

CS-254 Divide and Conquer Sorting

Dr. Terry Mason

Sorting

Sorting is the process of re-arranging data items in ascending or descending order.

There are many different sorting algorithms, each with its own properties. We will look at these algorithms:

- ◆ ~~Brute Force Algorithms $O(n^2)$~~
 - ⇒ ~~selection sort~~
 - ~~bubble sort~~
- ◆ ~~Reduce and Conquer $O(n^2)$ best case $O(n)$~~
 - ~~insertion sort~~
- ◆ Divide and Conquer (Better than $O(n^2)$)
 - quick sort
 - merge sort

Quadratic Sorting Algorithms

The first class of sorting algorithms we will study are called **quadratic sorting algorithms** because in the worst case their performance is $O(n^2)$, where n is the size of the array.

Note that for our discussion of sorting algorithms we will sort an array of integers in ascending order (from smallest to largest), however all algorithms can be reversed to sort in descending order.

- ◆ Further, we can modify our sorting algorithms to be template functions which would allow us to sort any data type that overloads the "<" operator.

The name of our array will be `data` with indices from $0..n-1$ and values `data[0]`, `data[1]`, ..., `data[n-1]`.

Sorting

Divide and Conquer

To achieve faster sorting than $O(n^2)$, we will have to use **divide and conquer** techniques. These algorithms are all recursive in nature. The basic strategy is:

- ◆ If the problem is trivial then solve it (base case)
- ◆ Otherwise break the problem into smaller subproblems, solve the subproblems, then combine the solutions to solve the initial problem.

We will study two such algorithms: Merge Sort and Quick Sort.

- ◆ They both use the same basic strategy but differ on their decisions on how to divide into subproblems and combine the subproblem results.

Merge Sort

Merge sort is a recursive sorting algorithm that divides the array near its midpoint, sorts the two half-arrays by recursive calls, and then merges the two halves to get a new sorted array of elements.

Merge sort algorithm:

- ◆ If array is of size 0 or 1, then it is sorted. (base case)
- ◆ Otherwise: (recursive case)
 - Partition array of size n into two equal parts (approximately of size $n/2$).
 - Sort each of the two parts independently.
 - Merge the two sorted parts to create a sorted array.
- ◆ Split formula:
 - Bottom half indices: 0 to $n/2-1$, top half indices: $n/2$ to $n-1$


