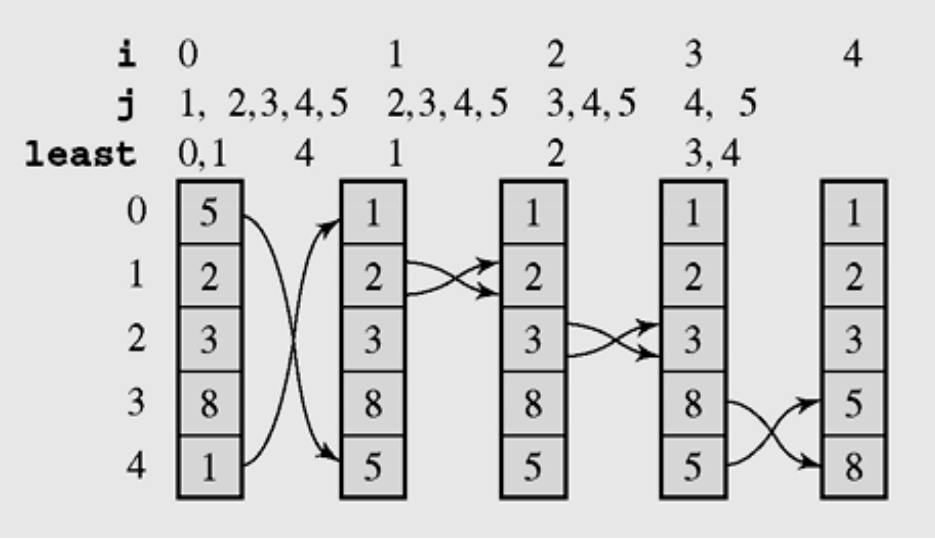
- **Selection sort** is an attempt to localize the exchanges of array elements by finding a misplaced element first and putting it in its final place

```
selectionsort(data[])
  for i = 0 to data.length-2
    select the smallest (or largest) element data[k] among
      data[i], . . . , data[data.length-1];
    swap  data[i] with data[k];
```

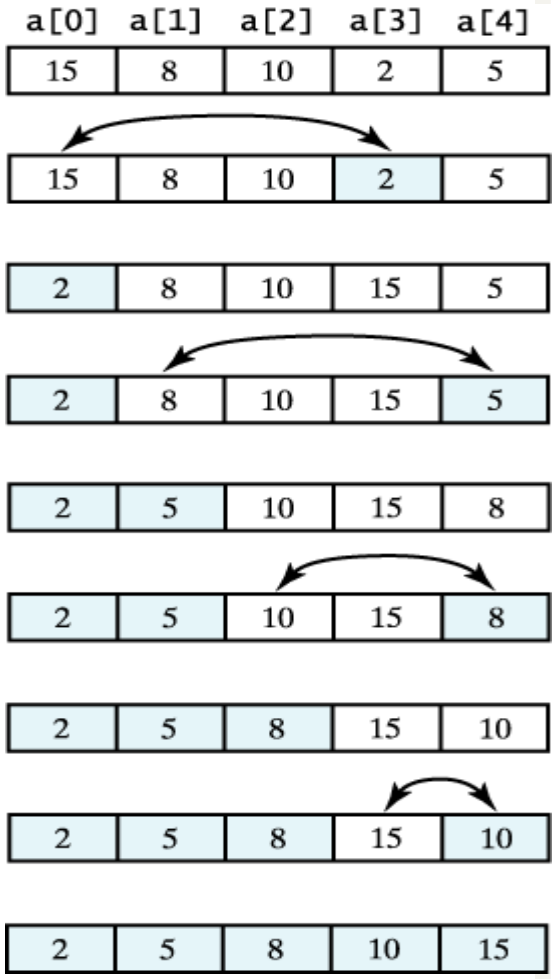
Complexity $O(n^2)$

Selection Sort example

A selection sort of an array of integers into ascending order.



The array [5 2 3 8 1] sorted by selection sort



Selection sort code

```
void selectSort() //Simple Selection Sort
   { int i,j,k;int min;
     for(i=0;i<n-1;i++)
        { min=a[i];k=i;
          for(j=i+1;j<n;j++)
               if(a[j]<min) {k=j;min=a[j];}
          if(k!=i) swap(a,i,k);
        }
     }
```

Insertion sort algorithm

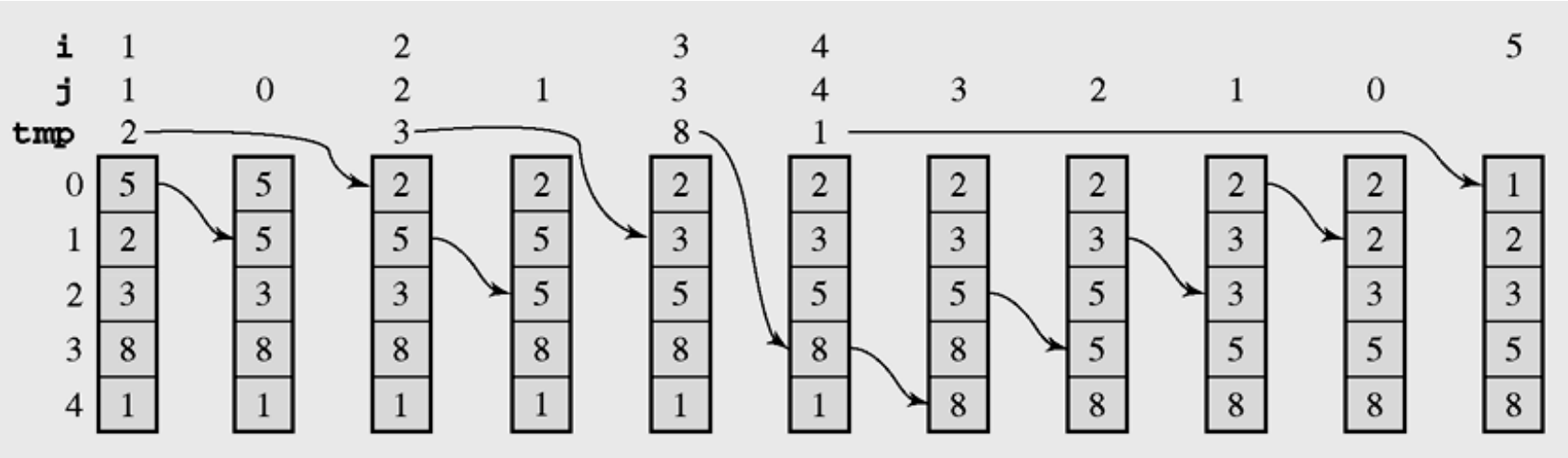
- An **insertion sort** starts by considering the two first elements of the array `data`, which are `data[0]` and `data[1]`
- Next, the third element, `data[2]`, is considered and inserted into its proper place (ascending)

```
insertionsort(data[]) {
    for i = 1 to data.length-1
        tmp = data[i];
        move all elements data[j] greater than tmp by one position;
        place tmp in its proper position;
```

Complexity $O(n^2)$

Best case $O(n)$

Insertion sort example

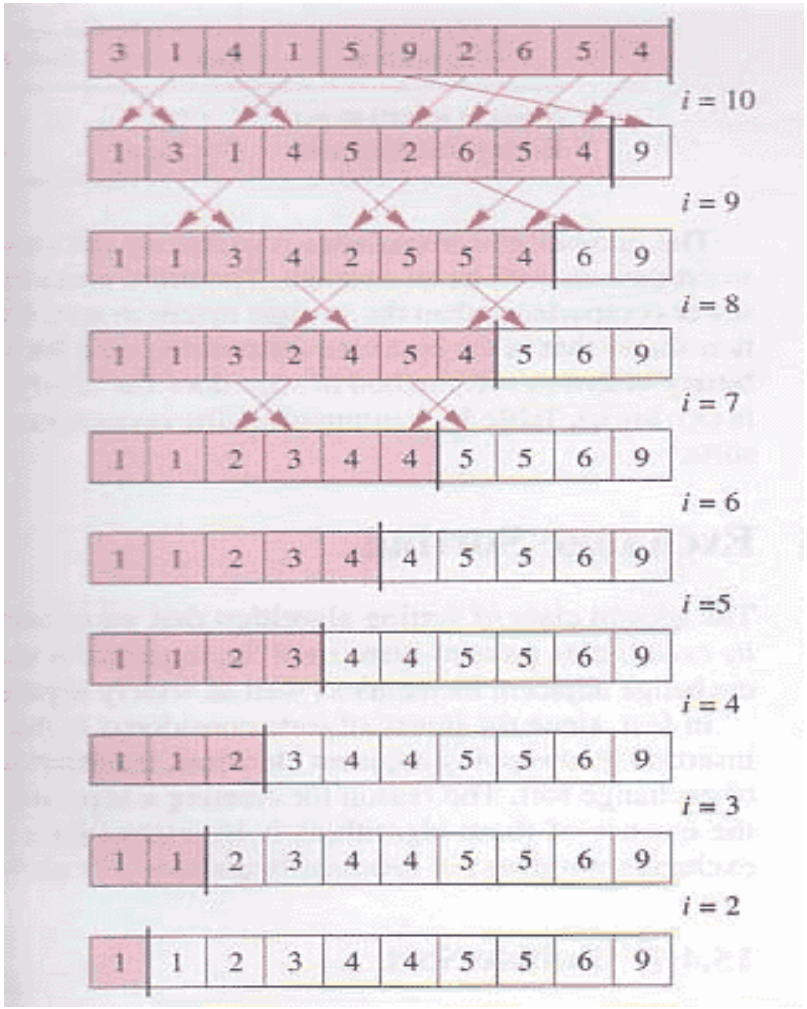


The array [5 2 3 8 1] sorted by insertion sort

Insertion sort code

```
void insertSort()
  {  int i,j,x;
     for(i=1;i<n;i++)
       { x=a[i]; j=i;
         while(j>0 && x<a[j-1])
           { a[j]=a[j-1];  j--;
           };
         a[j]=x;
       }
     }
  }
```

