CSC 225 - Summer 2019

Sorting I

Bill Bird

Department of Computer Science
University of Victoria

June 4, 2019

Sorting Problems

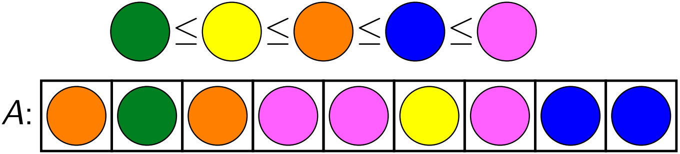
SORTARRAY

Input:	An array A of comparable values.
Result:	The input array A is rearranged into sorted order.

SORTLIST

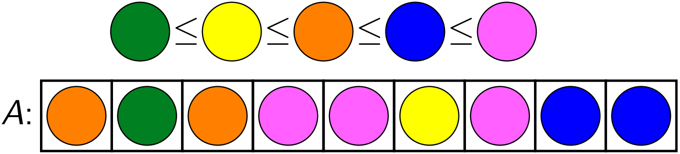
Input:	A linked list L of comparable values.
Output:	A list containing the elements of L in sorted order.

Comparison Sorting (1)



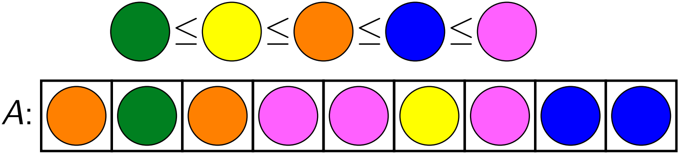
For now, we are interested in algorithms which sort any *comparable* data. We may know nothing about the data besides the result of comparisons like $a \leq b$.

Comparison Sorting (2)



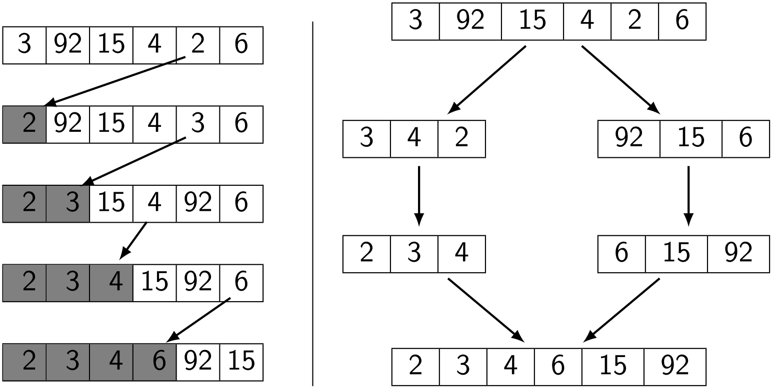
Algorithms which rely on comparisons like $a \leq b$ to rearrange their input data are called **comparison sorting** algorithms.

Comparison Sorting (3)



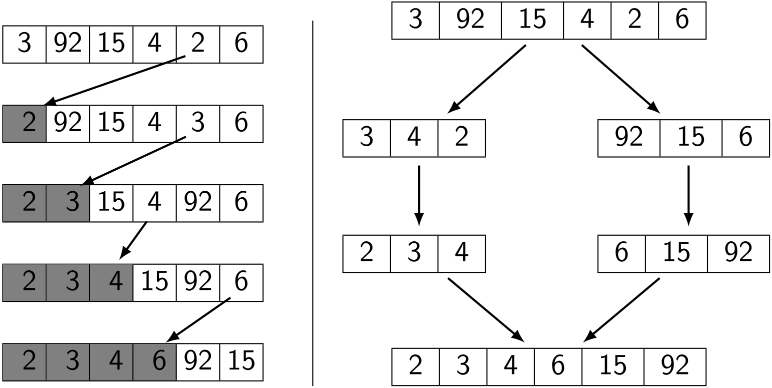
However, it is easier to visualize relationships between integers, so the examples in these slides will use integers. We will see integer-specific sorting methods later in the course.