Merge Sort Code

```
void mergeSort(int data[], int n)
{
    int n1;    // Size of first subarray
    int n2;    // Size of second subarray

    if (n > 1)
    {
        n1 = n/2;
        n2 = n - n1;

        mergeSort(data, n1);    // Sort data[0]..data[n1-1]
        mergeSort((data+n1), n2); // Sort data[n1]..data[n-1]

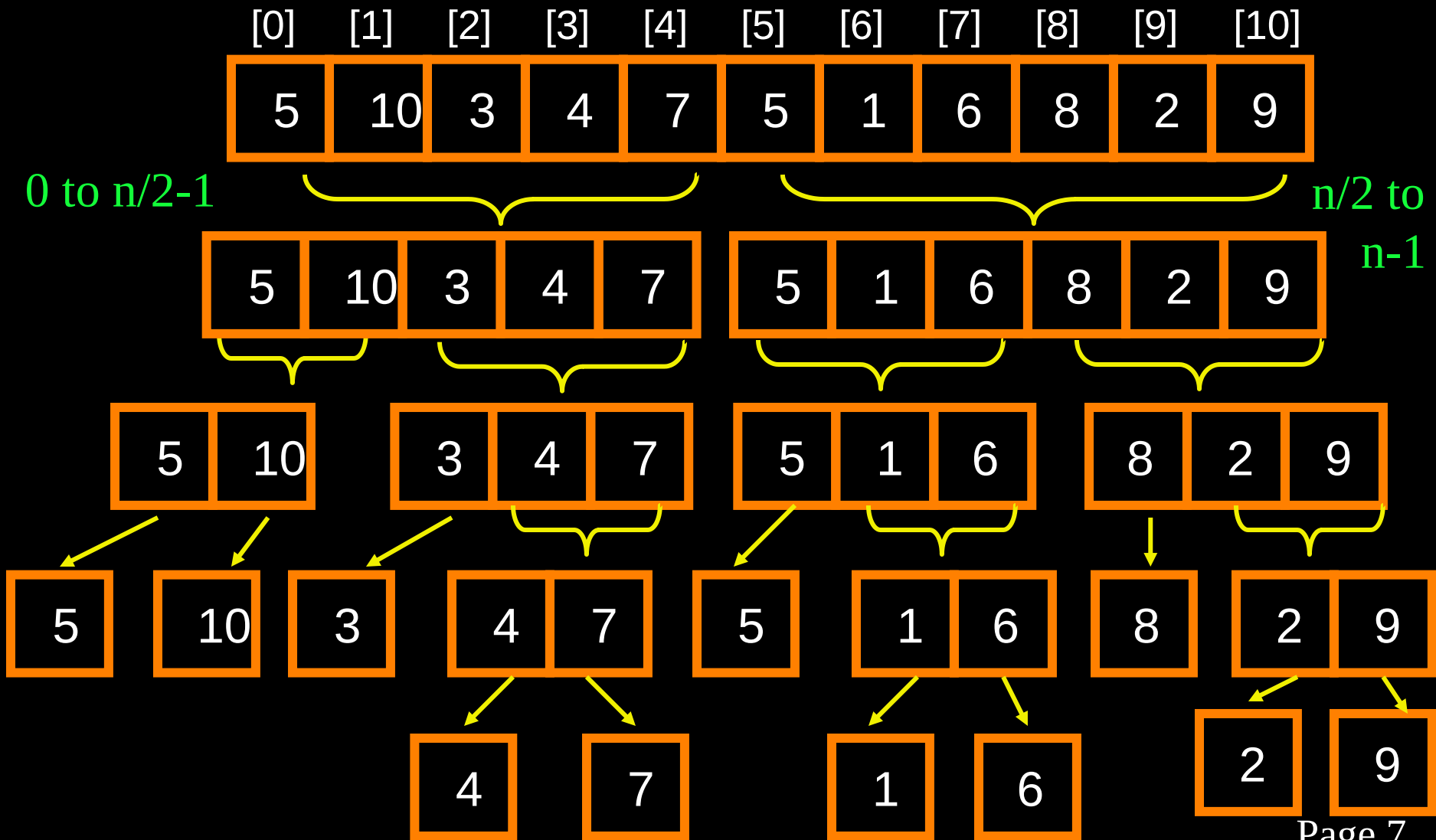
        // Merge the two sorted halves
        merge(data, n1, n2);
    }
}
```