Bubble sort




```
void bubbleSort()
    { int i; boolean swapped;
      do
        { swapped=false;
          for(i=0;i<n-1;i++)
          if(a[i]>a[i+1])
              { swap(a,i,i+1);
                swapped=true;
              }
        }
      while(swapped);
    }
```

Complexity $O(n^2)$

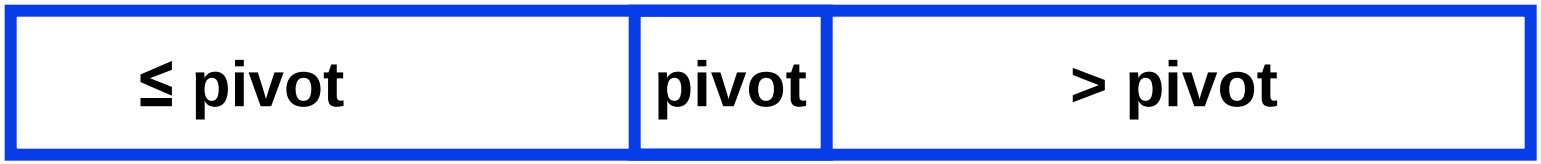
Efficient Sorting Algorithms

Quicksort - 1

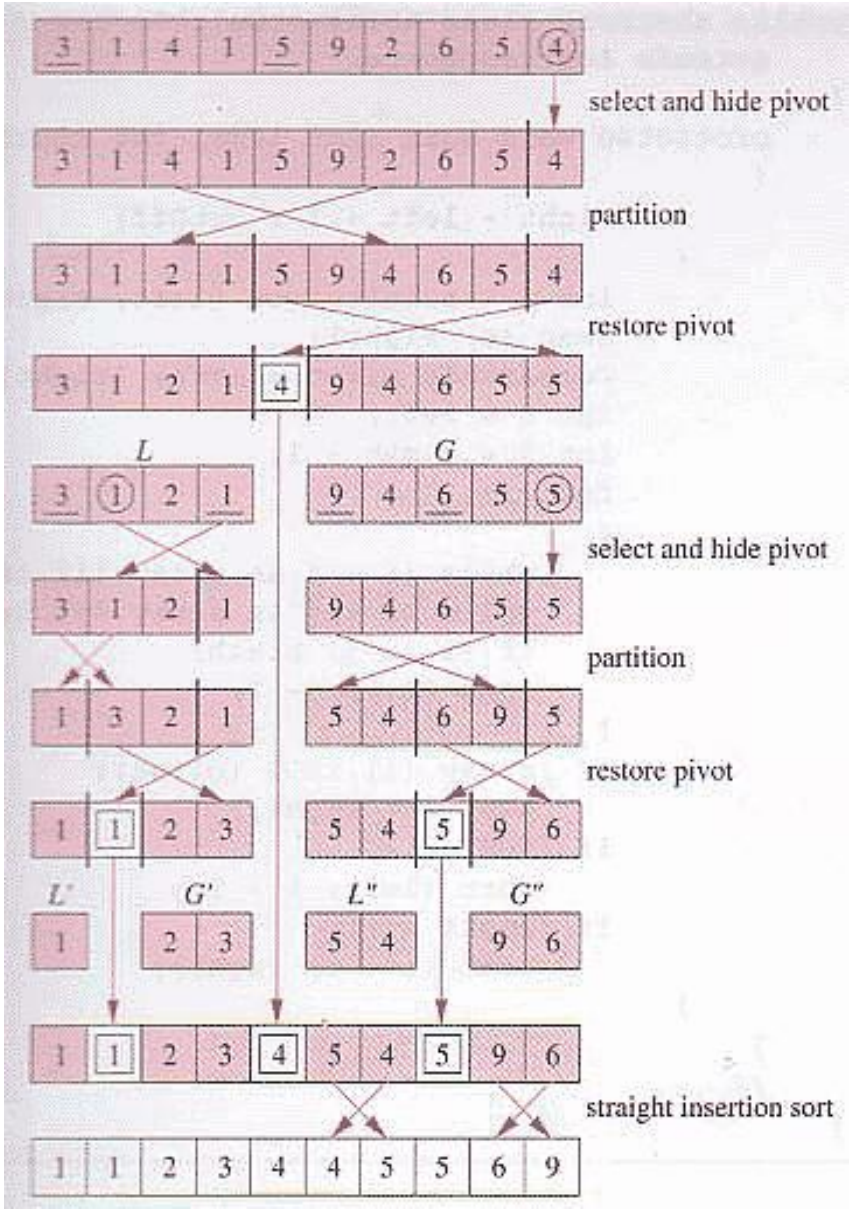
- Efficient sorting algorithm
  - Discovered by C.A.R. Hoare
- Example of Divide and Conquer algorithm
- Two phases
  - Partition phase
    - Divides the work into half
  - Sort phase
    - Conquers the halves!

Quicksort - 2

- 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



- Conquer
  - Apply the same algorithm to each half



Quicksort - 3

How do we Partition?

1. Define the pivot value as the contents of Table[First]
2. Initialize Up to First and Down to Last
3. Repeat
    4. Increment Up until Up selects the first element greater than the pivot value
    5. Decrement Down until it selects the first element less than or equal to the pivot value
    6. if Up < Down exchange their values

   until Up meets or passes Down
7. Exchange Table[First] and Table[Down]
8. Define PivIndex as Down

Quicksort code

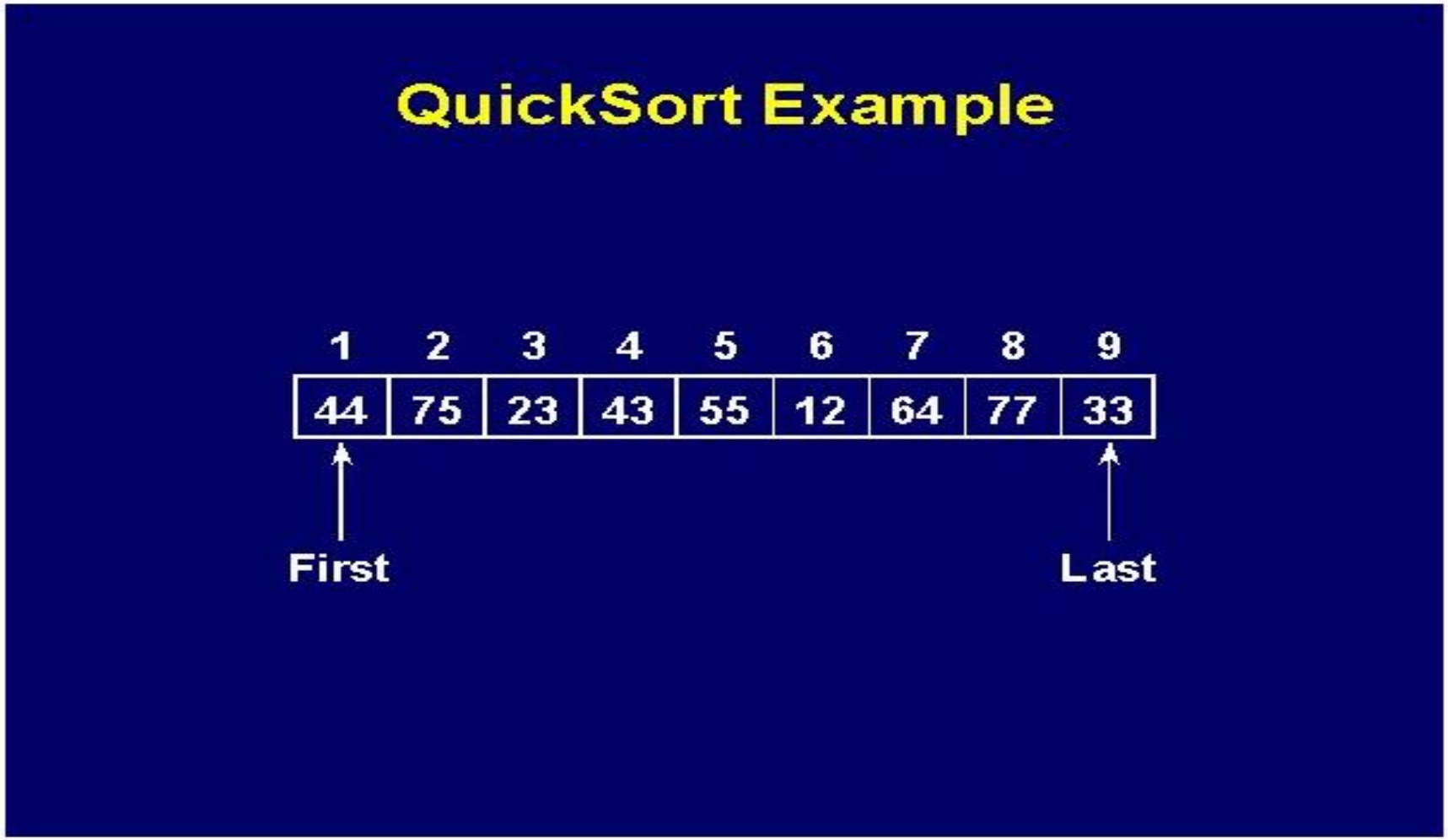
QuickSort Procedure Code

```
void QuickSort ( int Table [ ], int First, int Last ) {
   int PivIndex;
   if ( First < Last ) {
        PivIndex = Partition ( Table, First, Last );
        QuickSort ( Table, First, PivIndex-1 );
        QuickSort ( Table, PivIndex+1, Last ); }
   }
```