Sorting (1)



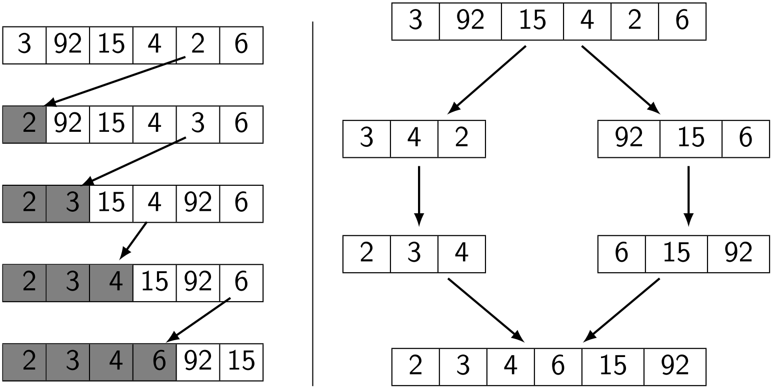
This lecture will cover two algorithmic approaches to general purpose sorting.

Sorting (2)



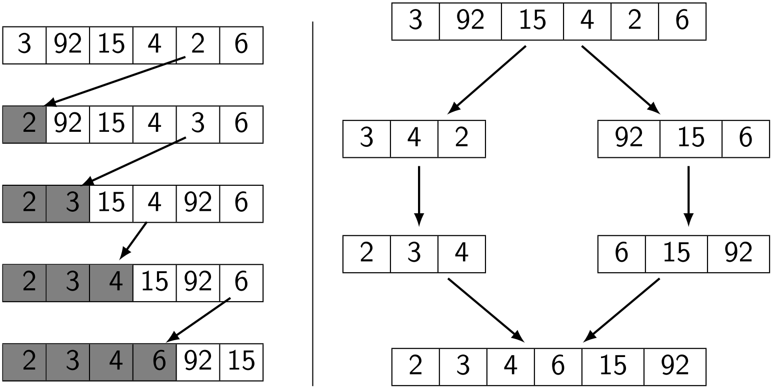
Selection Approach: Sort the array by moving one element at a time into its sorted position.

Sorting (3)



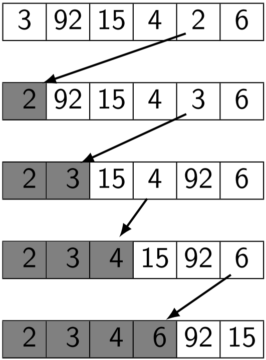
Selection Sort (left) uses the selection approach.

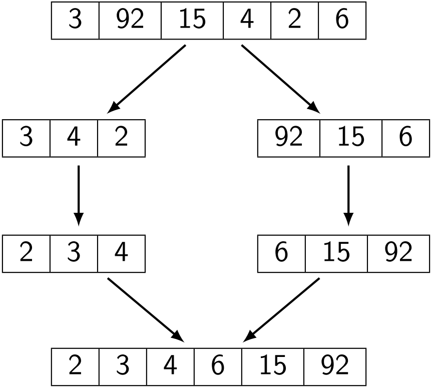
Sorting (4)



Divide and Conquer Approach: Sort the array by dividing it into parts, sorting the parts, then combining the sorted parts together.

Sorting (5)





Quicksort (right) uses the divide and conquer approach.