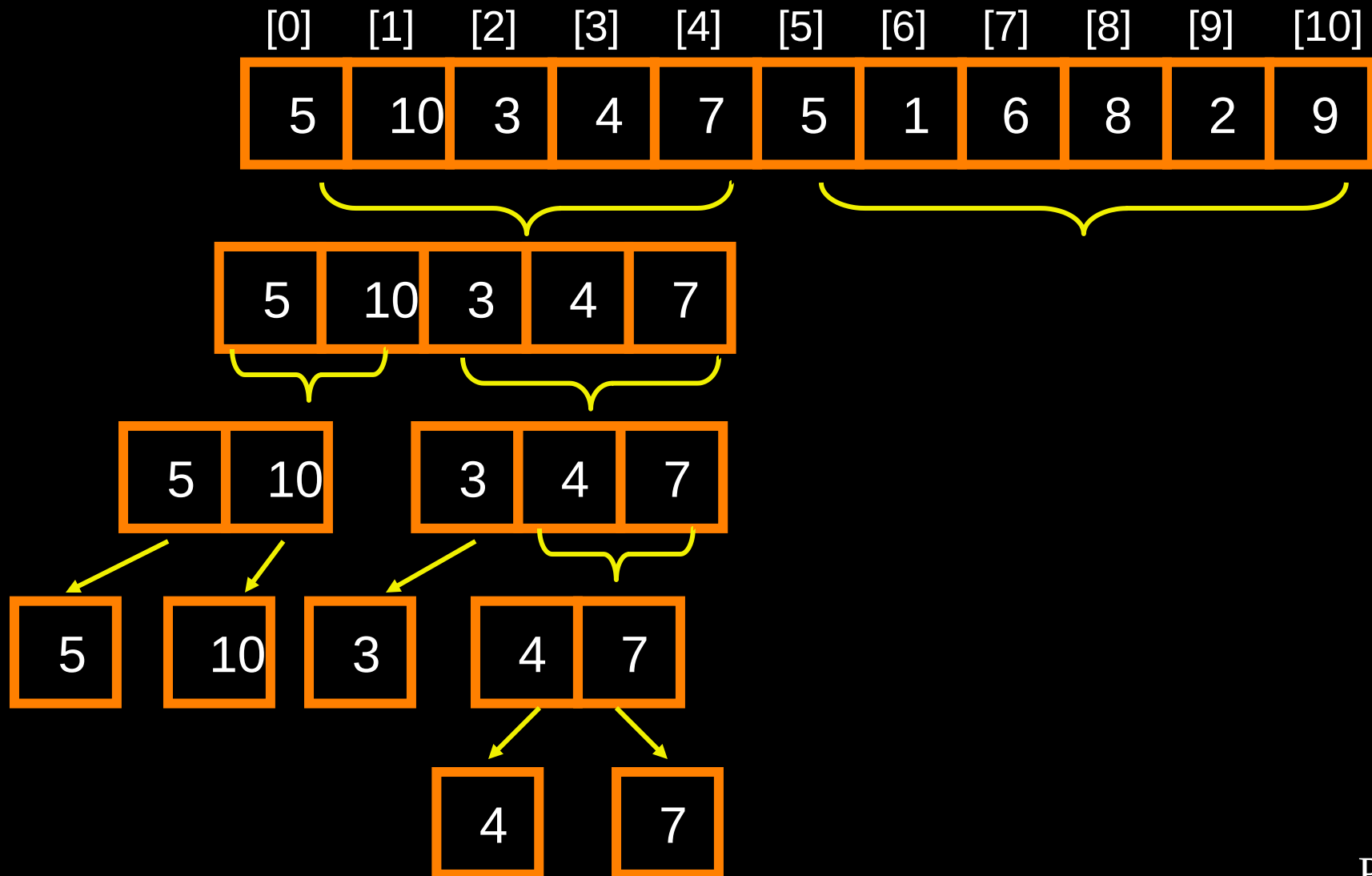
Pointer arithmetic:
This points to element `data[n1]`, but will be treated as a pointer to first element of a subarray.

