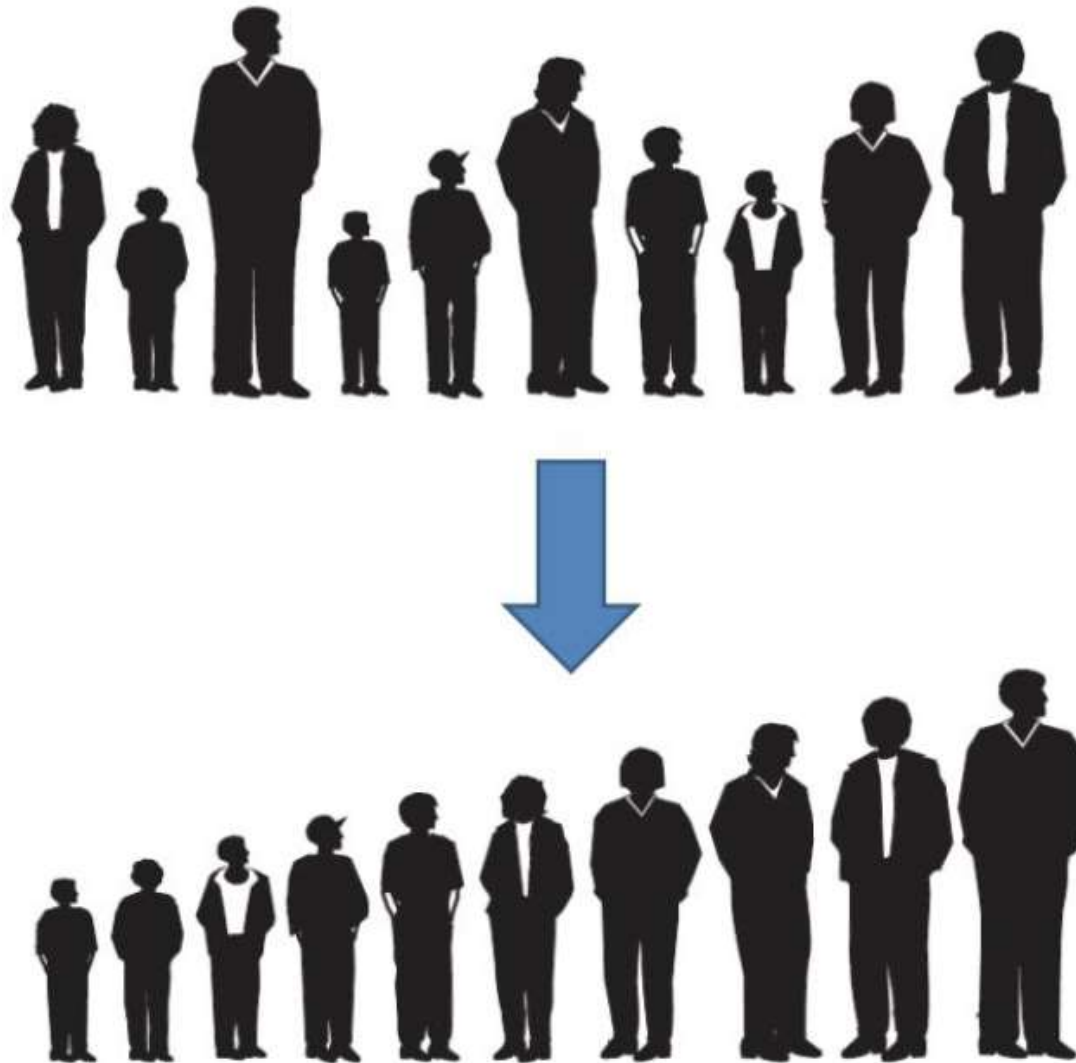


Sorting Algorithms

A Familiar Example



Sorting

- ***Sorting*** is a process that organizes a collection of data into either ascending or descending order.
- As a human being, we can perform this task quickly
 - Spot the tallest person/shortest person right away
 - Simultaneously compare multiple persons
- A computer cannot see the big picture all at once
 - It's limited to compare two numbers at a time, swap or copy them, and move on to the next pair
 - However, computers can do these really fast.

Sorting Algorithms

- There are many sorting algorithms, such as:
 - Selection Sort
 - Insertion Sort
 - Bubble Sort
 - Merge Sort
 - Quick Sort
 - Heapsort
 - Radix sort
 - Counting sort
 - ...

Selection Sort

Intuition:

- Round 1: Find the smallest element
- Round 2: Find the second smallest element
- Round 3: Find the third smallest element
- ...

A list of n elements requires $n-1$ passes over the array to completely rearrange the data.

Selection Sort

- The array is viewed as two sublists, *sorted* and *unsorted*, which are divided by an imaginary wall.
- Find the smallest element from the unsorted sublist and swap it with the element at the beginning of the unsorted data.
- Move the imaginary wall between the two sublists one element ahead.
- Each time we move one element from the unsorted sublist to the sorted sublist, we say that we have completed a sort pass.
- A list of n elements requires $n-1$ passes to completely rearrange the data.

Selection Sort

Sorted

Unsorted

23	78	45	8	32	56
----	----	----	---	----	----

Original List

Selection Sort

Sorted

Unsorted

23	78	45	8	32	56
----	----	----	---	----	----

Original List

8	78	45	23	32	56
---	----	----	----	----	----

After pass 1

8	23	45	78	32	56
---	----	----	----	----	----

After pass 2

8	23	32	78	45	56
---	----	----	----	----	----

After pass 3

8	23	32	45	78	56
---	----	----	----	----	----

After pass 4

8	23	32	45	56	78
---	----	----	----	----	----

After pass 5

Selection Sort Algorithm

```
template <class Item>
void selectionSort( Item a[], int n)
{
    for (int i = 0; i < n-1; i++) {
        int min = i;
        // find the index of the next minimum
        for (int j = i+1; j < n; j++)
            if (a[j] < a[min]) min = j;
        // swap a[i] and a[min]
        Item tmp = a[i];
        a[i] = a[min];
        a[min] = tmp;
    }
}
```

Selection Sort

- Number of swaps?

$O(N)$ in all cases (best case, worst case, average case)

- Number of comparisons?

$$N(N-1)/2 = O(N^2)$$

Selection Sort – Analysis

- The inner for loop executes the size of the unsorted part minus 1 (from 1 to $n-1$), and in each iteration we make one key comparison.
 - ➔ # of key comparisons = $(n-1)+(n-2)+\dots+1 = n*(n-1)/2$
 - ➔ **So, Selection sort is $O(n^2)$**
- The best case, the worst case, and the average case of the selection sort algorithm are same.
 - ➔ all of them are **$O(n^2)$**

Selection Sort – Analysis

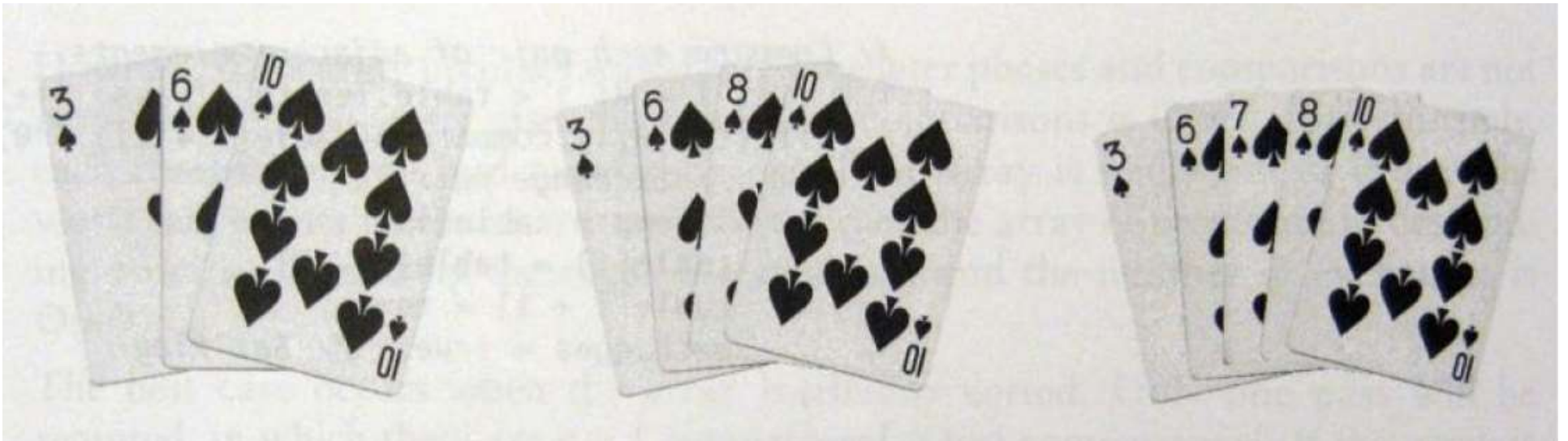
- The best case, the worst case, and the average case of the selection sort algorithm are all are **$O(n^2)$**
 - This means that the behavior of the selection sort algorithm does not depend on the initial organization of data.
 - Since $O(n^2)$ grows so rapidly, the selection sort algorithm is appropriate only for small n .
 - Although the selection sort algorithm requires $O(n^2)$ key comparisons, it only requires $O(n)$ swaps.
 - A selection sort could be a good choice if data moves are costly but key comparisons are not costly (short keys, long records).

