The Quick Sort

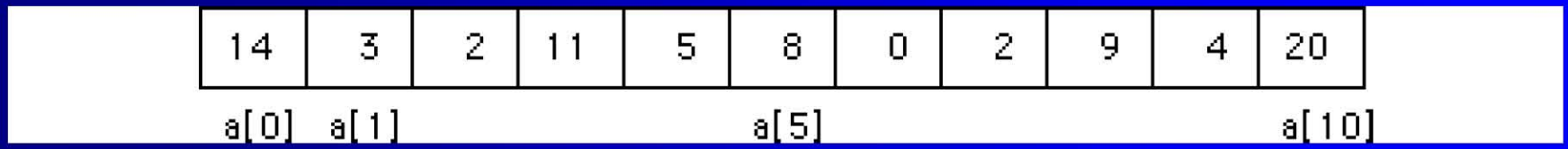
Textbook Authors:
Ken Lambert & Doug Nance
PowerPoint Lecture
by Dave Clausen

Quick Sort Description

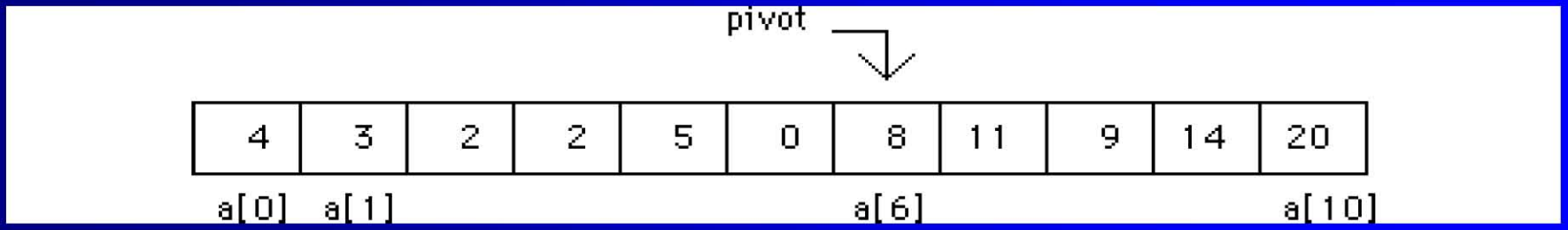
- One of the fastest sorting techniques available is the *quick sort*.
- Like binary search, this method uses a recursive, divide and conquer strategy.
- The basic idea is to separate a list of items into two parts, surrounding a distinguished item called the *pivot*.
- At the end of the process, one part will contain items smaller than the pivot and the other part will contain items larger than the pivot.

Quick Sort Trace

- If an unsorted list (represented as vector `a`) originally contains:



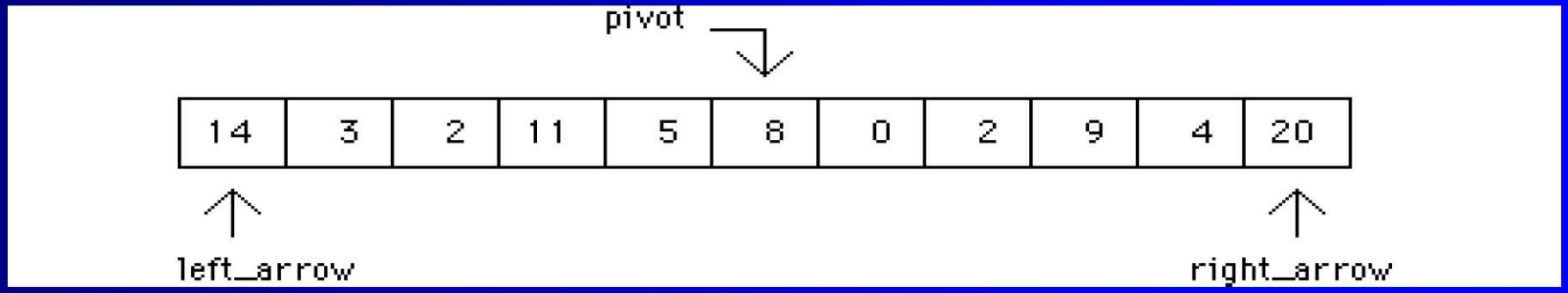
- we might select the item in the middle position, `a[5]`, as the pivot, which is 8 in our illustration. Our process would then put all values less than 8 on the left side and all values greater than 8 on the right side.
- This first subdivision produces



- Now, each sublist is subdivided in exactly the same manner. This process continues until all sublists are in order. The list is then sorted. This is a recursive process.