Sorting Algorithms

	Running Time		
	Best Case	**Expected Case**	**Worst Case**
Selection Based			
Heap Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Divide and Conquer			
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Quicksort	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$
Other			
Bubble Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Radix Sort[1]	$\Theta(dn + b)$	$\Theta(dn + b)$	$\Theta(dn + b)$

[1] Integers only: d-digit values in base b

Sorting Algorithms

	Running Time		
	Best Case	**Expected Case**	**Worst Case**
Selection Based			
Heap Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Divide and Conquer			
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Quicksort	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$
Other			
Bubble Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Radix Sort[1]	$\Theta(dn + b)$	$\Theta(dn + b)$	$\Theta(dn + b)$

[1] Integers only: d-digit values in base b

Selection Sort (1)

Iteration	Unsorted Array					
0	9	16	1	25	4	36
1	**1**	16	9	25	4	36
2	**1**	**4**	9	25	16	36
3	**1**	**4**	**9**	25	16	36
4	**1**	**4**	**9**	**16**	25	36
5	**1**	**4**	**9**	**16**	**25**	36
6	**1**	**4**	**9**	**16**	**25**	**36**

Sorted Array

Selection sort repeatedly finds the minimum element in the array and moves it to the front (by swapping). At the end of iteration i, the i^{th} element of the sorted array is in position.

Selection Sort (2)

```
procedure SelectionSortIterative(A, n)
    for i ← 0, ..., n − 2 do
        min ← i
        for j ← i + 1, ..., n − 1 do
            if A[j] < A[min] then
                min ← j
            end if
        end for
        if min ≠ i then
            Swap A[min] and A[i]
        end if
    end for
end procedure
```

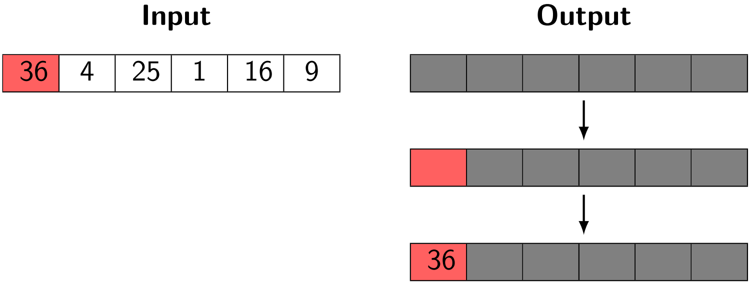
Selection sort is $\Theta(n^2)$ in all cases (notice that the contents of the array have no influence over the loop bounds).

Sorting Algorithms

	Running Time		
	Best Case	**Expected Case**	**Worst Case**
Selection Based			
Heap Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Divide and Conquer			
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Quicksort	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$
Other			
Bubble Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Radix Sort[1]	$\Theta(dn + b)$	$\Theta(dn + b)$	$\Theta(dn + b)$

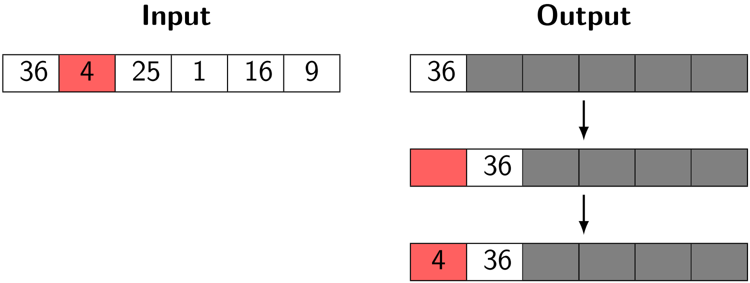
[1] Integers only: d-digit values in base b

Insertion Sort (1)



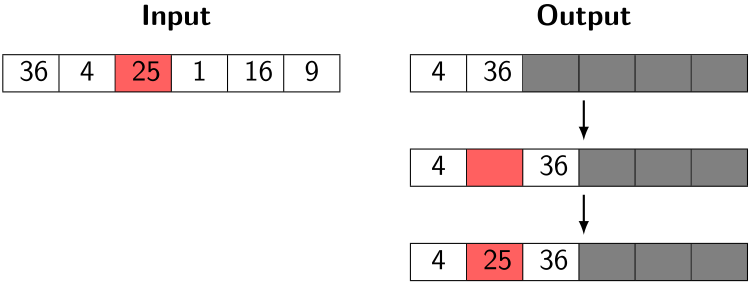
Insertion sort constructs the sorted ordering from the input array by inserting each element in sorted order.