Merge Sort

Splitting Procedure

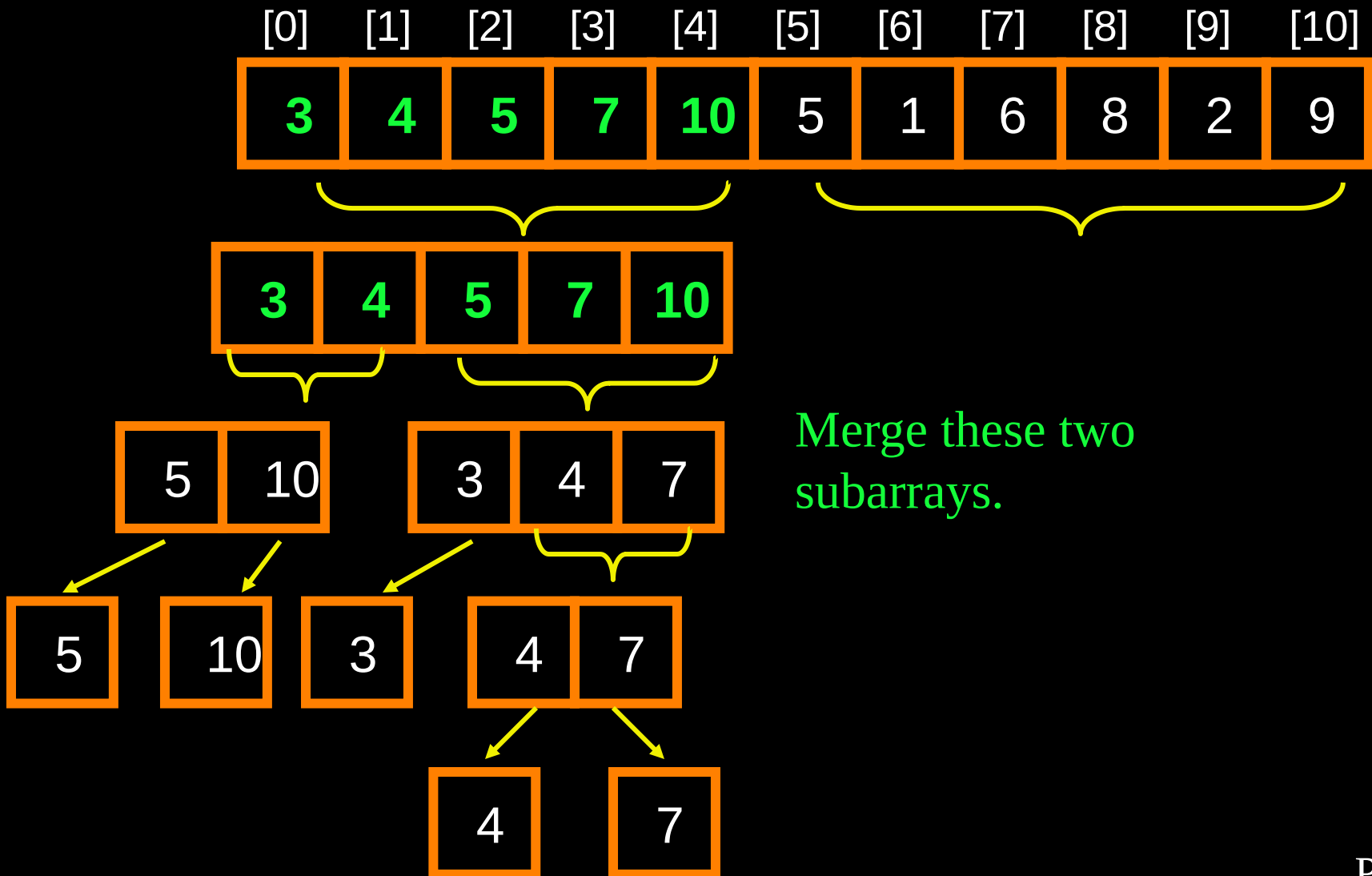


Merge Sort Example



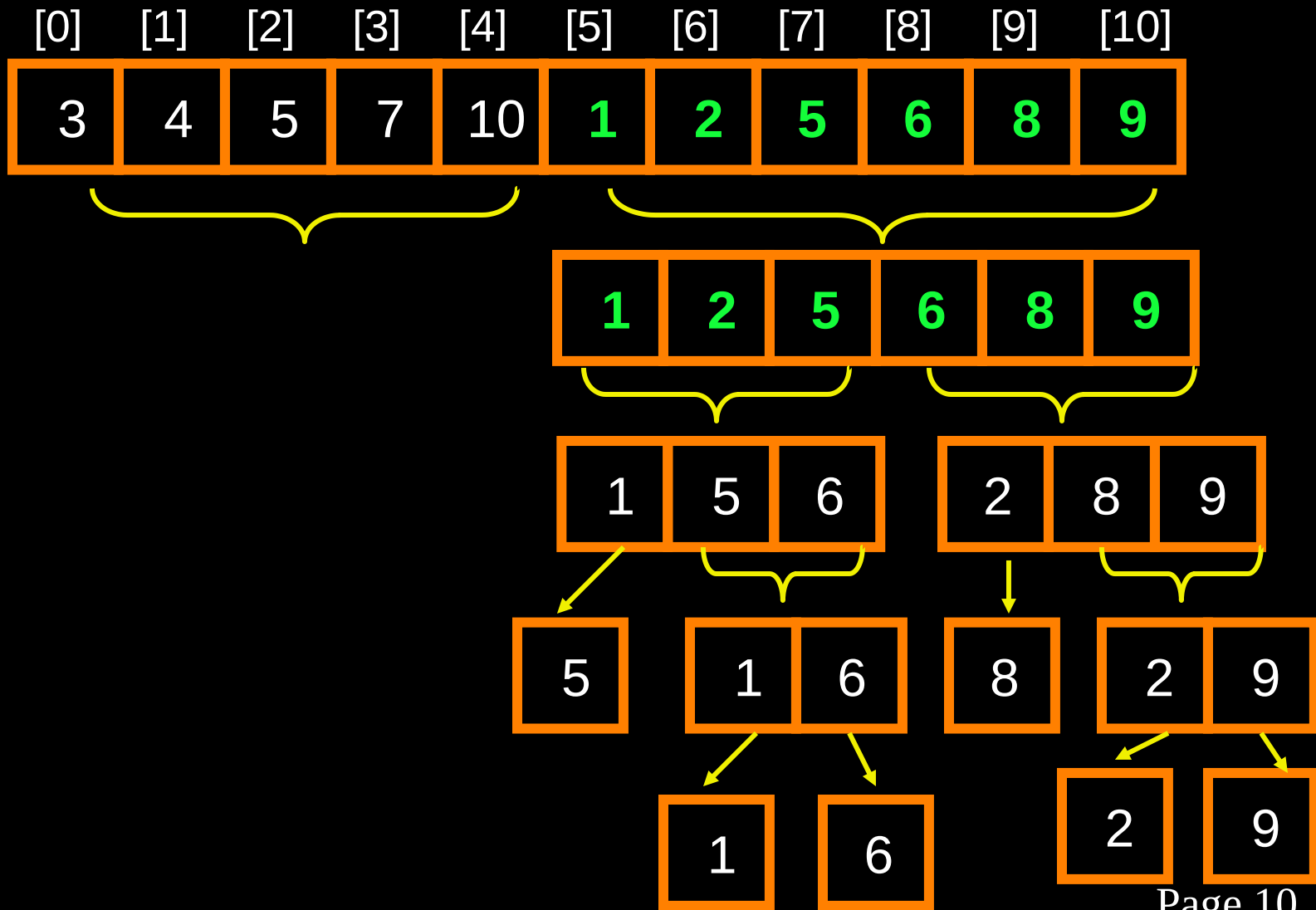