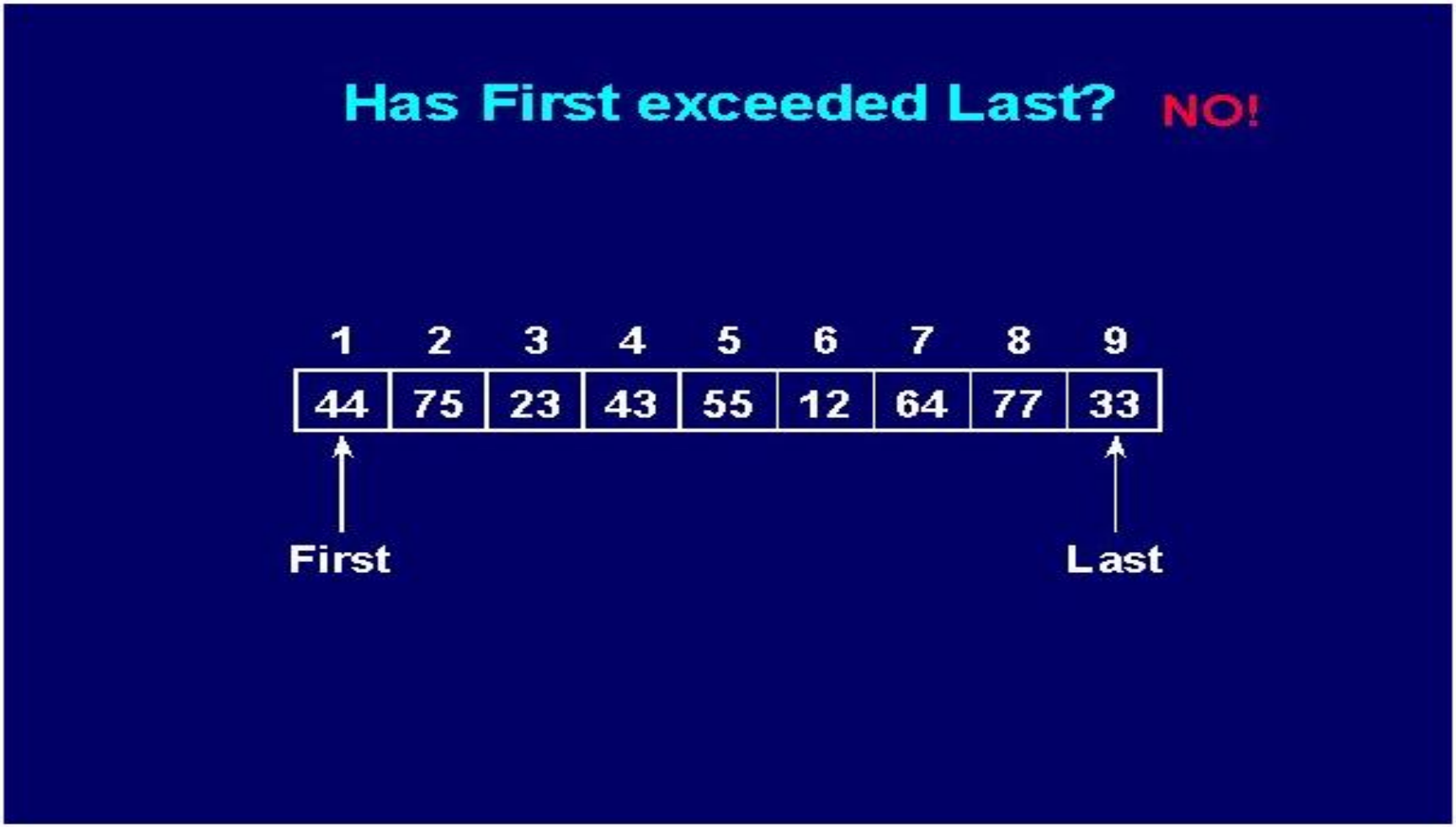
Quicksort complexity

- Quick Sort
  - *In the best case and average case: O(n* log *n) but ….*
  - Can be O*(n^2), the (rear) worst case. Moreover, the constant hidden in the O-notation is small.*
  - Depends on pivot selection
    - Median-of-3
    - Random pivot
    - *Better but not guaranteed*