Insertion Sort (2)



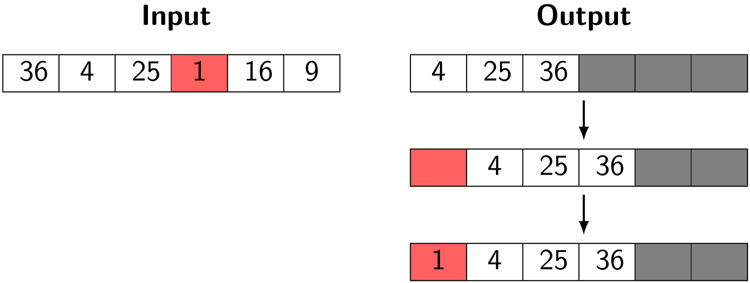
Insertion sort constructs the sorted ordering from the input array by inserting each element in sorted order.

Insertion Sort (3)



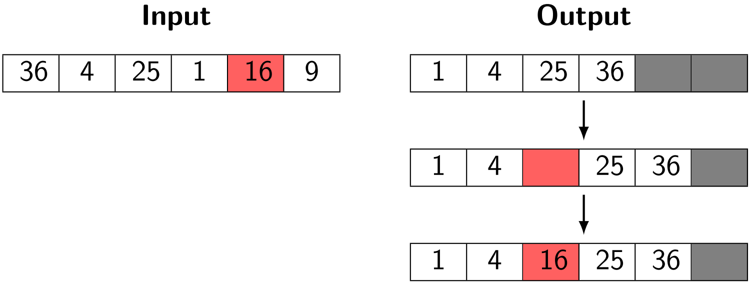
Insertion sort constructs the sorted ordering from the input array by inserting each element in sorted order.

Insertion Sort (4)



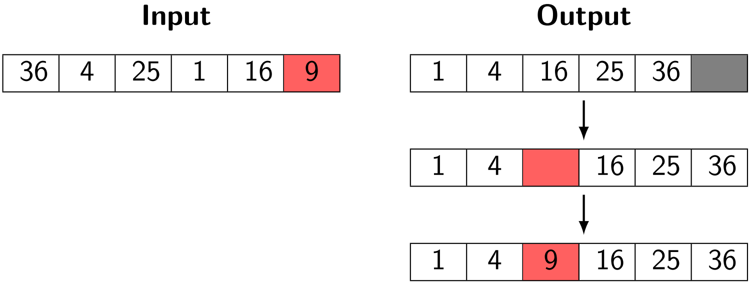
Insertion sort constructs the sorted ordering from the input array by inserting each element in sorted order.

Insertion Sort (5)



Insertion sort constructs the sorted ordering from the input array by inserting each element in sorted order.

Insertion Sort (6)



Insertion sort constructs the sorted ordering from the input array by inserting each element in sorted order.

Insertion Sort (7)

```
procedure InsertionSortIterative(A, n)
    B ← New array of size n
    B[0] ← A[0]
    for i ← 1, ..., n − 1 do
        k ← i
        while k > 0 and B[k − 1] > A[i] do
            B[k] ← B[k − 1]
            k ← k − 1
        end while
        B[k] ← A[i]
    end for
    return B
end procedure
```

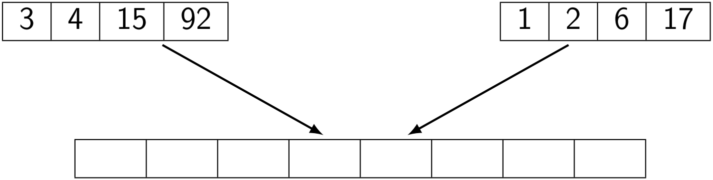
Insertion sort is $\Theta(n^2)$ in the worst case. The actual running time varies depending on the structure of the input array.