Merge Sort

Example (2)



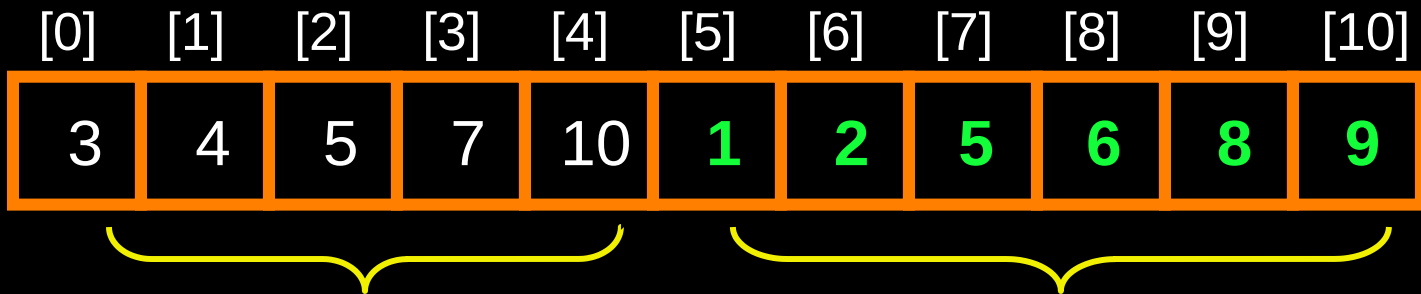
Merge Sort

Example (3)



Merge Sort

Example (4)



Final Merge.