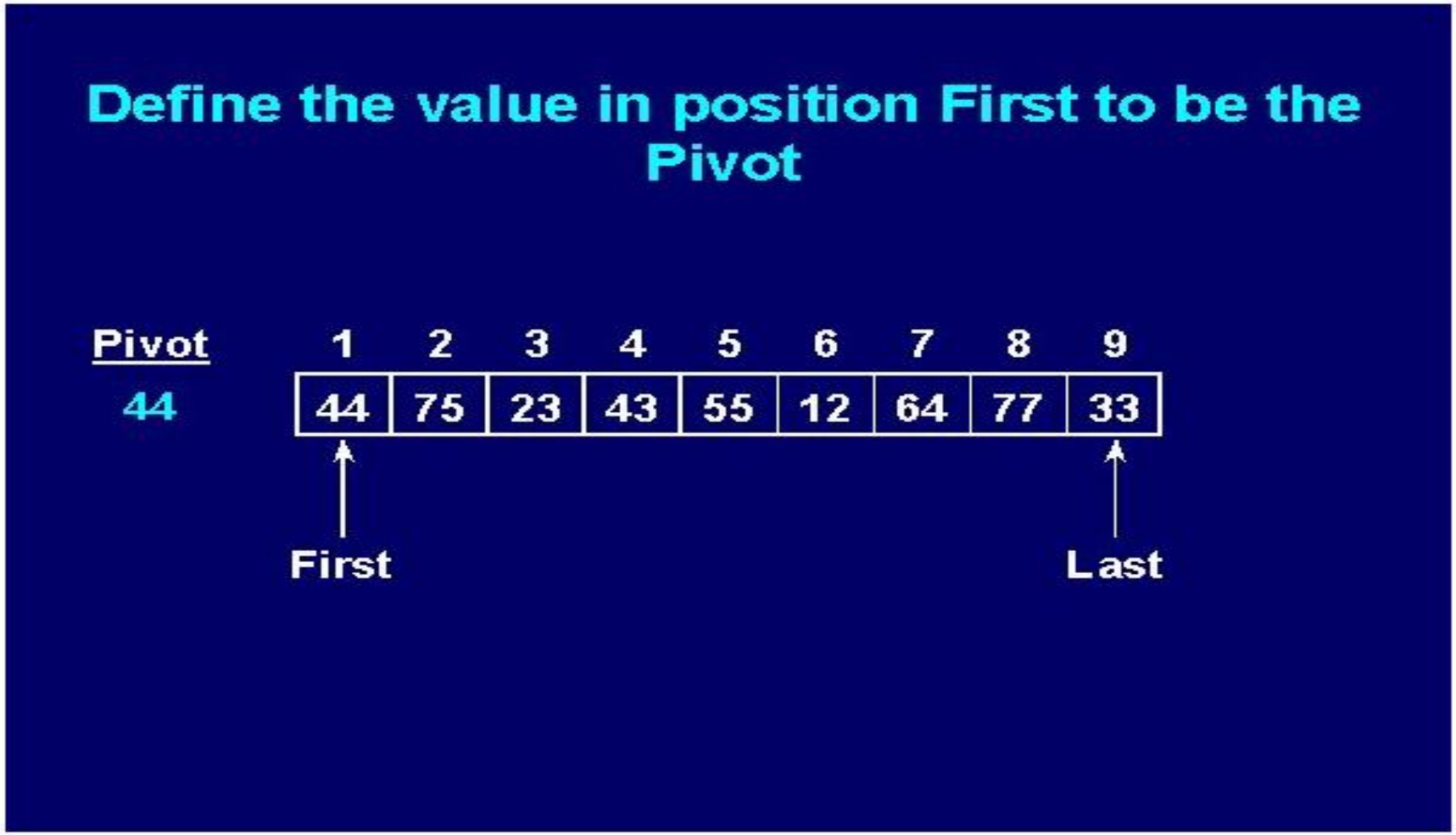
Quicksort example - 1



Quicksort example - 2

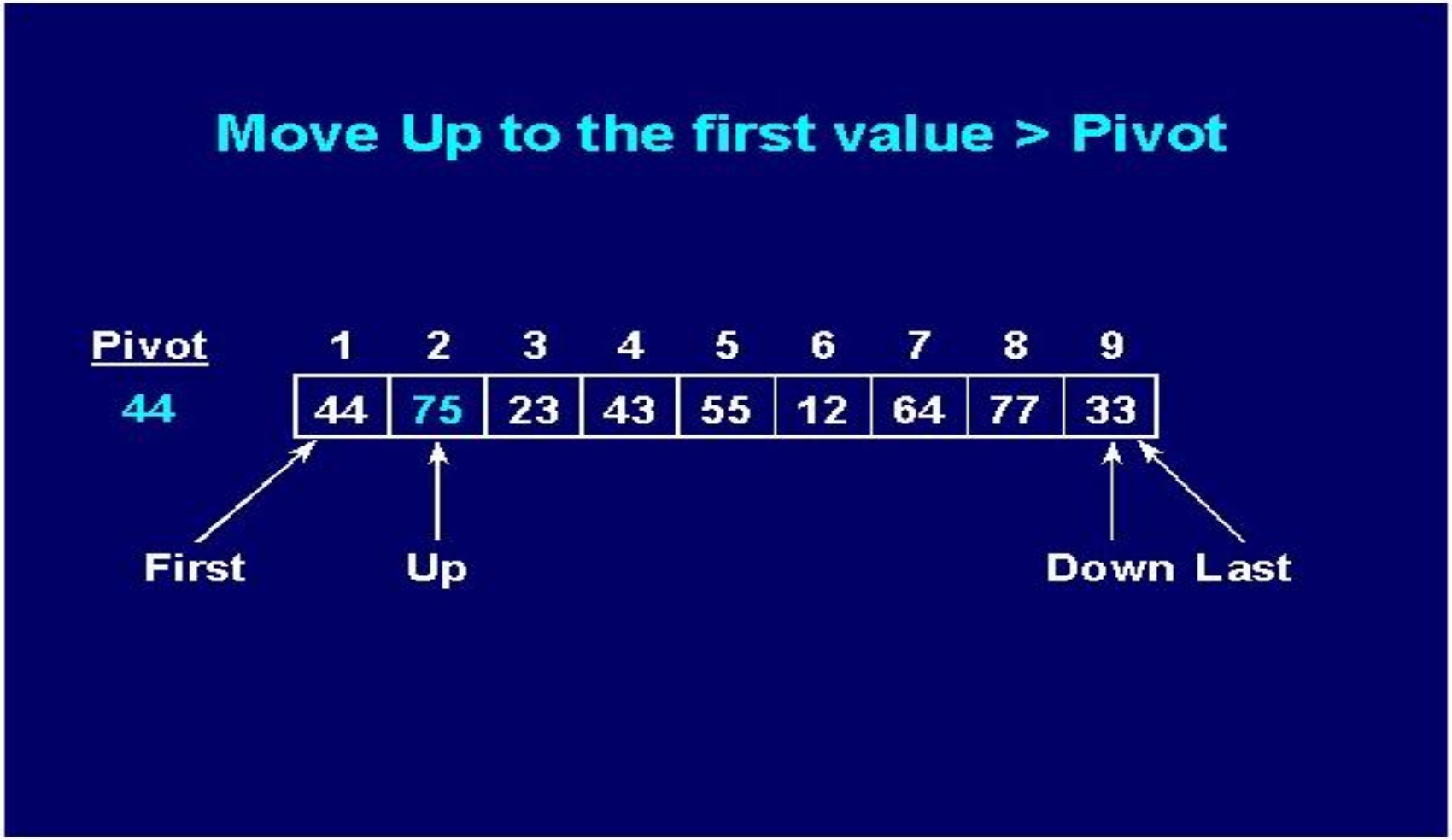


Quicksort example - 3

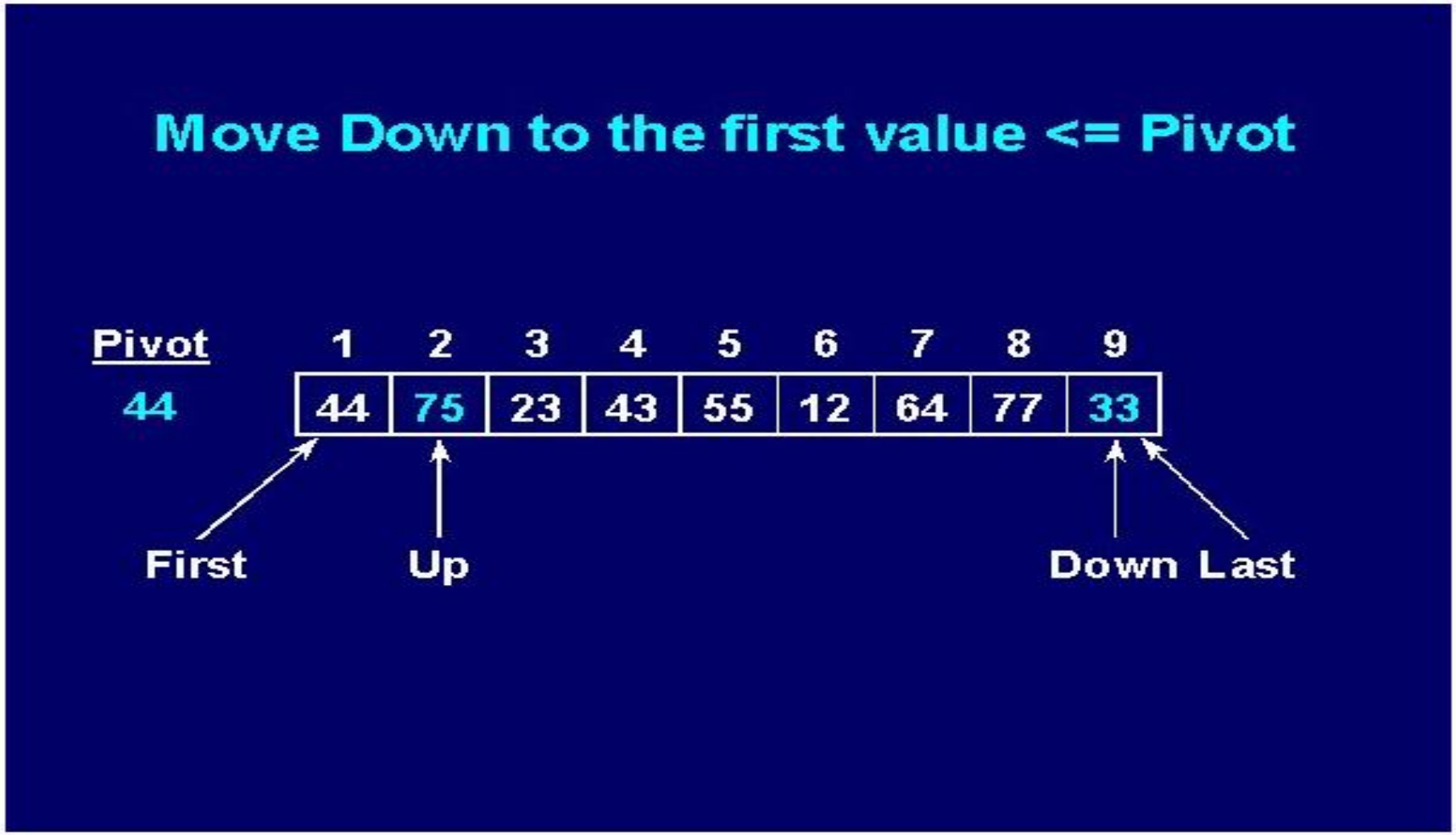


Quicksort example - 4

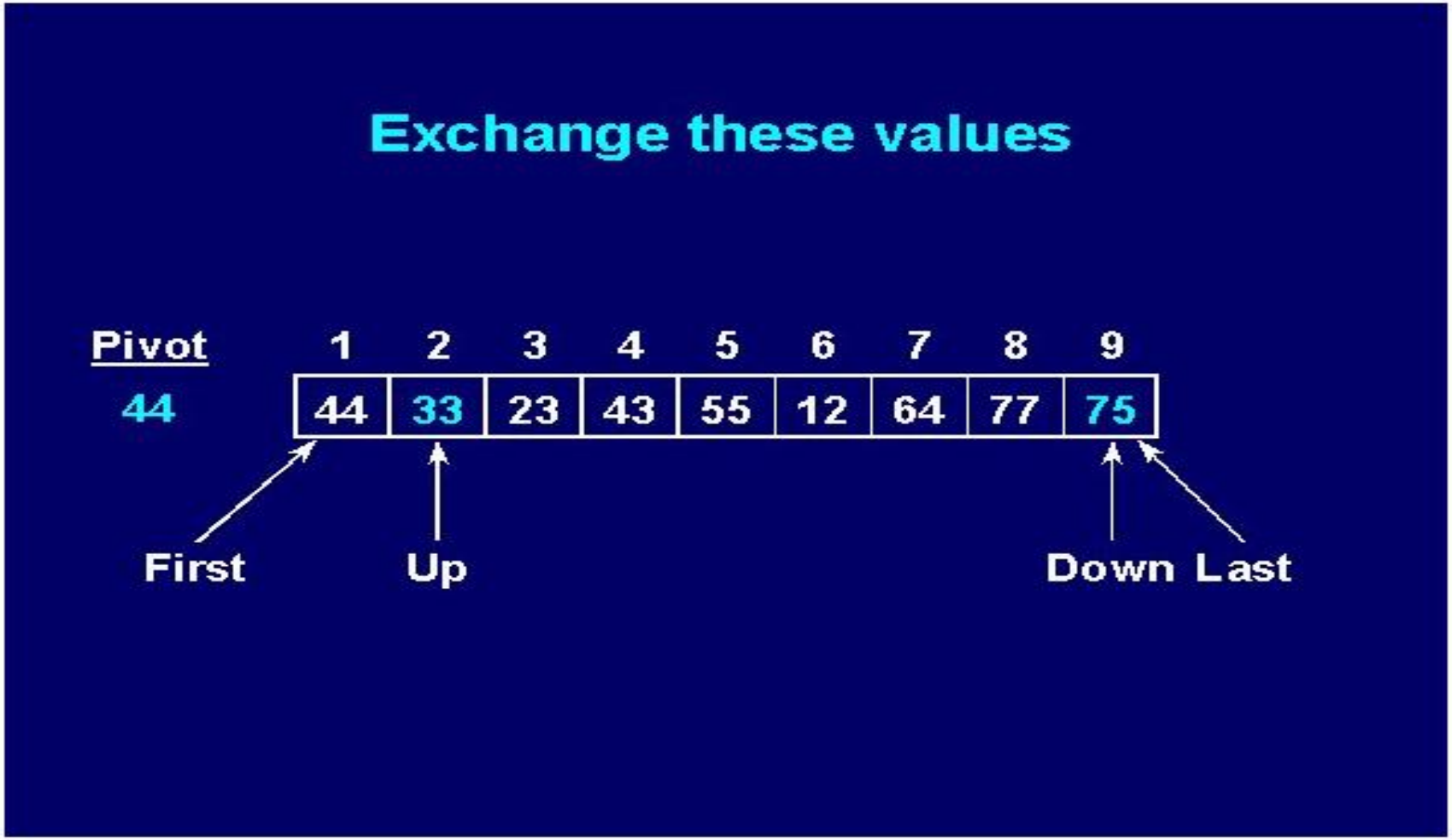
Quicksort example - 5



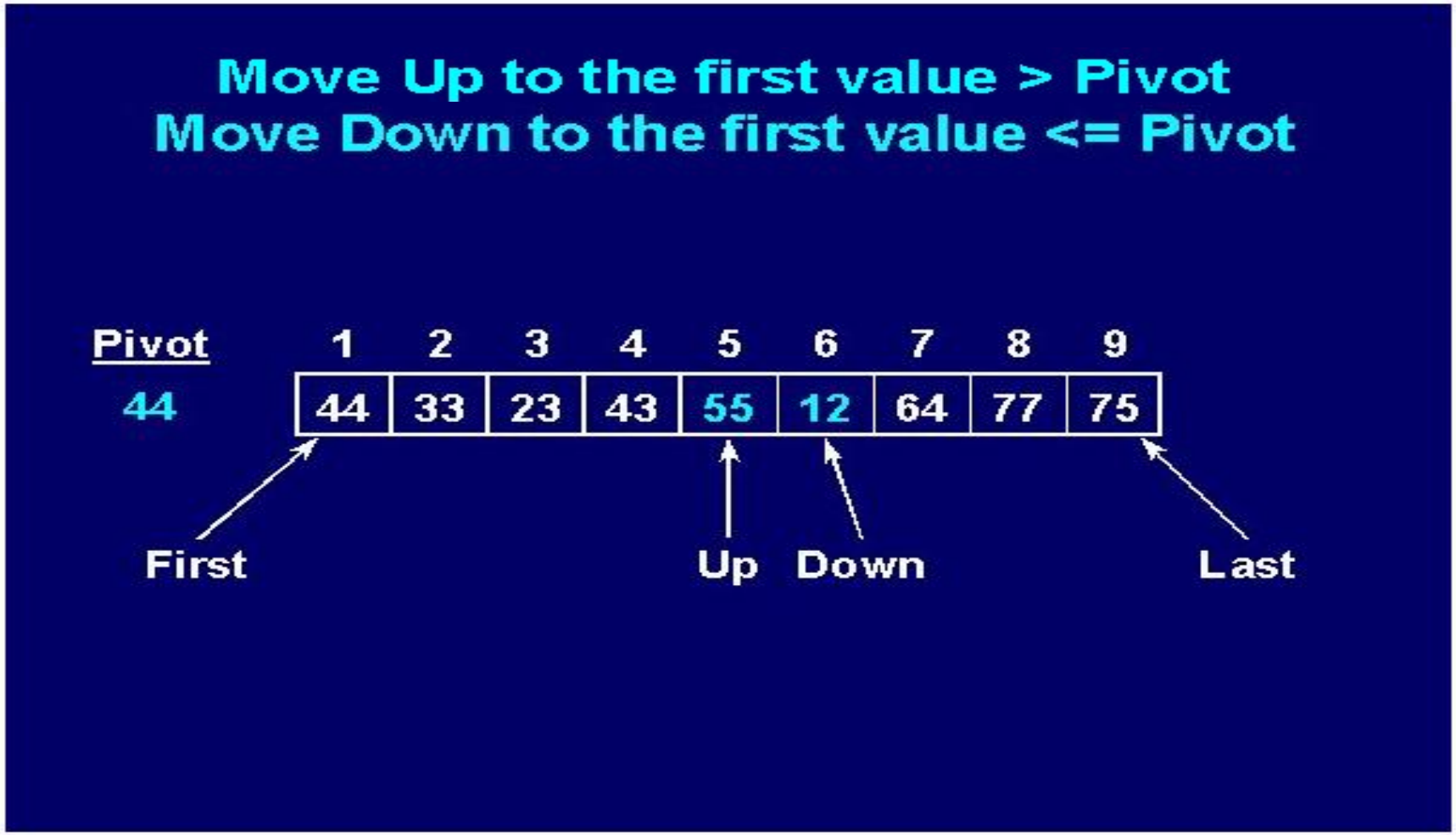