Comparison of N , $\log N$ and N^2

<u>N</u>	<u>O(LogN)</u>	<u>O(N²)</u>
16	4	256
64	6	4K
256	8	64K
1,024	10	1M
16,384	14	256M
131,072	17	16G
262,144	18	6.87E+10
524,288	19	2.74E+11
1,048,576	20	1.09E+12
1,073,741,824	30	1.15E+18

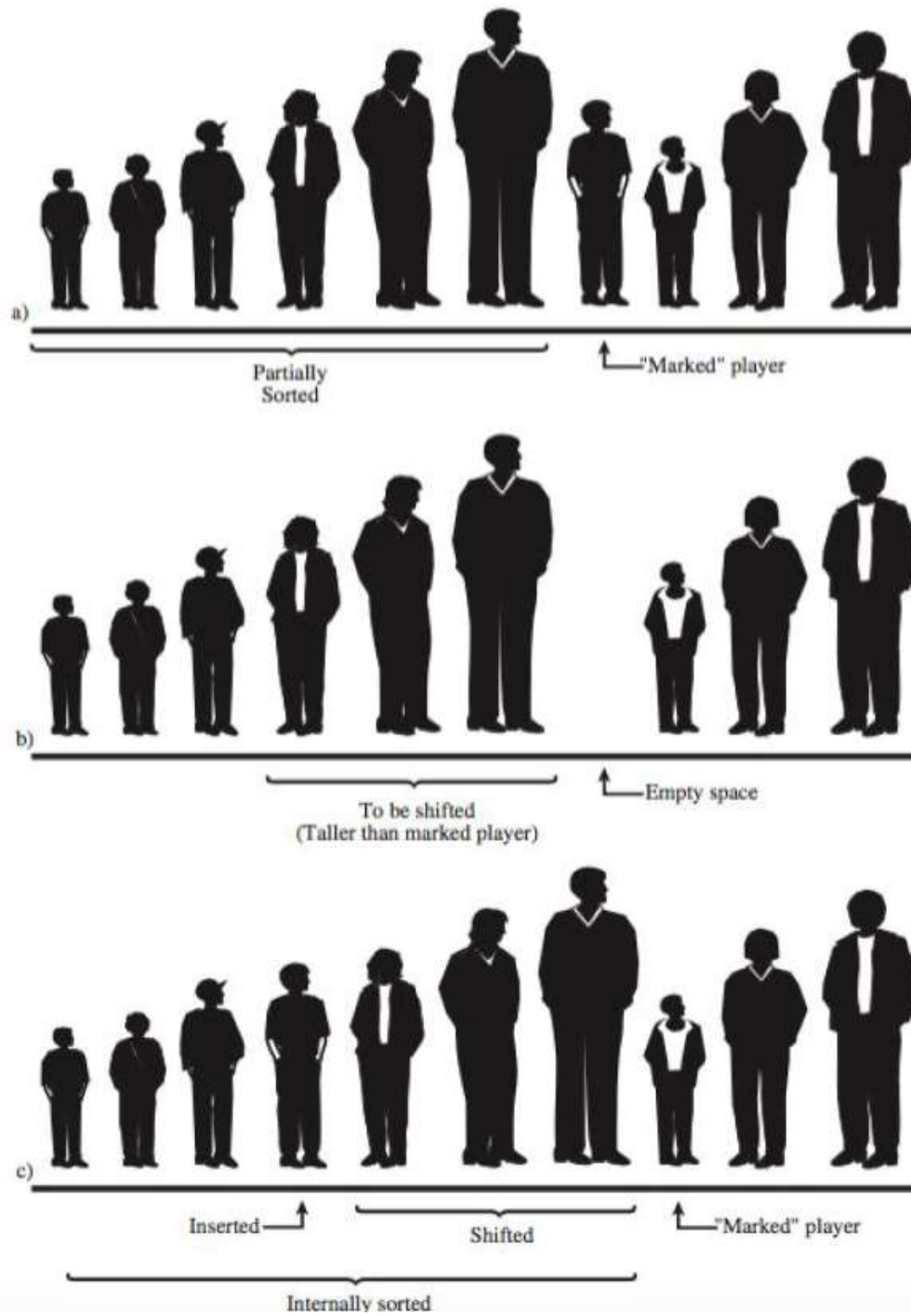
Insertion Sort

How do you sort a set of poker cards?



Insertion Sort

- Insertion sort is a simple sorting algorithm that is appropriate for small inputs.
 - Most common sorting technique used by card players.
- The array is viewed as two parts: sorted and unsorted.
- In each pass, the first element of the unsorted part is picked up, transferred to the sorted sublist, and inserted at the appropriate place.
- A list of n elements will take at most $n-1$ passes to sort the data.



Insertion Sort

Sorted

Unsorted

23	78	45	8	32	56
----	----	----	---	----	----

Original List

Insertion Sort

Sorted

Unsorted

23	78	45	8	32	56
----	----	----	---	----	----

Original List

23	78	45	8	32	56
----	----	----	---	----	----

After pass 1

23	45	78	8	32	56
----	----	----	---	----	----

After pass 2

8	23	45	78	32	56
---	----	----	----	----	----

After pass 3

8	23	32	45	78	56
---	----	----	----	----	----

After pass 4

8	23	32	45	56	78
---	----	----	----	----	----

After pass 5

Insertion Sort Algorithm

```
template <class Item>
void insertionSort(Item a[], int n)
{
    for (int i = 1; i < n; i++)
    {
        Item tmp = a[i];

        for (int j=i; j>0 && tmp < a[j-1]; j--)
            a[j] = a[j-1];
        a[j] = tmp;
    }
}
```

Insertion Sort – Analysis

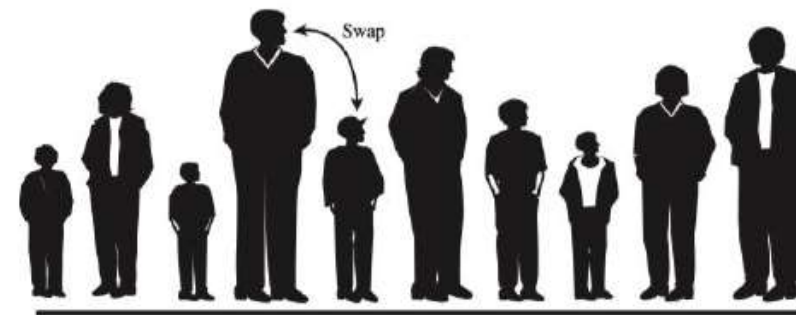
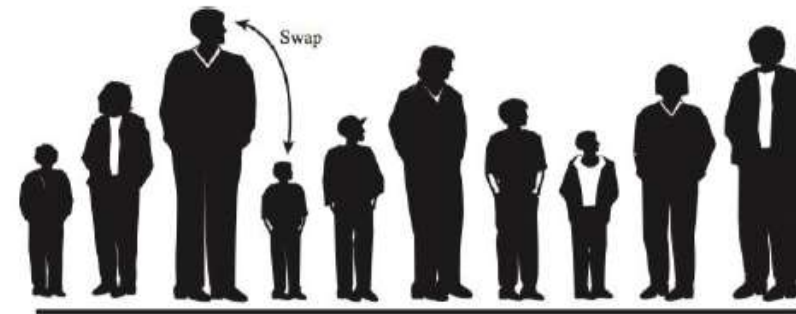
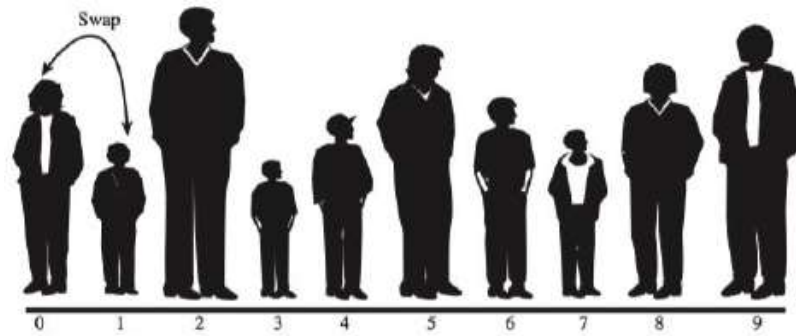
- Running time depends on not only the size of the array but also the contents of the array.
- **Best-case:** $\rightarrow O(n)$
 - Array is already sorted in ascending order.
 - Inner loop body will not be executed.
 - The number of moves: $2*(n-1) \rightarrow O(n)$
 - The number of comparisons: $(n-1) \rightarrow O(n)$
- **Worst-case:** $\rightarrow O(n^2)$
 - Array is in reverse order:
 - Inner loop is executed $i-1$ times, for $i = 1, 2, 3, \dots, n-1$
 - The number of moves: $2*(n-1) + (1+2+\dots+n-1) = 2*(n-1) + n*(n-1)/2 \rightarrow O(n^2)$
 - The number of key comparisons: $(1+2+\dots+n-1) = n*(n-1)/2 \rightarrow O(n^2)$
- Average-case: $\rightarrow O(n^2)$
 - We have to look at all possible initial data organizations.
- **So, Insertion Sort is $O(n^2)$**

Bubble Sort

Intuition

- Round 1: Find the largest element
- Round 2: Find the largest smallest element
- Round 3: Find the largest smallest element
- ...