Quick Sort Trace 2

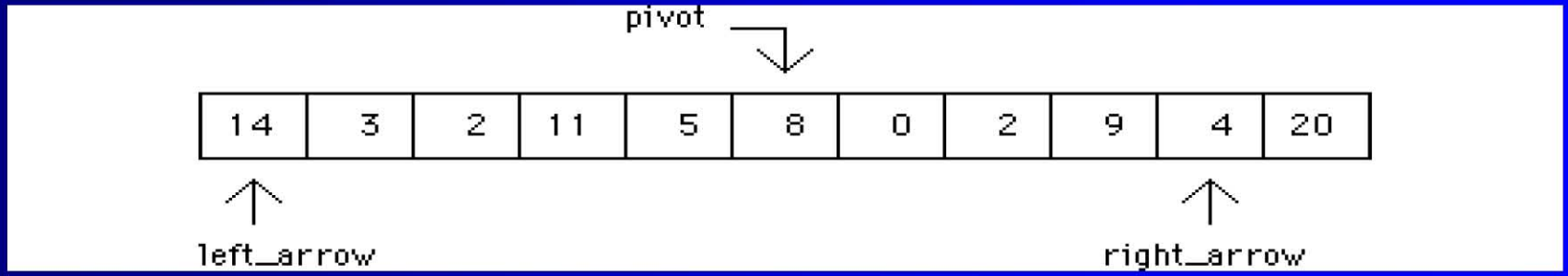
- We choose the value in the middle for the pivot.
- As in binary search, the index of this value is found **by `(first + last) / 2`**,
  - where **`first`** and **`last`** are the indices of the initial and final items in the vector representing the list.
  - We then identify a **`left_arrow`** and **`right_arrow`** on the far left and far right, respectively.
  - This can be envisioned as:



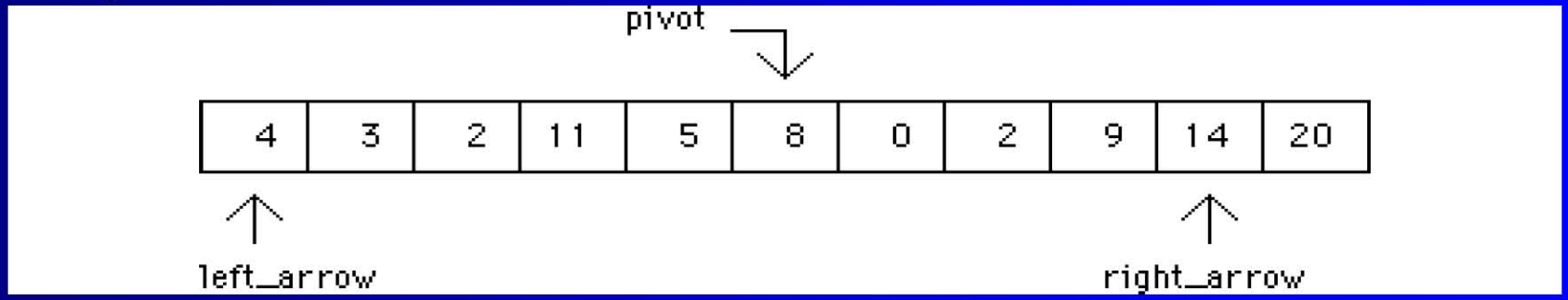
  - where **`left_arrow`** and **`right_arrow`** initially represent the lowest and highest indices of the vector items.

Quick Sort Trace 3

- Starting on the right, the **`right_arrow`** is moved left until a value less than or equal to the pivot is encountered. This produces