Quicksort example - 6



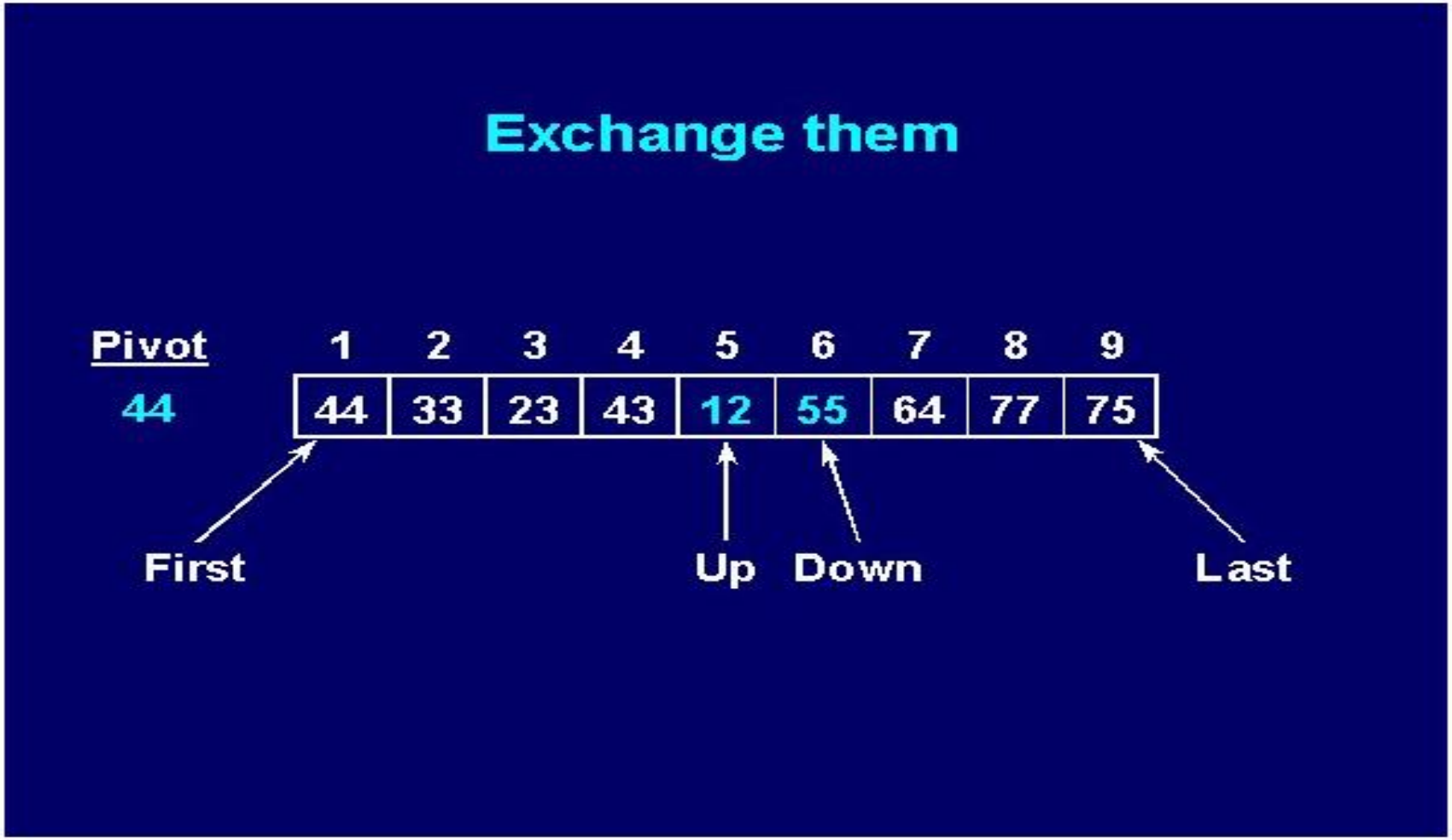
Quicksort example - 7



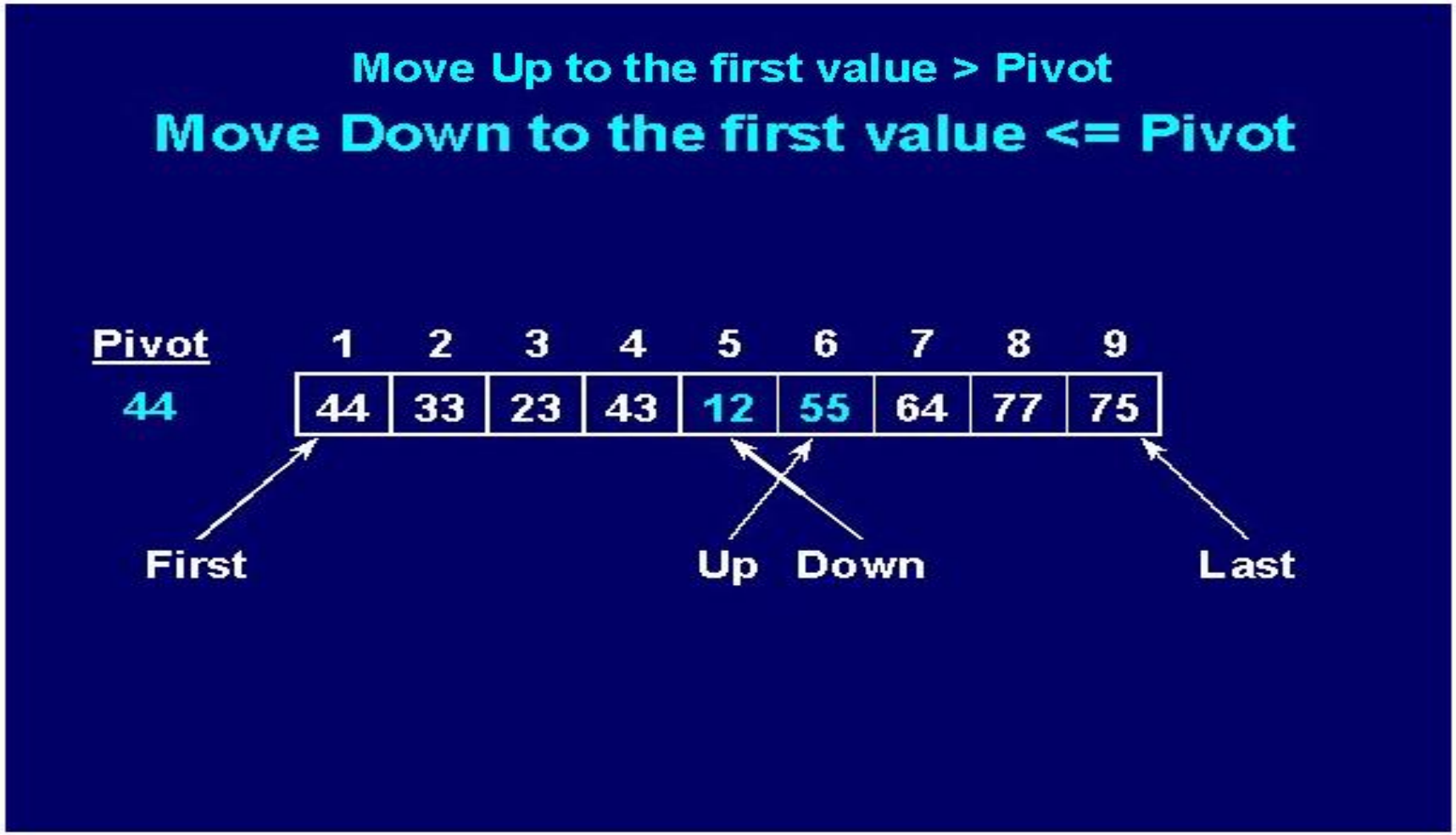
Quicksort example - 8



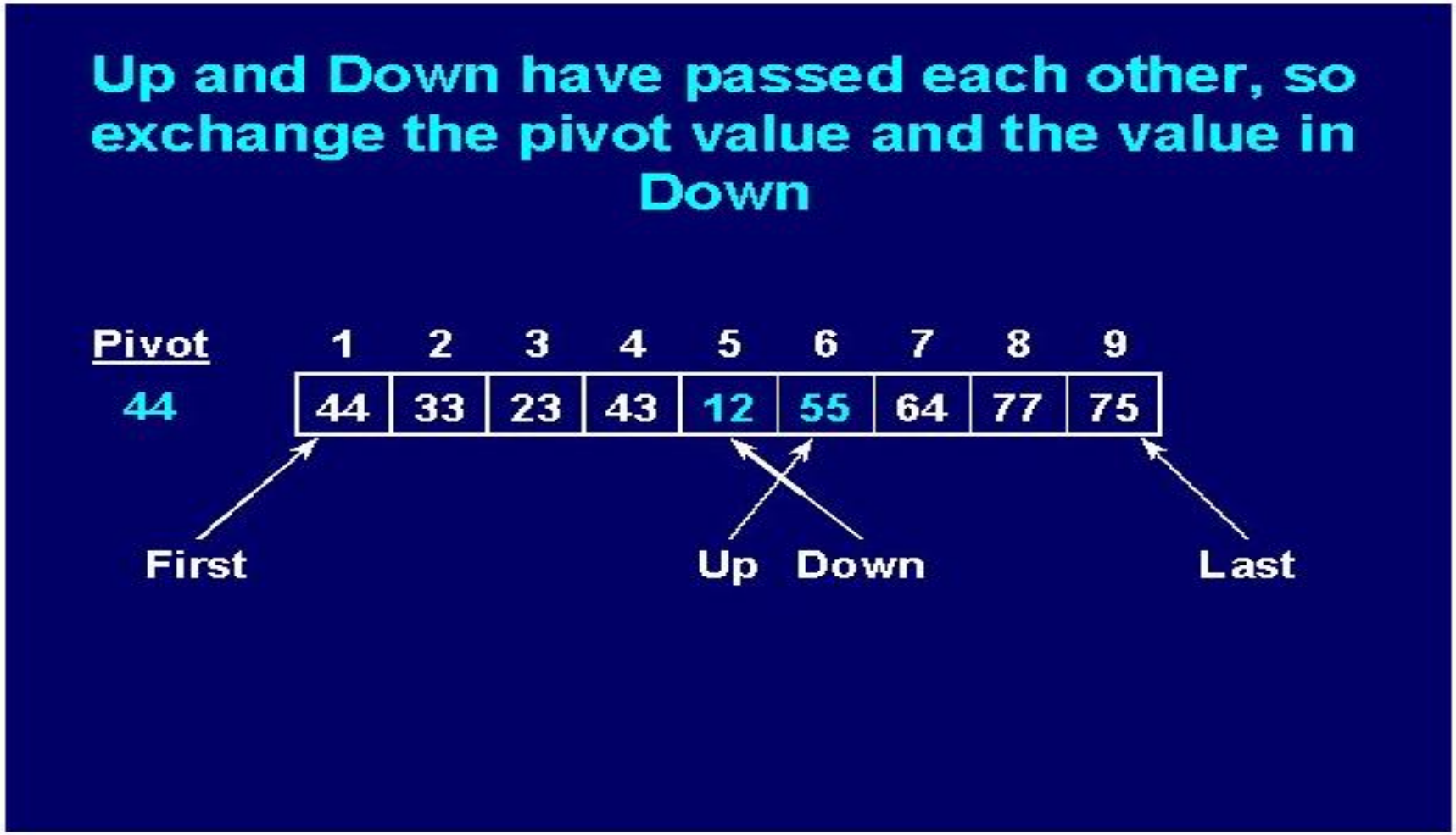
Quicksort example - 9



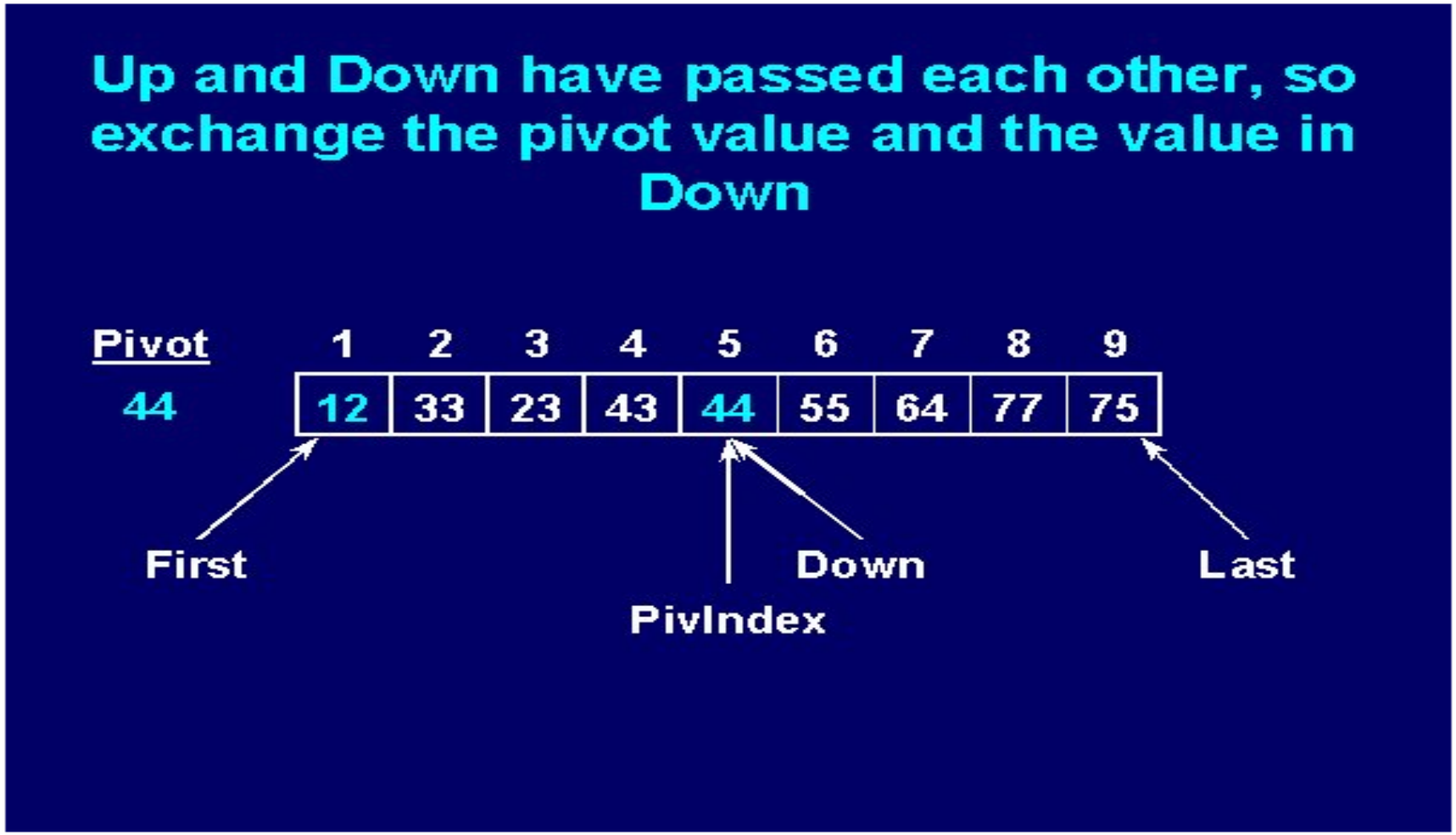
Quicksort example - 10



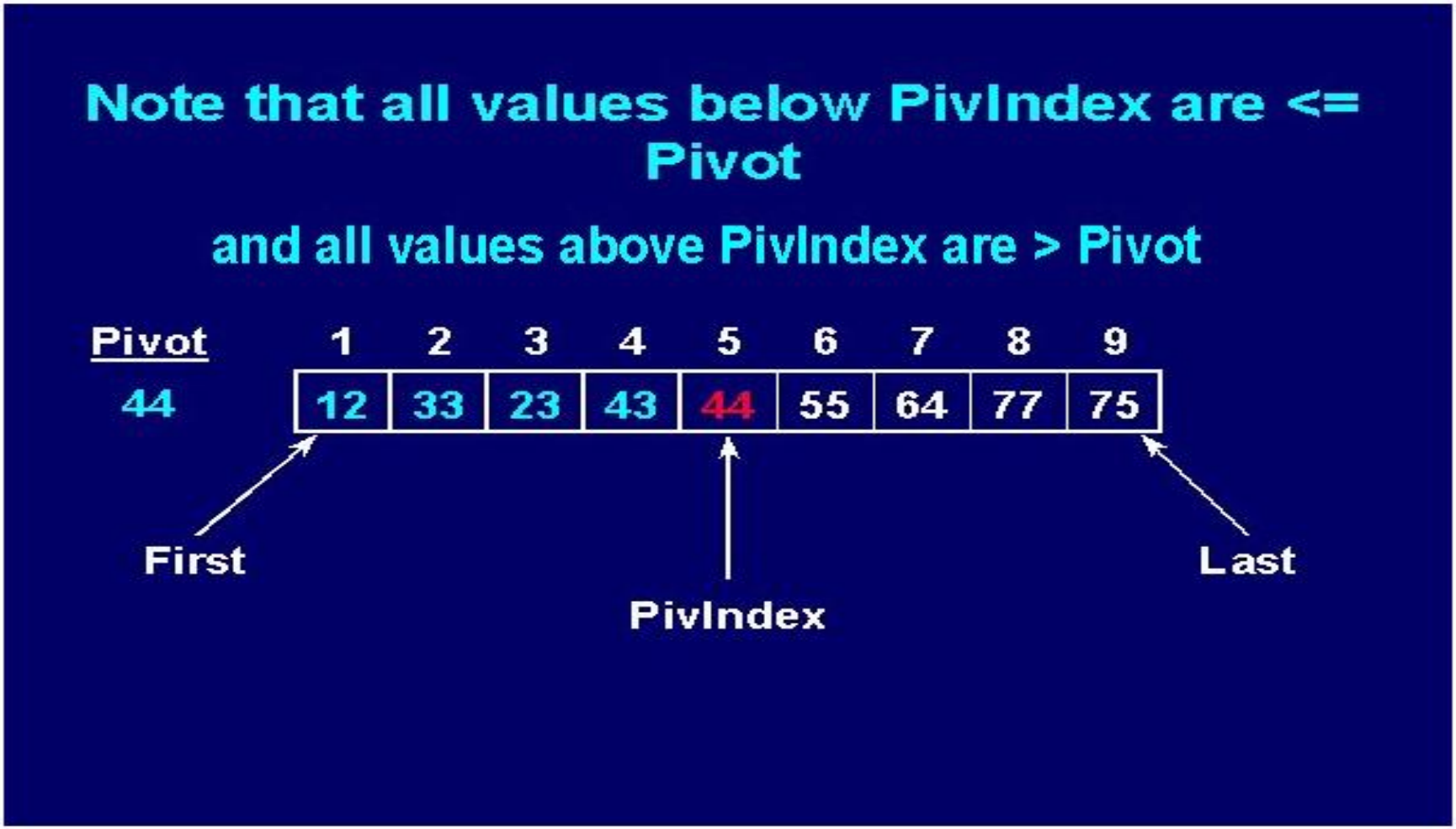