Bubble sort finds the largest element in each round by repeatedly comparing every two adjacent numbers, and swapping them if the one on the left is larger.



Bubble Sort

- After one pass, we find the largest number in that round



- It is like the 'biggest' bubble floats to the top of the surface, hence the name 'bubble sort'

Bubble Sort

- The array is viewed as two sublists: sorted and unsorted.
- The largest element is bubbled from the unsorted list and moved to the sorted sublist.
- After that, the wall moves one element back, increasing the number of sorted elements and decreasing the number of unsorted ones.
- Each time an element moves from the unsorted part to the sorted part one sort pass is completed.
- Given a list of n elements, bubble sort requires up to $n-1$ passes over the array to sort the data.

Bubble Sort

23	78	45	8	32	56
----	----	----	---	----	----

Original List

Bubble Sort

23	78	45	8	32	56
----	----	----	---	----	----

Original List

23	45	8	32	56	78
----	----	---	----	----	----

After pass 1

23	8	32	45	56	78
----	---	----	----	----	----

After pass 2

8	23	32	45	78	56
---	----	----	----	----	----

After pass 3

8	23	32	45	56	78
---	----	----	----	----	----

After pass 4

Bubble Sort Algorithm

```
template <class Item>
void bubbleSort(Item a[], int n)
{
    bool swapped = true;

    while(swapped) {
        swapped = false;
        for (int i=1; i<n; i++)
            if (a[i-1] > a[i]) {
                Item temp = a[i];
                a[i] = a[i-1];
                a[i-1] = temp;
                swapped = true;           // signal exchange
            }
    }
}
```

Bubble Sort – Analysis

- **Best-case:** $\rightarrow O(n)$
 - Array is already sorted in ascending order.
 - The number of moves: 0 $\rightarrow O(1)$
 - The number of comparisons: $(n-1)$ $\rightarrow O(n)$
- **Worst-case:** $\rightarrow O(n^2)$
 - Array is in reverse order:
 - Outer loop is executed $n-1$ times,
 - The number of moves: $3 \cdot (1+2+\dots+n-1) = 3 \cdot n \cdot (n-1)/2$
 $\rightarrow O(n^2)$
 - The number of comparisons: $n(n-1)$ $\rightarrow O(n^2)$
- **Average-case:** $\rightarrow O(n^2)$
 - We have to look at all possible initial data organizations.
- **So, Bubble Sort is $O(n^2)$**

Recap: $O(N^2)$ Sorting Algorithms

- **Bubble sort** uses repeated comparisons and swaps to find the biggest element in each pass, and positions it toward the end of the array.
(<https://www.youtube.com/watch?v=lyZQPjUT5B4>)
- **Selection sort** reduces the number of swaps by only performing one swap at the end of each pass.
(<https://www.youtube.com/watch?v=Ns4TPTC8whw>)
- **Insertion Sort** eliminates swaps and replaces them with copies, which are 3 times faster
(<https://www.youtube.com/watch?v=ROaIU379I3U>)
- They are all quadratic cost: **$O(N^2)$** in the worst and average cases.

Merge Sort

- Merge sort algorithm is a **divide-and-conquer** sorting algorithm.
- It is a recursive algorithm.
- Cost is $O(N \log N)$, much faster than simple sorting algorithms.
- Requires additional memory space
 - A temporary array as large as the input array
 - So it is not an in-place sorting algorithm

Merging Two Sorted Arrays

- This is the key step in Merge Sort.
- Assume two subsets of the array (Left and Right) are already sorted
- Merge them into array $temp$ such that $temp$ contains all elements from Left and Right, and remains sorted.
- Note the two subsets may have different sizes. In fact one of them may be empty! Must correctly handle all cases!
- Example:

A: 23 47 81 95

B: 7 14 39 55 62 74

A: 23 47 81 95

B: 7 14 39 55 62 74

Merging Two Sorted Arrays

```
const int MAX_SIZE = maximum-number-of-items-in-array;
void merge_TwoArrays(T A[], int sizeA,
                     T B[], int sizeB, T C[]) {
    int ai=0, bi=0, ci=0;    // separate indices for arrays
    while (ai < sizeA && bi < sizeB){
        if (A[ai] < B[bi])
            C[ci++] = A[ai++];
        else
            C[ci++] = B[bi++];
    }
    while (ai < sizeA) C[ci++] = A[ai++];
    while (bi < sizeB) C[ci++] = B[bi++];
}
```

In the actual MergeSort, arrays A and B will be subarrays of the original array. So they will be from the same array, just at different starting locations.

Merging Two Sorted Arrays

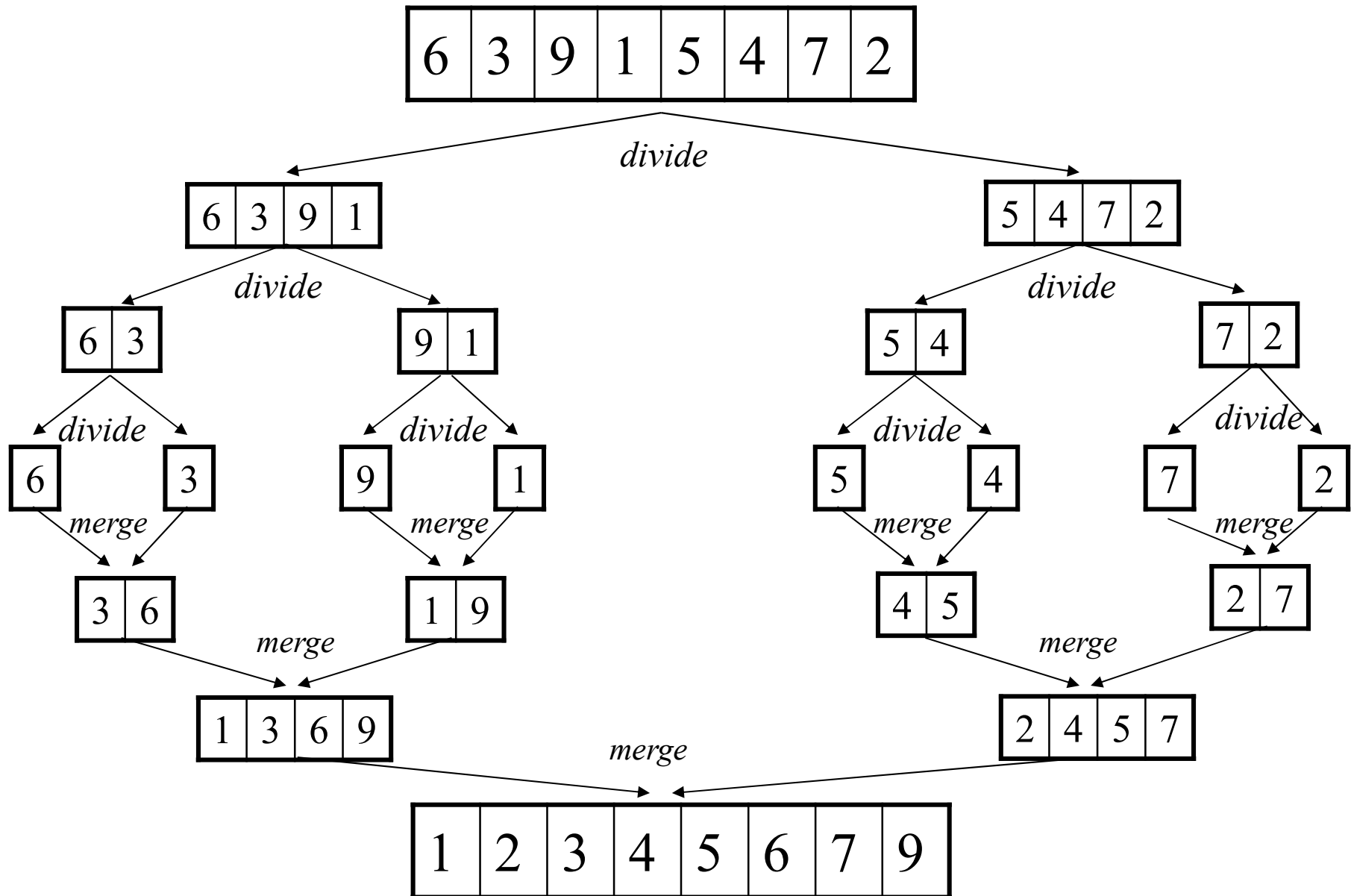
- Is it possible that both Left and Right have remaining elements after the first while loop?
- What happens if Left (A) is empty to begin with?
- How many copy (assignment) instructions?