Merge Function

The difficult component of the merge sort is writing the merge function that takes two sorted arrays and combines them into a larger sorted array.

Merge algorithm:

- ◆ Create a new temporary array that will store the sorted result.
- ◆ Keep track of the # of elements that we have copied from each subarray into the temporary array.
- ◆ Take the smallest element in either array and copy it to the temporary array. Update the copied counter for this array.
- ◆ Continue copying elements until one array has no elements left to copy. Then, copy all the remaining elements from this array.
- ◆ Finally, copy the sorted temporary array back to original array, and delete dynamic memory used for temporary array.

Merge Method for Merge Sort

```
void merge(int data[], int n1, int n2)
// Precondition: data is an array of size n1+n2
// data[0]..data[n1-1] are sorted in ascending order, and
// data[n1]..data[n1+n2-1] are sorted as well
// Postcondition: data[0]..data[n1+n2-1] is sorted (asc.)
{
    int *temp;           // Dynamic temp. array
    int copied  = 0; // # elements copied to temp
    int copied1 = 0; // # copied from first array
    int copied2 = 0; // # copied from second array
    int i;             // Array index to copy from temp
                        // back into data

    // Allocate memory for the temporary dynamic array.
    temp = new int[n1+n2];
```

Merge Method for Merge Sort (2)

```
// Merge elements, copying to the temporary array.
while ((copied1 < n1) && (copied2 < n2))
{
    if (data[copied1] < (data + n1)[copied2])
        temp[copied++] = data[copied1++]; // First
    else
        temp[copied++] = (data + n1)[copied2++]; // 2nd
}

// Copy any remaining entries in left or right parts.
while (copied1 < n1)
    temp[copied++] = data[copied1++];
while (copied2 < n2)
    temp[copied++] = (data+n1)[copied2++];
```

Merge Method for Merge Sort (3)

```
// Copy from temp back to the data array
//    and release temp's memory.
for (i = 0; i < n1+n2; i++)
    data[i] = temp[i];
delete [] temp;
}
```

Merge Sort Analysis

Merge Sort analysis:

	<u>Swaps</u>	<u>Comparisons</u>
Best case	$n \cdot \log_2 n$	$n \cdot \log_2 n$
Worst case	$n \cdot \log_2 n$	$n \cdot \log_2 n$

Thus, merge sort performs $n \cdot \log_2 n$ swaps and comparisons **regardless** of the input array and is **$O(n \cdot \log_2 n)$** .

- ◆ Note that a weakness of merge sort is that it also requires an additional array of size n for use in the merge procedure.

Merge sort is faster than the quadratic sorts, but quick sort and heap sort are faster. However, merge sort is useful for external sorting where the entire file cannot fit in memory.

Quick Sort

Quick sort is a recursive sorting algorithm that divides the array into two subarrays, sorts the two half-arrays by recursive calls, and then merges the two halves to get a new sorted array of elements.

- ◆ Quick sort is similar to merge sort except that dividing the array is more complicated, but merging is easier.

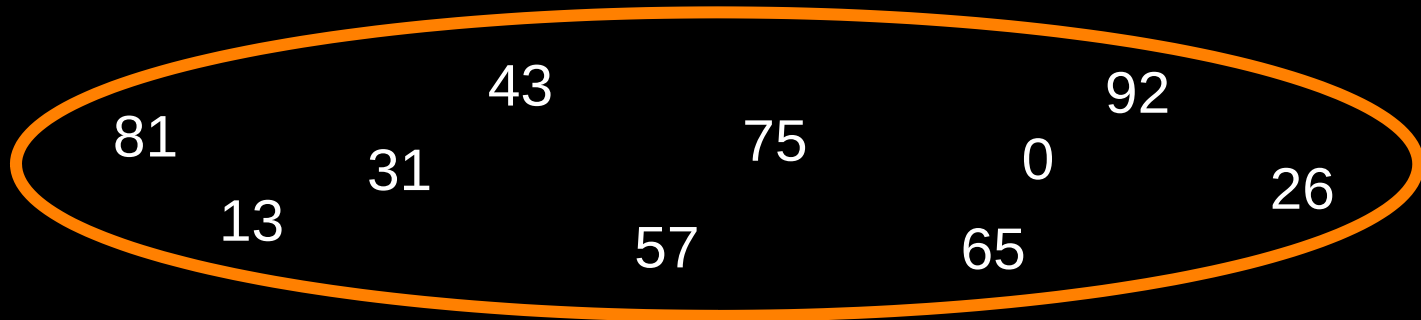
The basic idea behind quick sort is:

- ◆ Pick a value that belongs near the middle of the array.
- ◆ Take this value (called the pivot) and put it at the middle of the array. Then, partition the remaining data, so that values less than the pivot are in smaller indices and those larger than the pivot are in larger indices.

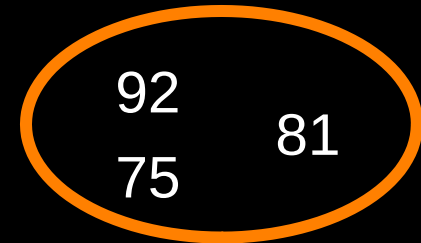
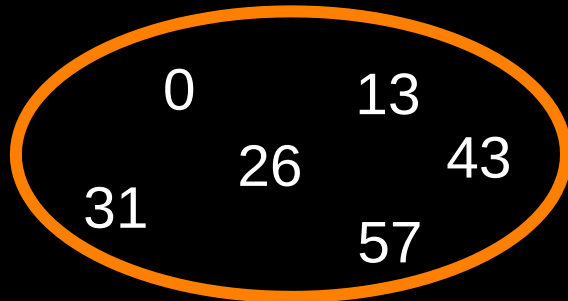
Recursively select the pivot and perform the partitioning until the array is sorted.

Quick Sort Partitioning

By example



Select pivot
Partition



Quick Sort Algorithm

Quick sort algorithm:

- ◆ Select a control key from among $\text{data}[0].. \text{data}[n-1]$.
 - ▮ Let K be the control key or the pivot value.
- ◆ Partition $\text{data}[0].. \text{data}[n-1]$ into three sets:
 - ▮ $A = \{\text{All elements with key} < K\}$
 - ▮ $B = \{K\}$
 - ▮ $C = \{\text{All elements with key} > K\}$
- ◆ Sort sets A and C . (recursively)

Quick Sort Code

```
void quickSort(int data[], int n)
{   int n1;           // Size of 1st subarray (small)
    int n2;           // Size of 2nd subarray (large)
    int pivot_index;  // Index of pivot

    if (n > 1)
    { // partition() selects a pivot and partitions array
      partition(data, n, pivot_index);

      // Compute sizes of subarrays
      n1 = pivot_index;
      n2 = n - n1 - 1; // -1 removes the pivot
      // Recursive calls to sort subarrays
      quickSort(data, n1); // Sort data[0]..data[n1-1]
      quickSort((data + pivot_index + 1), n2); // n1+1 to n-1
    }
}
```

Quick Sort Partition Algorithm

To this point, we have said nothing about how to partition the array and select an appropriate pivot value.

We want the pivot value to be near the middle of the sorted array, but there is no way to know for sure until the array is actually sorted.

- ◆ For now, we will just use the first element of the array as the pivot value.

The partition algorithm will move all values $<$ pivot to the front of the array, and all values $>$ pivot to the end of the array.

- ◆ However, since the pivot may not belong in the exact middle, we do not know where the dividing line between the small part and the large part of the array will be. Thus, we work inward from the two ends of the array.

Quick Sort Partition Algorithm (2)

Partition algorithm idea:

- ◆ Work inward from the two ends of the array.
- ◆ Move smaller elements to the beginning, larger elements to the end.
- ◆ Thus, one segment of smaller elements grows from the left, and another segment of larger elements grows from the right.
- ◆ When these two growing segments meet, the array has been partitioned.
- ◆ The pivot element is then inserted at the boundary of these two segments.