Quicksort example - 11



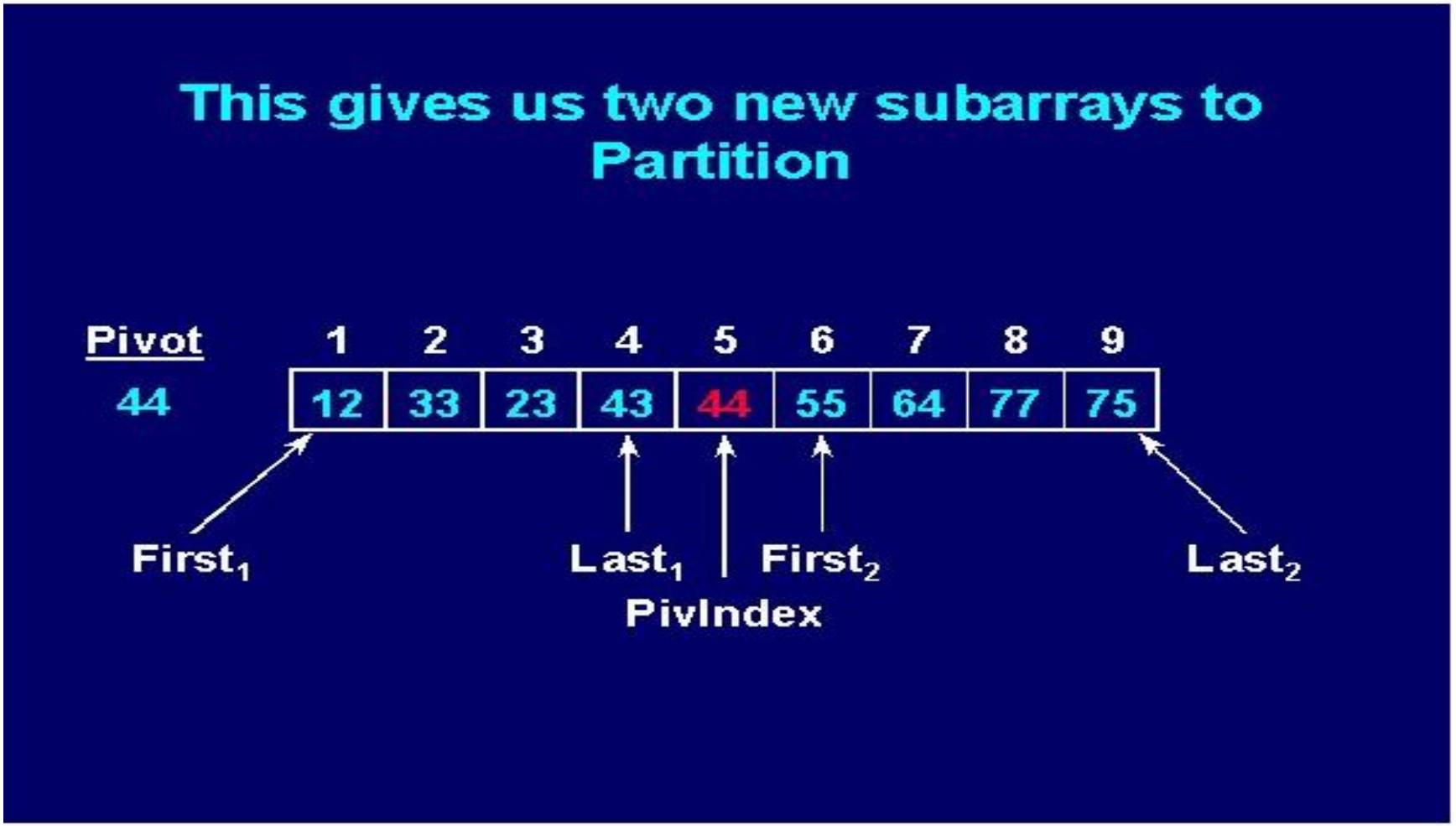
Quicksort example - 12



Quicksort example - 13



Quicksort example - 14

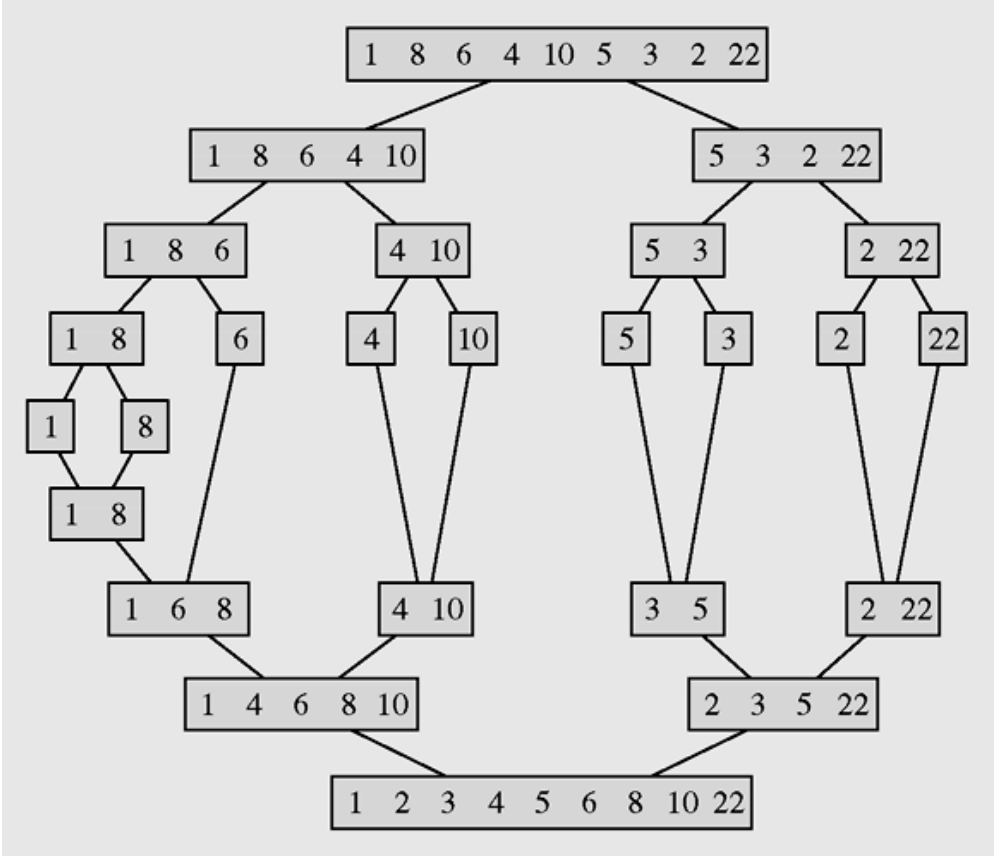


Mergesort

- **Mergesort** makes partitioning as simple as possible and concentrates on merging sorted halves of an array into one sorted array
- Complexity is O(nlogn)
- It was one of the first sorting algorithms used on a computer and was developed by John von Neumann

```
mergesort(data)
 if data have at least two elements
     mergesort(left half of data);
     mergesort(right half of data);
     merge(both halves into a sorted list);
```