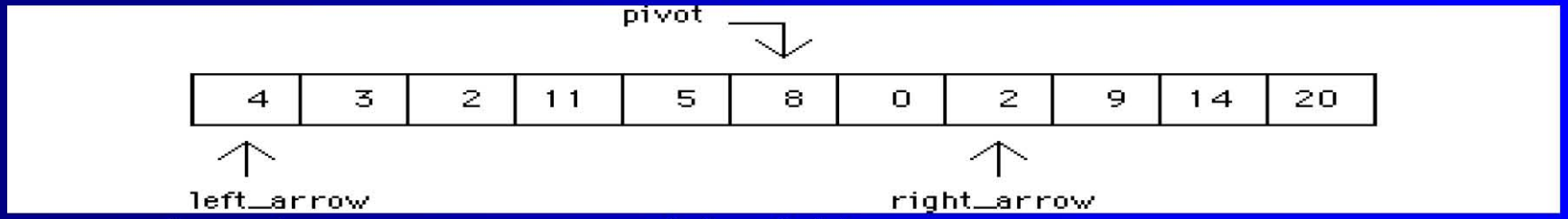


- In a similar manner, **`left_arrow`** is moved right until a value greater than or equal to the pivot is encountered. This is the situation just encountered. Now the contents of the two vector items are **swapped** to produce

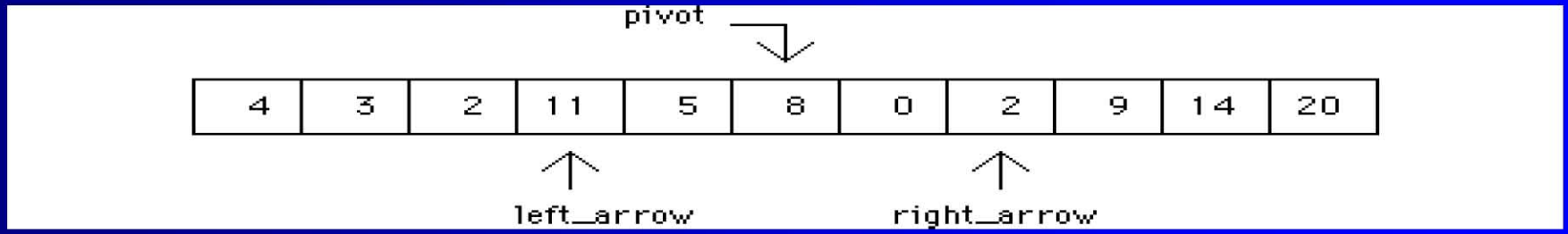


Quick Sort Trace 4

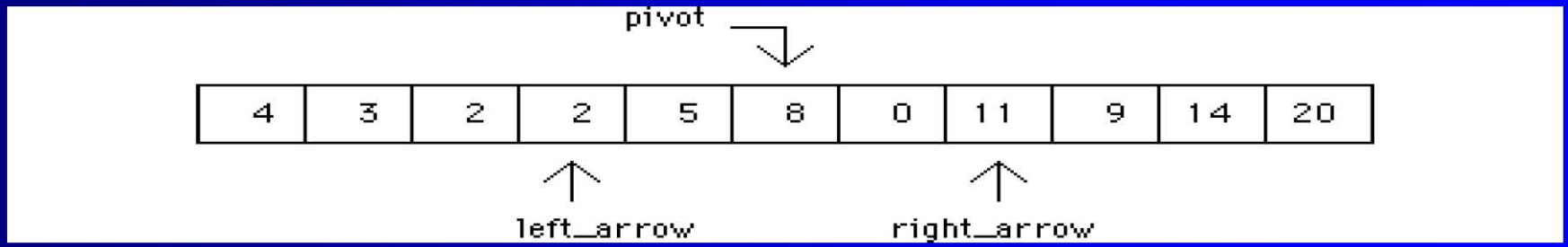
- We continue by moving **`right_arrow`** left to produce