Size of Left (A) + size of Right (B)

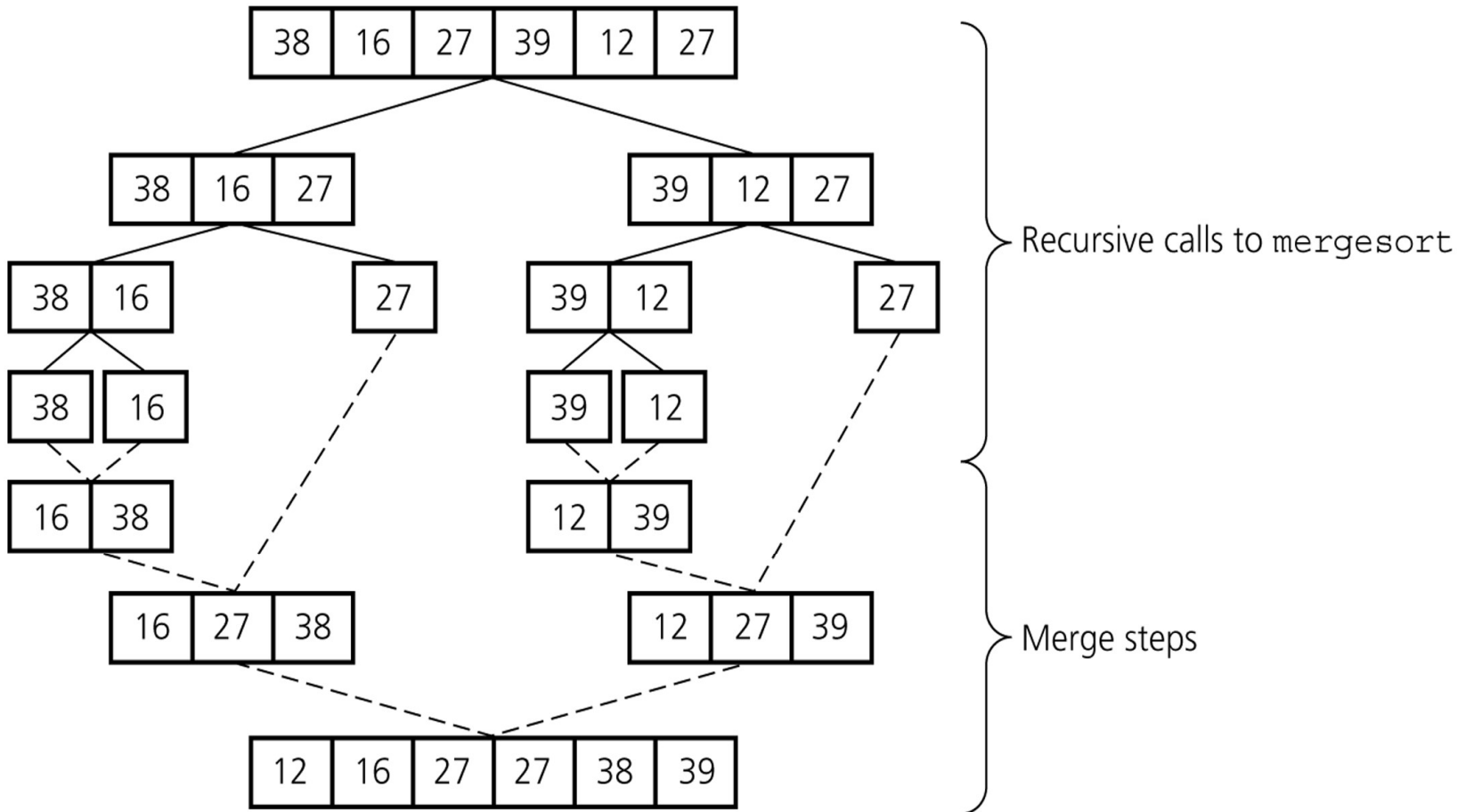
Merge Sort

- Once we know how to do the merge, Merge Sort is quite simple:
 - **Divide** the array into two halves
 - Sort each half (**Conquer**). How? Recursion!
 - Call merge() to merge two halves.
- What's the base case of the recursion?
 - When there is only 1 element left to sort, it's trivially sorted, so return immediately.

Mergesort - Example



Example



Mergesort

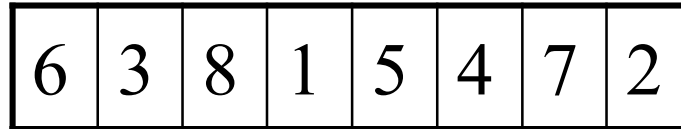
```
void mergesort(T theArray[], int first, int last) {  
    if (first < last) {  
        int mid = (first + last)/2;           // index of midpoint  
        mergesort(theArray, first, mid);  
        mergesort(theArray, mid+1, last);  
  
        // merge the two halves  
        merge(theArray, first, mid, last);  
    }  
} // end mergesort
```

Merge

```
void merge(T theArray[], int first, int mid, int last){
    T tempArray[MAX_SIZE]; // temporary array
    int first1 = first;    // beginning of first subarray
    int last1 = mid;       // end of first subarray
    int first2 = mid + 1;  // beginning of second subarray
    int last2 = last;      // end of second subarray
    int index = first1;    // next available location in tempArray
    while (first1 <= last1) && (first2 <= last2) {
        if (theArray[first1] < theArray[first2]) {
            tempArray[index++] = theArray[first1++];
        }
        else
            tempArray[index++] = theArray[first2++];
    }
    while (first1 <= last1) tempArray[index++] = theArray[first1++];
    while (first2 <= last2) tempArray[index++] = theArray[first2++];
    // copy the result back into the original array
    for (index = first; index <= last; ++index)
        theArray[index] = tempArray[index];
} // end merge
```

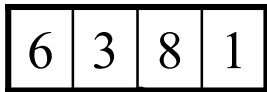
Mergesort - Example

mergesort(a,0, 7):

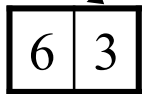


```
void mergesort(T theArray[],int first,int last){
    if (first < last) {
        int mid = (first + last)/2;
        mergesort(theArray, first, mid);
        mergesort(theArray, mid+1, last);
        merge(theArray, first, mid, last);
    }
}
```

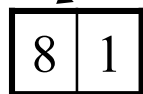
mergesort(a,0, 3):



mergesort(a,0, 1):



mergesort(a,2, 3):



mergesort(a,0, 0):



mergesort(a,1, 1):



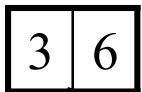
mergesort(a,2, 2):



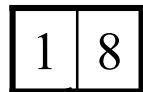
mergesort(a,3, 3):



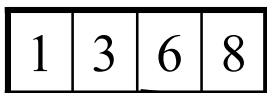
merge(a,0, 0,1):



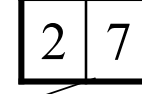
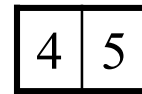
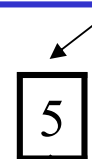
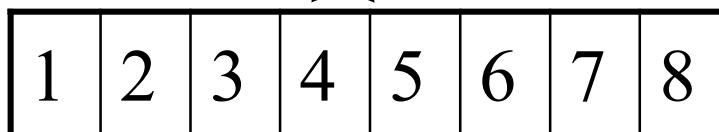
merge(a,2, 2,3):



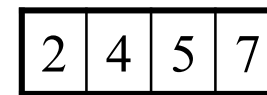
merge(a,0, 1,3):



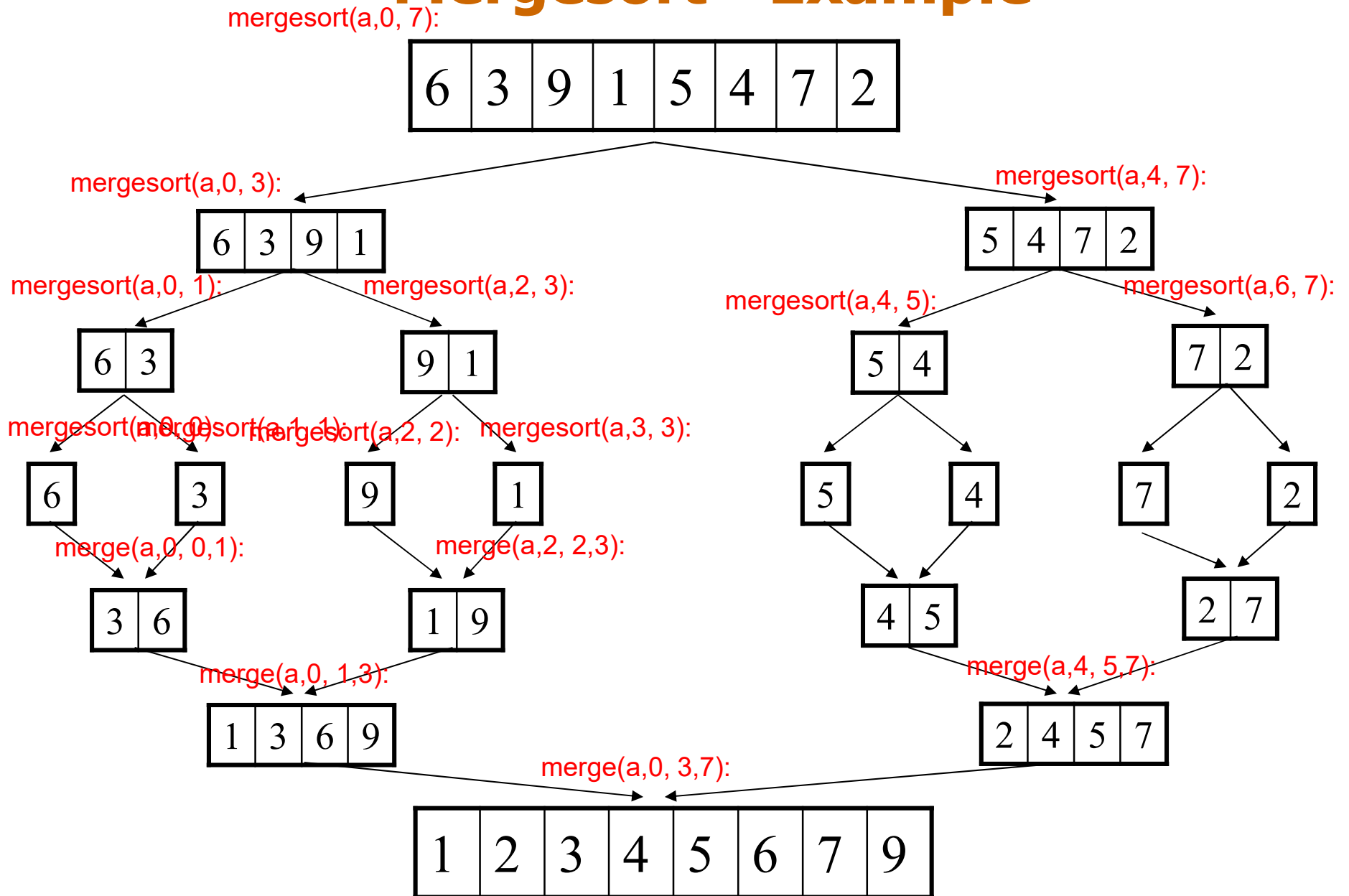
merge(a,0, 3,7):