Sorting Algorithms

	Running Time		
	Best Case	**Expected Case**	**Worst Case**
Selection Based			
Heap Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Divide and Conquer			
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Quicksort	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$
Other			
Bubble Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Radix Sort[1]	$\Theta(dn + b)$	$\Theta(dn + b)$	$\Theta(dn + b)$

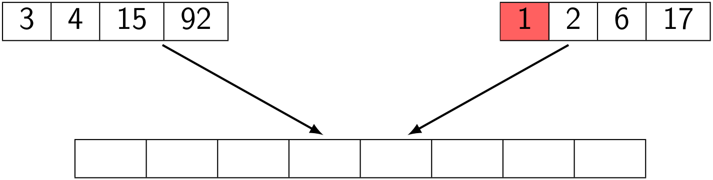
[1] Integers only: d-digit values in base b

Merge Sort (1)



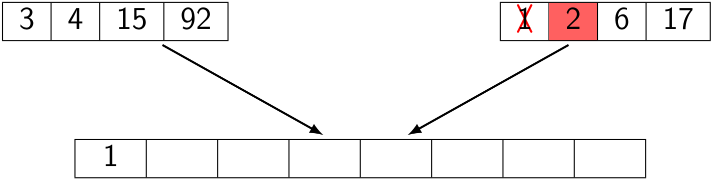
Observation: Two sorted arrays of size $n/2$ can be merged into a single sorted array with a simple linear algorithm.

Merge Sort (2)



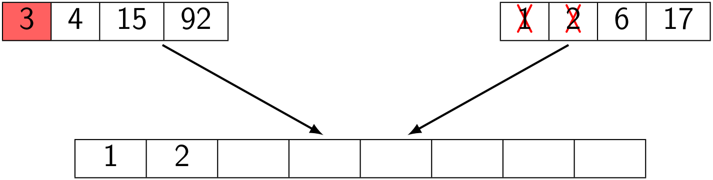
Observation: Two sorted arrays of size $n/2$ can be merged into a single sorted array with a simple linear algorithm.

Merge Sort (3)



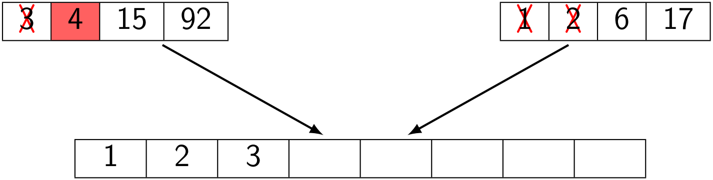
Observation: Two sorted arrays of size $n/2$ can be merged into a single sorted array with a simple linear algorithm.

Merge Sort (4)



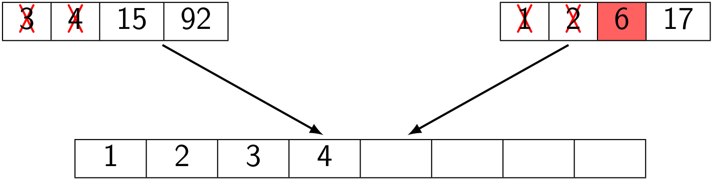
Observation: Two sorted arrays of size $n/2$ can be merged into a single sorted array with a simple linear algorithm.

Merge Sort (5)



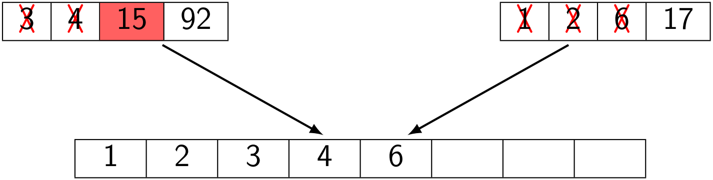
Observation: Two sorted arrays of size $n/2$ can be merged into a single sorted array with a simple linear algorithm.