- and moving **`left_arrow`** right yields

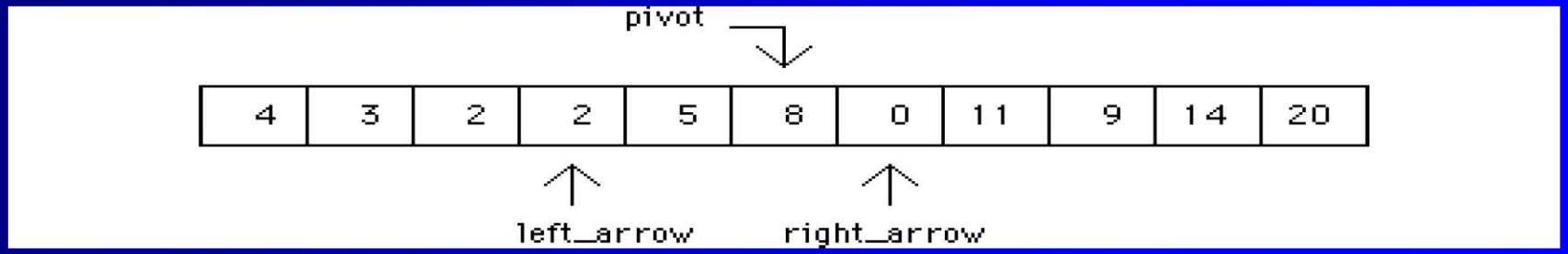


- These values are exchanged to produce

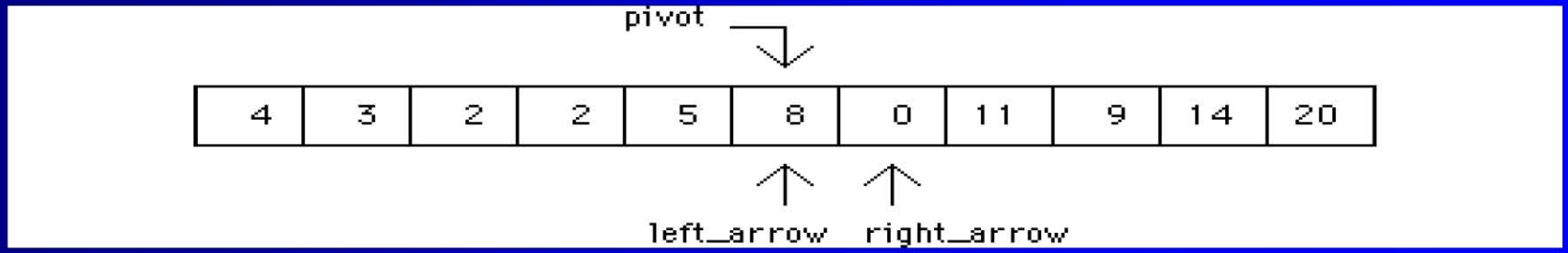


Quick Sort Trace 5

- This process stops when `left_arrow > right_arrow` is `TRUE`. Since this is still `FALSE` at this point, the next `right_arrow` move produces

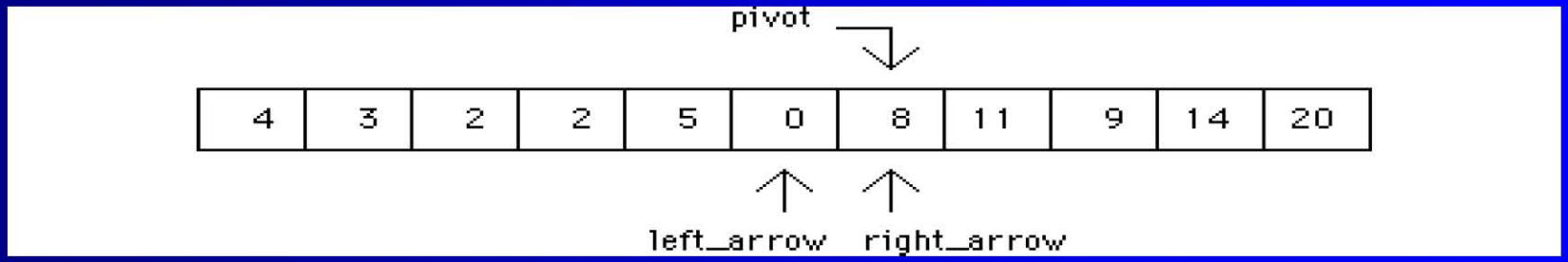


- and the `left_arrow` move to the right yields

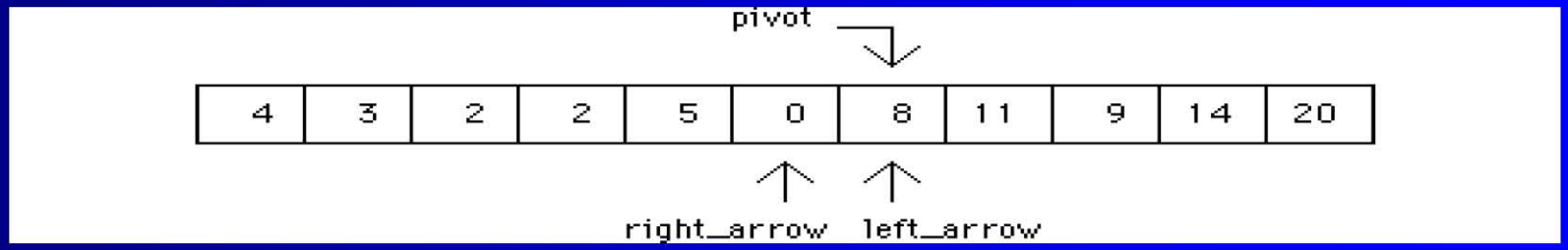


Quick Sort Trace 6

- Because we are looking for a value greater than or equal to **`pivot`** when moving left, **`left_arrow`** stops moving and an exchange is made to produce