merge(a,4, 5,7):

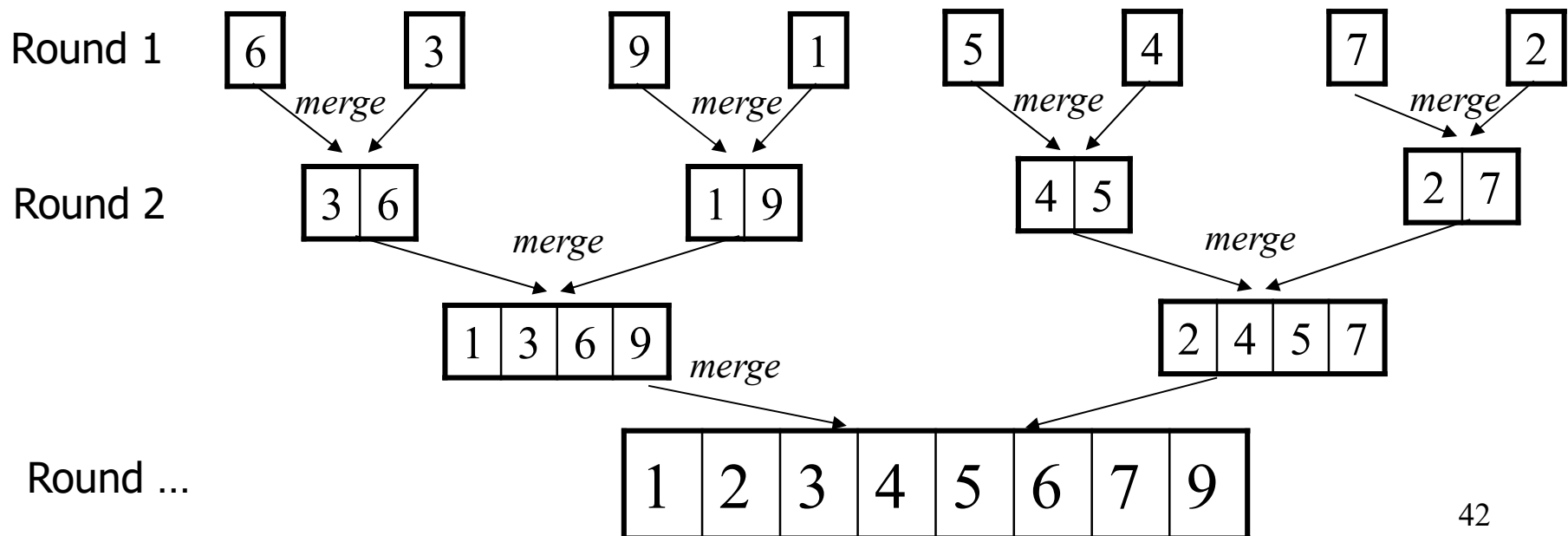


Mergesort - Example



Merge Sort Cost Analysis

- The input array has N elements (just assume N is a power of 2). What is the cost of Merge Sort?
- How much work (i.e. comparisons + copies) is involved in each round, in terms of N ?
- How many rounds are there?



Merge Sort Cost Analysis

- The input array has N elements (just assume N is a power of 2). What is the cost of Merge Sort?
 - How much work (i.e. comparisons + copies) is involved in each round, in terms of N ?
 - **$O(N)$**
 - How many rounds are there?
 - **$\log_2 N$**
- Therefore the total cost of Merge Sort is **$O(N \log N)$**

Mergesort - Analysis

The complexity of mergesort can be defined using the following recurrence equation:

$$T(n) = 2T(n/2) + n$$

Solving this equation:

$$T(1) = 1$$

$$T(n) = 2 T(n/2) + n$$

$$\begin{aligned} 1^{\text{st}} \text{ subst: } &= 2 (2 T(n/4) + n/2) + n \\ &= 4 T(n/4) + 2n \end{aligned}$$

$$\begin{aligned} 2^{\text{nd}} \text{ subst: } &= 4 (2 T(n/8) + n/4) + 2n \\ &= 8 T(n/8) + 3n \end{aligned}$$

$$\text{In general: } 2^k T(n/2^k) + kn$$

When $k = \log_2 n$:

$$= 2^{\log n} T(n/2^{\log n}) + n \log n$$

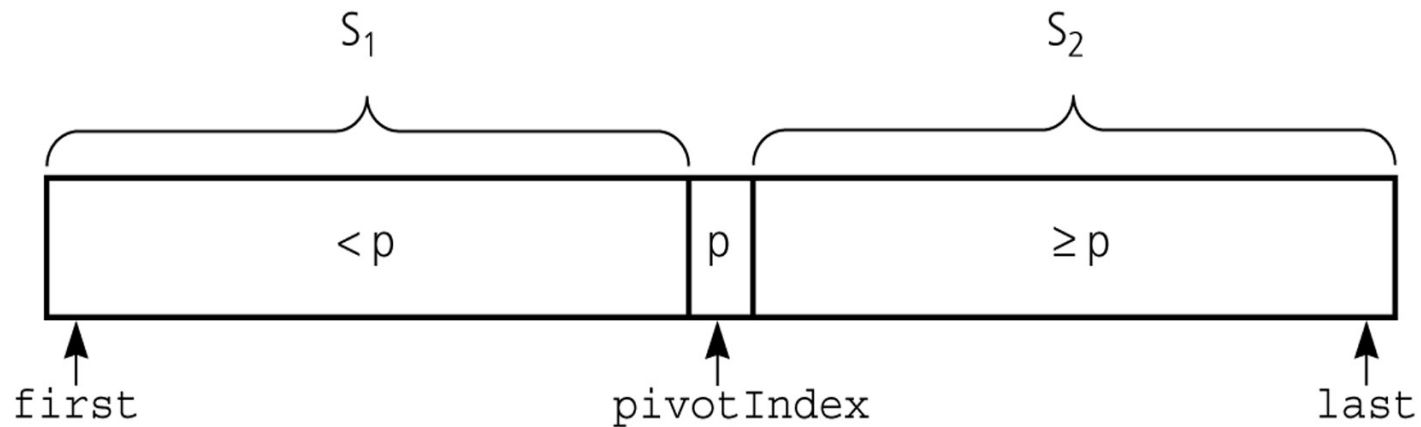
$$\Rightarrow \mathbf{O(n \log n)}$$

Quicksort

- Like mergesort, quicksort is also based on the *divide-and-conquer* paradigm.
- But it uses this technique in a somewhat opposite manner, as all the hard work is done *before* the recursive calls.
- It works as follows:
 1. First, it partitions an array into two parts using a selected *pivot* element,
 2. Then, it sorts the parts independently,
 3. Finally, it combines the sorted subsequences by a simple concatenation.

Partition

- Partitioning places the pivot in its correct position within the array.



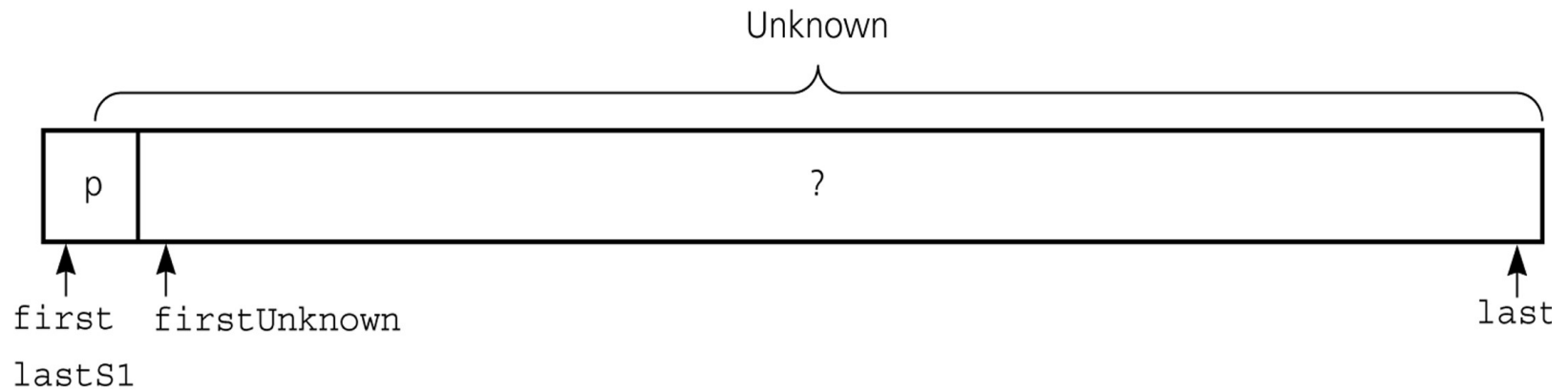
- Arranging the array elements around the pivot p generates two smaller sorting problems.
 - sort the left section of the array, and sort the right section of the array.
 - when these two smaller sorting problems are solved recursively, our bigger sorting problem is solved.