Merge Sort (6)



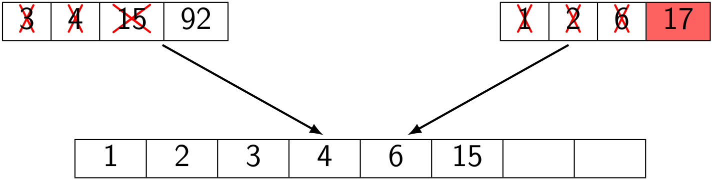
Observation: Two sorted arrays of size $n/2$ can be merged into a single sorted array with a simple linear algorithm.

Merge Sort (7)



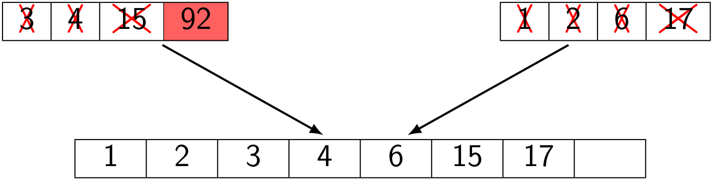
Observation: Two sorted arrays of size $n/2$ can be merged into a single sorted array with a simple linear algorithm.

Merge Sort (8)

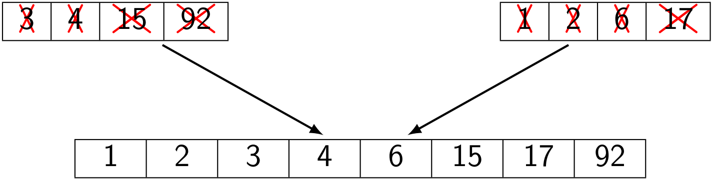


Observation: Two sorted arrays of size $n/2$ can be merged into a single sorted array with a simple linear algorithm.

Merge Sort (9)



Observation: Two sorted arrays of size $n/2$ can be merged into a single sorted array with a simple linear algorithm.



Observation: Two sorted arrays of size $n/2$ can be merged into a single sorted array with a simple linear algorithm.

Merge Sort (11)

```
procedure MERGE(A_1, A_2)
    n_1 ← LENGTH(A_1)
    n_2 ← LENGTH(A_2)
    A ← New array of size n_1 + n_2
    idx1 ← 0
    idx2 ← 0
    idx_out ← 0
    while idx1 < n_1 and idx2 < n_2 do
        if A_1[idx1] < A_2[idx2] then
            A[idx_out] ← A_1[idx1]
            idx1 ← idx1 + 1
        else
            A[idx_out] ← A_2[idx2]
            idx2 ← idx2 + 1
        end if
        idx_out ← idx_out + 1
    end while
    Copy the remainder of A_1 (after index idx1) into A
    Copy the remainder of A_2 (after index idx2) into A
    return A
end procedure
```

The array-based MERGE function above is $\Theta(n_1 + n_2)$.

Merge Sort (12)

```
procedure MERGE(A1, A2)
    n1 ← LENGTH(A1)
    n2 ← LENGTH(A2)
    A ← New array of size n1 + n2
    idx1 ← 0
    idx2 ← 0
    idx_out ← 0
    while idx1 < n1 and idx2 < n2 do
        if A1[idx1] < A2[idx2] then
            A[idx_out] ← A1[idx1]
            idx1 ← idx1 + 1
        else
            A[idx_out] ← A2[idx2]
            idx2 ← idx2 + 1
        end if
        idx_out ← idx_out + 1
    end while
    Copy the remainder of A1 (after index idx1) into A
    Copy the remainder of A2 (after index idx2) into A
    return A
end procedure
```