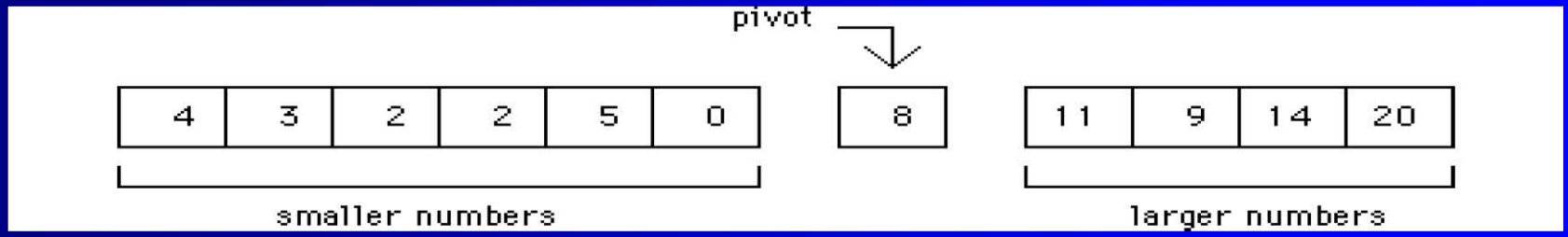


- Notice that the pivot, 8, has been exchanged to occupy a new position. This is acceptable because **`pivot`** is the value of the item, not the index. As before, **`right_arrow`** is moved left and **`left_arrow`** is moved right to produce



Quick Sort Trace 7

- Since **`right_arrow < left_arrow`** is **`TRUE`**, the first subdivision is complete. At this stage, numbers smaller than **`pivot`** are on the left side and numbers larger than **`pivot`** are on the right side. This produces two sublists that can be envisioned as



- Each sublist can now be sorted by the same function. This would require a recursive call to the sorting function. In each case, the vector is passed as a parameter together with the **`right`** and **`left`** indices for the appropriate sublist.

Quick Sort Source Code C ++

```
void QuickSort(apvector<int> &list, int left, int right)
{
    int pivot, leftArrow, rightArrow;
    leftArrow = left;
    rightArrow = right;
    pivot = list[(left + right) / 2];
    do
    {
        while (list[rightArrow] > pivot)
            --rightArrow;
        while (list[leftArrow] < pivot)
            ++leftArrow;
        if (leftArrow <= rightArrow)
        {
            Swap_Data(list[leftArrow], list[rightArrow]);
            ++leftArrow;
            --rightArrow;
        }
    } while (rightArrow >= leftArrow);
    if (left < rightArrow)
        QuickSort(list, left, rightArrow);
    if (leftArrow < right)
        QuickSort(list, leftArrow, right);
}
```

Quick Sort Interface Code C++

- For many of the recursive functions it is customary to start with an "interface" function, since recursive functions call themselves.
- Here is the "interface" for the Quick Sort (this function is called first, which in turn calls the recursive function QuickSort)

```
void Q_Sort(apvector<int> &list)
{
    QuickSort(list, 0, list.length() - 1);
}
```

- You can use logical size - 1 instead of list.length() –1 if the logical and physical sizes are not the same. Don't forget to pass logical size to the Q_Sort function.

Quick Sort uses Swap Data

- Quick Sort uses the same helper function called Swap Data that the Bubble Sort and Selection Sort used.
- The source code can be found on the next slide for your convenience.

Swap Data Source Code C++

```
void Swap_Data (int &number1, int &number2)
{
    int temp;

    temp = number1;
    number1 = number2;
    number2 = temp;
}
```

Big - O Notation

Big - O notation is used to describe the efficiency of a search or sort. The actual time necessary to complete the sort varies according to the speed of your system. Big - O notation is an approximate mathematical formula to determine how many operations are necessary to perform the search or sort. The Big - O notation for the Quick Sort is O(**$nlog_2n$**), because it takes approximately **$nlog_2n$** passes to find the target element.