Partition – Choosing the pivot

- First, we have to select a pivot element among the elements of the given array before partitioning.
- Which array item should be selected as pivot?
- Somehow we have to select a pivot, and we hope that we will get a good partitioning.
- Possible options:
 - choose a pivot randomly.
 - choose the first or last element as a pivot (it may not give a good partitioning).
 - use different techniques to select the pivot.

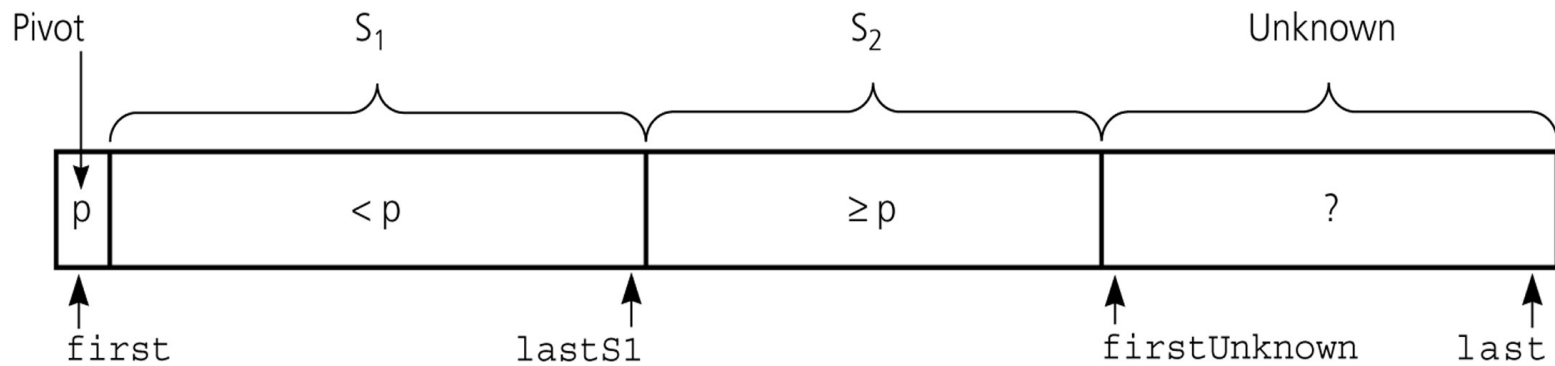
Partitioning

Initial state of the array



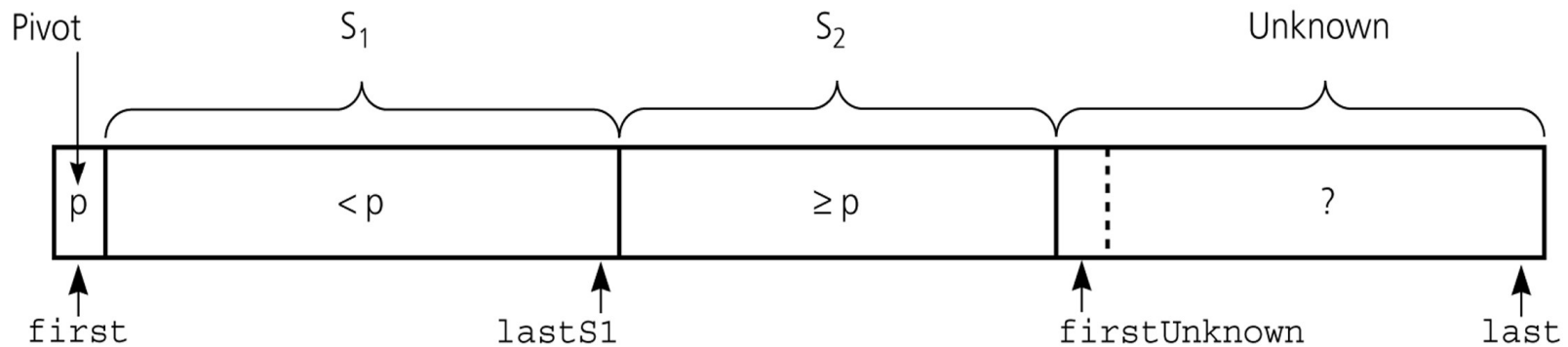
Partitioning (cont.)

Invariant for the partition algorithm



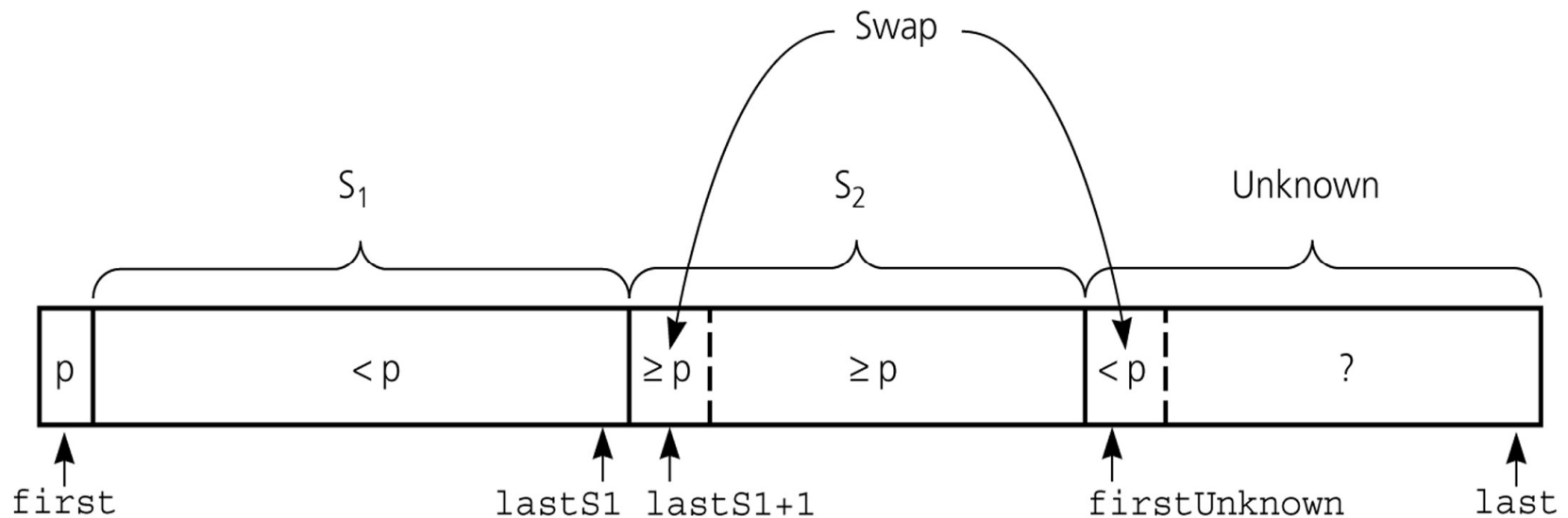
Partitioning (cont.)

Moving `theArray[firstUnknown]` **into** S_2 **by incrementing** `firstUnknown`.



Partitioning (cont.)

Moving `theArray[firstUnknown]` **into** S_1 **by swapping it with** `theArray[lastS1+1]` **and by incrementing both** `lastS1` **and** `firstUnknown`.