Quick Sort Partition Algorithm (3)

Partition algorithm:

- ◆ Keep track of location on small side **Lo** and location on large side, **Hi**.
- ◆ while (**Lo** < **Hi**)
 - ▮ Scan array forward from Lo to find location of element that is > pivot.
(data[Lo] > pivot)
 - ▮ Scan array backward from Hi to find location of element that is < pivot.
(data[Hi] < pivot)
 - ▮ Swap data[Lo] and data[Hi].
- ◆ Insert pivot at index Hi. That is, data[Hi]=pivot;

Quick Sort

Partitioning Example

Take data[0]=5
as pivot.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
5	10	3	4	7	5	1	6	8	2	9
Lo					Hi					

data[1] > pivot

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
5	10	3	4	7	5	1	6	8	2	9
Lo					Hi					

data[9] < pivot

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
5	10	3	4	7	5	1	6	8	2	9
Lo					Hi					

Quick Sort

Partitioning Example (2)

Swap data[1]
and data[9].

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
5	2	3	4	7	5	1	6	8	10	9
Lo					Hi					

data[4] > pivot

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
5	2	3	4	7	5	1	6	8	10	9
Lo					Hi					

data[6] < pivot

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
5	2	3	4	7	5	1	6	8	10	9
Lo					Hi					

Quick Sort

Partitioning Example (3)

Swap data[4]
and data[6].

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
5	2	3	4	1	5	7	6	8	10	9

Lo

Hi

No more
values < pivot.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
5	2	3	4	1	5	7	6	8	10	9

Hi

Lo

Hi stops at
data[5].

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
5	2	3	4	1	5	7	6	8	10	9

Hi

Lo

Quick Sort

Partitioning Example (4)

Swap data[5]
with data[0]
(pivot).
In this case,
has no effect.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
5	2	3	4	1	5	7	6	8	10	9
					Hi	Lo				

Quick Sort Partition Code

```
void partition(int data[], int n, int &pivot_index)
{
    int Hi;          // Last index in big array
    int Lo;          // Last index in small array
    int pivot;        // Pivot value
    pivot = data[0];
    Lo = 1; Hi = n - 1;
    do {
        while (Lo < n && data[Lo] <= pivot) Lo++;

        while (data[Hi] > pivot) Hi--;
        if (Lo < Hi)
            swap(data[Lo], data[Hi]);
    } while (Lo <= Hi);
    pivot_index = Hi;
    data[0] = data[pivot_index];
    data[pivot_index] = pivot;
}
```

Quick Sort Analysis

Quick sort analysis:

	<u>Swaps</u>	<u>Comparisons</u>
Best case	n	$n \cdot \log_2 n$
Worst case	n^2	n^2
Average case	$n \cdot \log_2 n$	$n \cdot \log_2 n$

Thus, quick sort in the worse case is $O(n^2)$, although on average it is $O(n \cdot \log_2 n)$.

Although in the worst case, quick sort is $O(n^2)$, on average it is one of the fastest main memory sorts. Minimizing the potential for worst case behavior requires selecting a good pivot on every recursive call.

Quick Sort Analysis

Selecting a Pivot

The worst case for quick sort occurs when the pivot we select does not partition the array at all.

- ◆ The pivot is either the largest or the smallest value in the part of the array that is unsorted.

Note that selecting the first index as a pivot leads to the worse case when the array is sorted.

- ◆ That is not good at all!

Other possible pivot selection strategies include:

- ◆ 1) Select the middle element of the array = $n/2$.
- ◆ 2) "Median of three rule" - use the middle value of `data[0]`, `data[n/2]`, and `data[n-1]` as the pivot.
- ◆ 3) Use one or more randomly selected elements from the array.

Sorting Summary

There are many different sorting algorithms, each of which has its own benefits and shortcomings.

Quadratic sorts are $O(n^2)$ and include selection sort, insertion sort, and bubble sort.

- ◆ Insertion sort is the fastest of the quadratic sorts.

Faster sorting can be achieved by using divide and conquer techniques. Quick sort and merge sort achieve $O(n \log_2 n)$ performance on average.