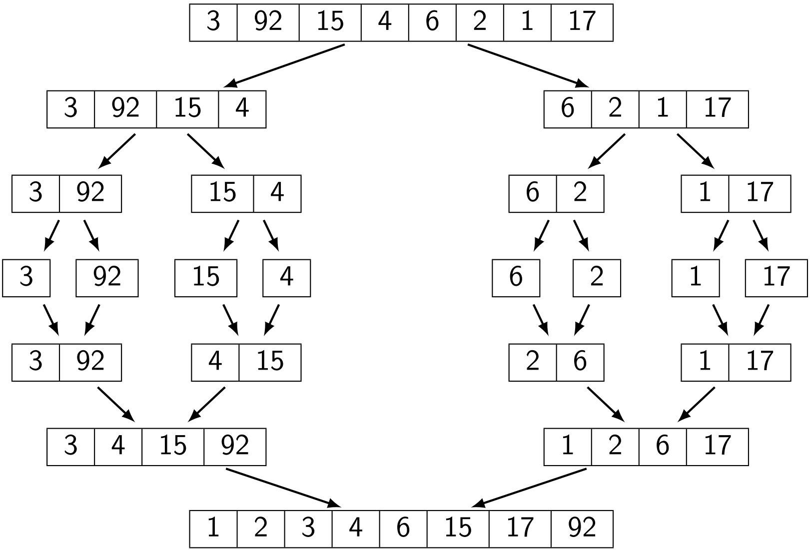
A list-based MERGE (not shown) is significantly more elegant (especially the recursive version).

Merge Sort (13)

```
procedure MergeSort(A)
    n ← Length(A)
    if n = 1 then
        //An array of size 1 is already sorted.
        return
    end if
    n1 ← ⌊n/2⌋
    n2 ← n − n1
    Split A into two arrays A1 (with size n1) and A2 (with size n2).
    MergeSort(A1)
    MergeSort(A2)
    Merge the sorted A1 and A2 together into A.
end procedure
```

Merge sort is a divide and conquer algorithm. The behavior of Merge sort is often illustrated with a diagram called a ‘Merge sort tree’ (see next slide).

Merge Sort (14)



Merge Sort Analysis (1)

```
procedure MergeSort(A)
    n ← Length(A)
    if n = 1 then
        //An array of size 1 is already sorted.
        return
    end if
    n1 ← ⌊n/2⌋
    n2 ← n − n1
    Split A into two arrays A1 (with size n1) and A2 (with size n2).
    MergeSort(A1)
    MergeSort(A2)
    Merge the sorted A1 and A2 together into A.
end procedure
```

Exercise: Find a recurrence for the worst case running time of merge sort.

Merge Sort Analysis (2)

```
procedure MergeSort(A)
    n ← Length(A)
    if n = 1 then
        //An array of size 1 is already sorted.
        return
    end if
    n1 ← ⌊n/2⌋
    n2 ← n − n1
    Split A into two arrays A1 (with size n1) and A2 (with size n2).
    MergeSort(A1)
    MergeSort(A2)
    Merge the sorted A1 and A2 together into A.
end procedure
```

- We can use the size n of the input array A as the parameter.
- It is fairly straightforward to assume that $T(1) = 1$.
- The split and merge steps are both $\Theta(n)$.

Merge Sort Analysis (3)

```
procedure MergeSort(A)
    n ← Length(A)
    if n = 1 then
        //An array of size 1 is already sorted.
        return
    end if
    n1 ← ⌊n/2⌋
    n2 ← n − n1
    Split A into two arrays A1 (with size n1) and A2 (with size n2).
    MergeSort(A1)
    MergeSort(A2)
    Merge the sorted A1 and A2 together into A.
end procedure
```

- Since we don't want to analyse the split and merge steps again, we can leave their exact operation counts as unknowns, giving the recursive case

$$T(n) = 2T(n/2) + c_1 n + c_2 n + 1$$

where c_1 is the constant for the split phase and c_2 is the constant for the merge phase.

Merge Sort Analysis (4)

```
procedure MergeSort(A)
    n ← Length(A)
    if n = 1 then
        //An array of size 1 is already sorted.
        return
    end if
    n1 ← ⌊n/2⌋
    n2 ← n − n1
    Split A into two arrays A1 (with size n1) and A2 (with size n2).
    MergeSort(A1)
    MergeSort(A2)
    Merge the sorted A1 and A2 together into A.
end procedure
```

- Solving this recurrence demonstrates that merge sort is $\Theta(n \log_2 n)$