Partition

```
int partition(int[] array, int first, int last) {  
    int pivot = array[first]; // chooses the first element  
    int lastS1 = first;  
    int firstUnknown = first + 1;  
    while (firstUnknown <= last) {  
        if (array[firstUnknown] < pivot) {  
            lastS1++; //the end of 1st part is incremented  
            swap(array, firstUnknown, lastS1); //elements swapped  
        }  
        firstUnknown++;  
    }  
    swap(array, first, lastS1); //places pivot in proper loc.  
    return lastS1;  
}
```



Return the position of the
pivot element after partition

Partition Example

- Input : 36, 5, 48, 51, 10, 7, 22
- Pivot : 36

Quicksort

```
void quicksort( int theArray[], int first, int last) {  
    int pivotIndex;  
    if (first < last) {  
        pivotIndex = partition(theArray, first, last);  
        quicksort(theArray, first, pivotIndex-1);  
        quicksort(theArray, pivotIndex+1, last);  
    }  
}
```

Quicksort – Analysis

Worst case: If we always select the smallest or largest element as the pivot, we'll not be able to divide the array into similar size partitions

- In that case, the complexity of the quicksort can be defined by:

$$T(n) = n + T(1) + T(n-1)$$

- This gives $O(n^2)$ complexity

Best case: Partitions are equal sized:

$$T(n) = n + 2T(n/2) \text{ (same as mergesort)}$$

- This gives $O(n \log n)$ complexity

Average case: quicksort has been proven to have $O(n \log n)$ complexity

- It also does not need an extra array like mergesort
- Therefore, it is the most popular sorting algorithm

Heapsort

- The **max heap** can be used to sort N items by
 - inserting every item into a max heap and
 - extracting every item by calling **deleteMax** N times, thus sorting the result.
- An algorithm based on this idea is *heapsort*.
- First, think about a trivial implementation of Heap Sort:

```
for(i=0; i<n; i++) maxheap.insert(array[i]);  
for(i=n-1; i>=0; i--) array[i]=maxheap.remove();
```

- It is an **$O(N \log N)$** worst-case sorting algorithm.
- Is there additional storage involved here? **Yes the heap!**

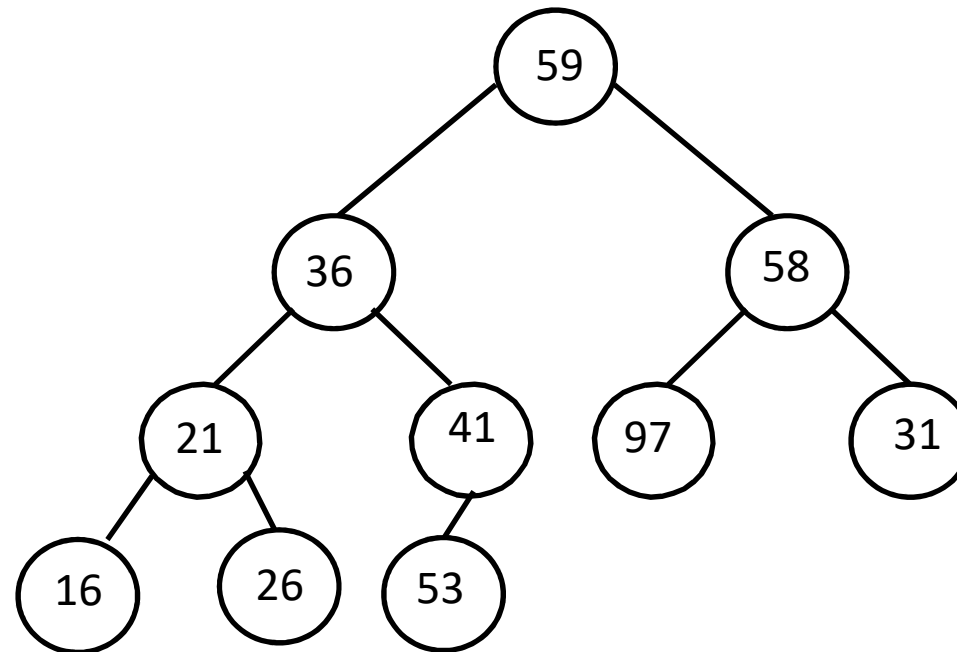
Heap Sort

- To eliminate the additional storage, recall that a heap is stored in an array. Since we already have an input array, the same array can be used as a heap. How?
 1. Convert the input array to a heap in place (heapify) (using buildheap algorithm that we discussed before)
 2. Repeatedly remove the root element from the heap and move it towards the end of the array.
 - This is conceptually similar to Selection sort, but each round costs only $O(\log N)$ due to heap operations.

Heapify Example

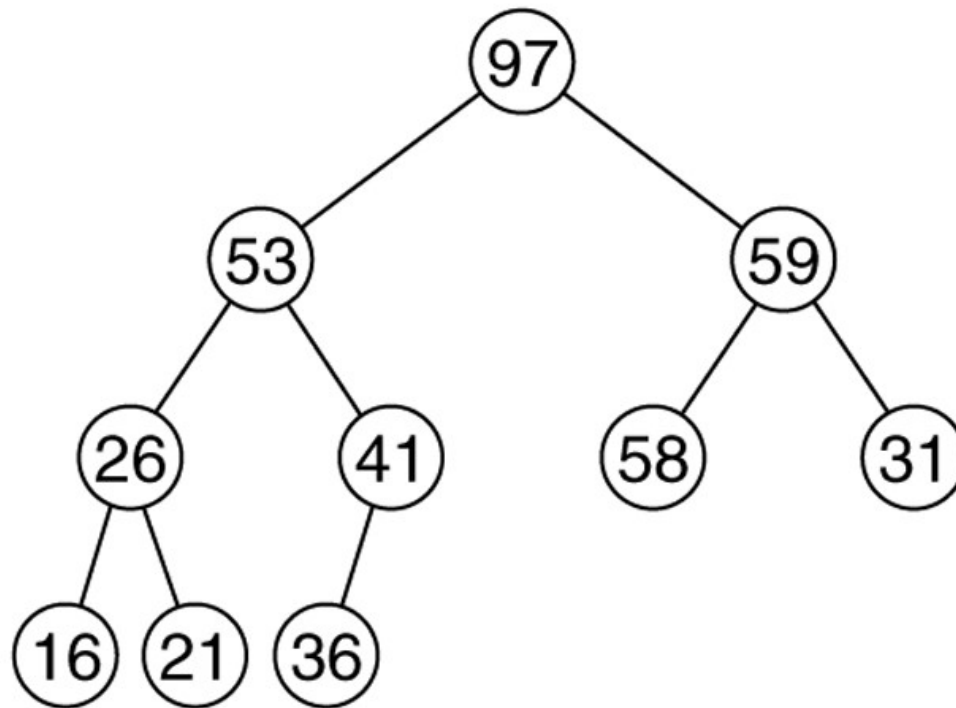
- Input array and how it looks like as a heap – this is not a valid heap yet

[0]	59
[1]	36
[2]	58
[3]	21
[4]	41
[5]	97
[6]	31
[7]	16
[8]	26
[9]	53



Heapsort

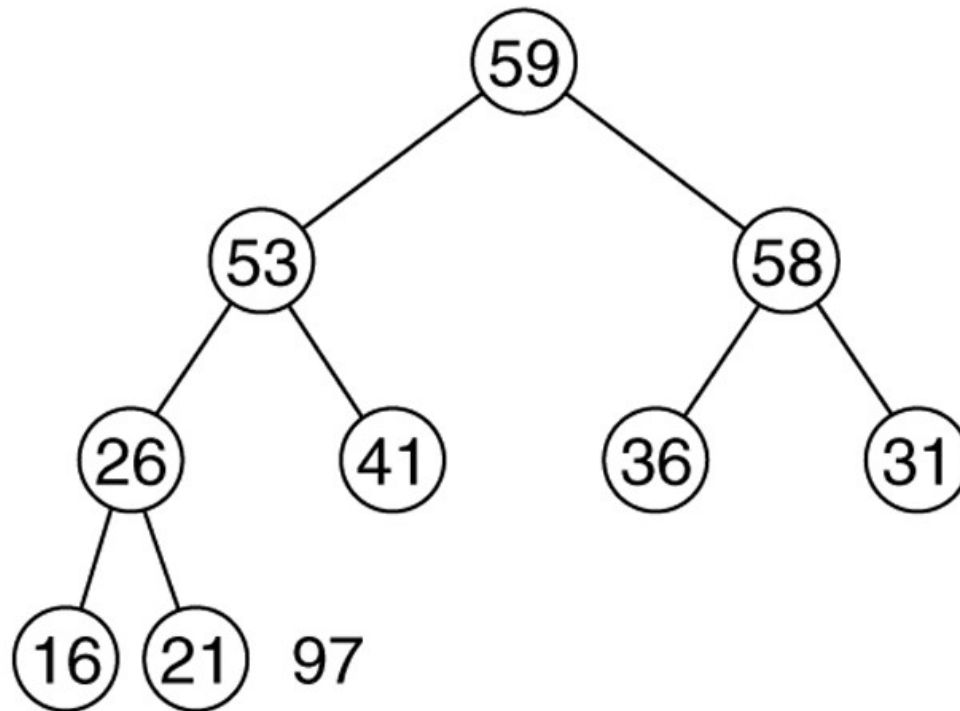
Max heap for the input sequence 59,36,58,21,41,97,31,16,26,53