Mergesort example



The array [1 8 6 4 10 5 3 2 22] sorted by mergesort

Merge sort code

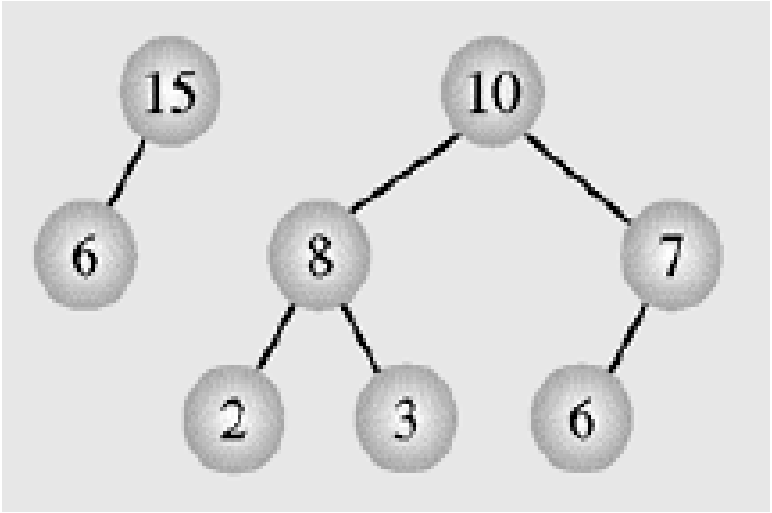
```
void mergeSort(int p, int r)
   {if(p>=r) return;
     int q = (p+r)/2;
     mergeSort(p,q);
     mergeSort(q+1,r);
     merge(p,q,r);
   }
```

```
public class Main
 {public static void main(String args[])
   {int [] b = {7,3,5,9,11,8,6,15,10,12,14};
    EffSort t = new EffSort(b);
    int n=b.length;
    t.mergeSort(0,n-1);t.display();
    System.out.println();
   }
 }
```

```
void merge(int p, int q, int r)
  {if(!(p<=q) && (q<=r)) return;
    int n,i,j,k,x; n = r-p+1;
    int [] b = new int[n];
    i=p;j=q+1;k=0;
    while(i<=q && j<=r)
     {if(a[i]<a[j])
        b[k++] = a[i++];
       else
        b[k++] = a[j++];
     }
    while(i<=q)  b[k++] = a[i++];
    while(j<=r)  b[k++] = a[j++];
    k=0;
    for(i=p;i<=r;i++) a[i] = b[k++];
  }
```

Heap data structure - 1

- **Heap** is a particular kind of binary tree, which has two properties:
  - The value of each node is greater than or equal to the values stored in each of its children.
  - If the height of the tree is h, then the tree is complete at level d for d = 1, 2, ..., h-1, and the leaves in the last level are all in the leftmost positions

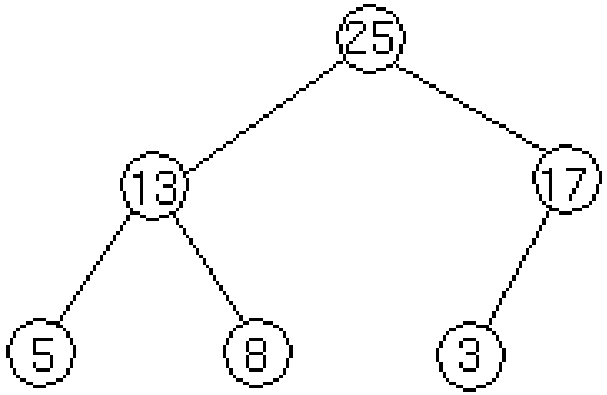


(This kind of binary tree is called nearly complete binary tree)

- These two properties define a **max heap**
- If "greater" in the first property is replaced with "less," then the definition specifies a **min heap**

Heap data structure - 2

We can represent heap by array in level order, going from left to right. The array corresponding to the heap above is [25, 13, 17, 5, 8, 3].



The root of the tree a[0] and given index *i* of a node, the indices of its parent, left child and right child can be computed:

```
PARENT (i)
    return floor((i-1)/2)
LEFT (i)
    return 2i+1
RIGHT (i)
    return 2(i + 1)
```

Heap Sort - 1

Heap sort consists of 2 steps:

- Transform the array to a heap:

  Suppose the array a to be sorted has n elements: a[0], a[1], ..., a[n-1]

  At first the heap has only one element: a[0]. Then

  for i =1 to n-1 we insert the element a[i] to heap so that the heap property is maintained.

- Transform the heap to sorted array:

  for k = 1 to n-1 do

  remove and put the root (a[0]) to the position n-k, then rearrange elements to keep the heap property.

Heap sort code

Addition and deletion are both **O(log*n*)** operations. We need to perform ***n*** additions and deletions, leading to an **O(*n*log*n*)** algorithm.

```
// Transform heap to sorted array
for(i=n-1;i>0;i--)
  { x=a[i];a[i]=a[0];
    f=0; //f is father
    s=2*f+1; //s is a left son
    // if the right son is larger then it is selected
    if(s+1<i && a[s]<a[s+1]) s=s+1;
    while(s<i && x<a[s])
      { a[f]=a[s]; f=s; s=2*f+1;
        if(s+1<i && a[s]<a[s+1]) s=s+1;
      };
     a[f]=x;
  };
```

```
//Transform the array to HEAP
int i,s,f;int x;
for(i=1;i<n;i++)
  { x=a[i]; s=i; //s  is a son, f=(s-1)/2 is
     father
    while(s>0 && x>a[(s-1)/2])
     { a[s]=a[(s-1)/2];   s=(s-1)/2;
     };
    a[s]=x;
  };
```

Complexity $O(n\log_2 n)$

Radix sort - 1

Radix Sort is a clever and intuitive little sorting algorithm. Radix Sort puts the elements in order by comparing the **digits of the numbers**. We will explain with an example.

Consider the following 9 numbers:

493 812 715 340 195 437 710 582 385

We should start sorting by comparing and ordering the **one's** digits:

Notice that the numbers were added onto the list in the order that they were found, which is why the numbers appear to be unsorted in each of the sublists above. Now, we gather the sublists (in order from the 0 sublist to the 9 sublist) into the main list again:

340 710 812 582 493 715 195 385 437

Note: The **order** in which we divide and reassemble the list is **extremely important**, as this is one of the foundations of this algorithm.

Digit	Sublist
0	340 710
1	
2	812 582
3	493
4	
5	715 195 385
6	
7	437
8	
9	

Radix sort - 2

Now, the sublists are created again, this time based on the **ten's** digit:

Digit	Sublist
0	
1	710 812 715
2	
3	437
4	340
5	
6	
7	
8	582 385
9	493 195

Now the sublists are gathered in order from 0 to 9:
710 812 715 437 340 582 385 493 195
Finally, the sublists are created according to the **hundred's** digit:

Digit	Sublist
0	
1	195
2	
3	340 385
4	437 493
5	582
6	
7	710 715
8	812
9	

At last, the list is gathered up again:
195 340 385 437 493 582 710 715 812
And now we have a fully sorted array! Radix Sort is very simple, and a computer can do it fast. When it is programmed properly, Radix Sort is in fact **one of the fastest sorting algorithms** for numbers or strings of letters.

Disadvantages

Still, there are some tradeoffs for Radix Sort that can make it less preferable than other sorts.
The speed of Radix Sort largely depends on the inner basic operations, and **if** the operations are not efficient enough, **Radix Sort *can* be slower than some other algorithms** such as Quick Sort and Merge Sort. These operations include the insert and delete functions of the sublists and the process of isolating the digit you want.

Sorting in java.util - 1

- Java provides two sets of versions of sorting methods: one for arrays and one for lists
- The utility class `Arrays` includes a method for:
  - Searching arrays for elements with binary search
  - Filling arrays with a particular value
  - Converting an array into a list, and sorting methods

Sorting in java.util - 2

- The sorting methods are provided for arrays with elements of all elementary types except Boolean
- For each type of sorting method there are two versions:
  - One for sorting an entire array
  - One for sorting a subarray

```
public static void sort(int[] a);
public static void sort(int[] a, int first, int last);
```

Summary

- Elementary Sorting Algorithms
  - Selection Sort
  - Insertion Sort
  - Bubble Sort
- Efficient Sorting Algorithms
  - Quick Sort
  - Merge Sort
  - Heap sort
  - Radix Sort
- Sorting in java.util

Reading at home

Text book: Data Structures and Algorithms in Java

- 12 Sorting and Selection 531
- 9.4.1 Selection-Sort and Insertion-Sort - 386
- bubble-sort (see Exercise C-7.51)
- 12.2 Quick-Sort - 544
- 12.1 Merge-Sort - 532
- 9.4.2 Heap-Sort - 388
- 12.3.2 Linear-Time Sorting: Bucket-Sort and Radix-Sort - 558
- 12.4 Comparing Sorting Algorithms - 561