97	53	59	26	41	58	31	16	21	36
0	1	2	3	4	5	6	7	8	9

Heapsort

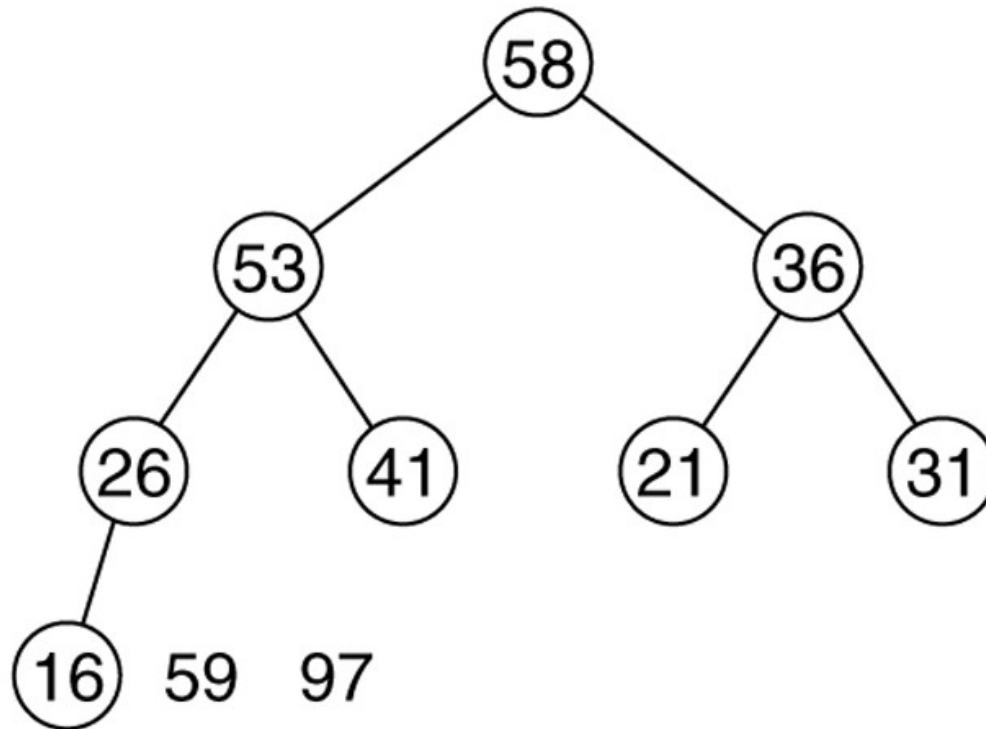
Heap after the first deleteMax operation



59	53	58	26	41	36	31	16	21	97
0	1	2	3	4	5	6	7	8	9

Heapsort

Heap after the second deleteMax operation



58	53	36	26	41	21	31	16	59	97
0	1	2	3	4	5	6	7	8	9

Implementation

- In the implementation of heapsort, the root is stored in position 0.
- Thus there are some minor changes in the code that we have seen earlier:
 - Since we use max heap, the logic of comparisons is changed from $>$ to $<$.
 - For a node in position i , the parent is in $(i-1)/2$, the left child is in $2i+1$ and right child is next to left child.
 - Bubbling down needs the current heap size which is lowered by 1 at every deletion.

The heapsort routine

```
template <class T>
void heapsort( vector<T> & a )
{
    for( int i = a.size()/2-1; i >= 0; i-- )    // heapify
        bubbleDown( a, i, a.size() );

    for( int j = a.size()-1; j > 0; j-- ) {
        swap( a[0], a[j] );                    // deleteMax
        bubbleDown( a, 0, j-1 );
    }
}
```

Modified bubbleDown

```
template <class T>
void bubbleDown(vector<T> & arr, int hole, int theSize )
{
    int child;
    T tmp = arr[ hole ];

    for( ; hole * 2 + 1 <= theSize; hole = child ) {
        child = hole * 2 + 1;
        if( child != theSize && arr[child + 1] > arr[child])
            child++;
        if( arr[ child ] > tmp )
            arr[ hole ] = arr[ child ];
        else
            break;
    }
    arr[ hole ] = tmp;
}
```


Analysis of Heapsort

- It is an $O(N \log N)$ algorithm.
 - First phase: Build heap $O(N)$
 - Second phase: N deleteMax operations: $O(N \log N)$.
- Detailed analysis shows that, the average case for heapsort is poorer than quick sort.
 - Quicksort's worst case however is far worse.
- An average case analysis of heapsort is very complicated, but empirical studies show that there is little difference between the average and worst cases.
 - Heapsort usually takes about twice as long as quicksort.
 - Heapsort therefore should be regarded as something of an insurance policy:
 - On average, it is more costly, but it avoids the possibility of $O(N^2)$.

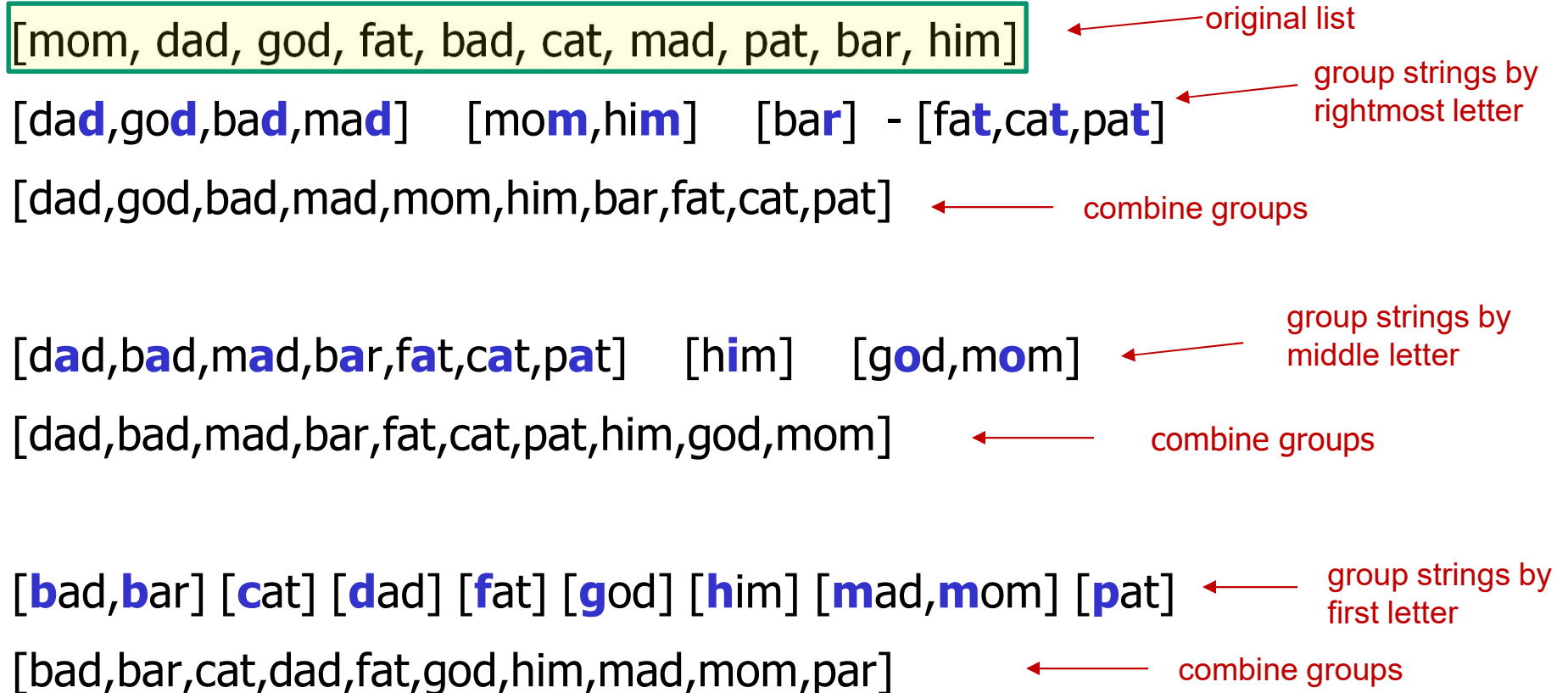
$O(N \log N)$ Sorting Summary

- **Merge Sort**
 - A key step is to merge two already sorted sub-arrays.
 - Requires an additional storage
- **Quick Sort**
 - A key step is to partition around a pivot element
 - Sorts in place
 - Does not guarantee log-linear time in worst-case scenarios.
- **Heap Sort**
 - Leverages heap's efficient insert/remove operations
 - The same input array is split into a heap region and sorted region
 - Sorts in place

Radix Sort

- Radix sort algorithm is different from other sorting algorithms that we have discussed.
 - It does not use key comparisons to sort an array.
- The radix sort treats each data item as a character string.
- Algorithm:
 - First group data items according to their rightmost character, and put these groups into order w.r.t. this rightmost character.
 - Then, combine these groups.
 - Repeat these grouping and combining operations for all other character positions in the data items from the rightmost to the leftmost character position.
 - At the end, the sort operation will be completed.

Radix Sort – Example



(SORTED)

Radix Sort – Example

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150

Original integers

(156**0**, 215**0**) (106**1**) (022**2**) (012**3**, 028**3**) (215**4**, 000**4**)

Grouped by fourth digit

1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004

Combined

(000**4**) (022**2**, 012**3**) (215**0**, 215**4**) (156**0**, 106**1**) (028**3**)

Grouped by third digit

0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283

Combined

(000**4**, 106**1**) (012**3**, 215**0**, 215**4**) (022**2**, 028**3**) (156**0**)

Grouped by second digit

0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560

Combined

(000**4**, 012**3**, 022**2**, 028**3**) (106**1**, 156**0**) (215**0**, 215**4**)

Grouped by first digit

0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154

Combined (sorted)

Radix Sort - Algorithm

// pseudocode

radixSort(theArray:ItemArray, n:integer, d:integer)

// sort n d-digit integers in the array theArray

for (j=d down to 1) {

 Initialize 10 groups to empty

 Initialize a counter for each group to 0

 for (i=0 through n-1) {

 k = jth digit of theArray[i]

 Place theArray[i] at the end of group k

 Increase kth counter by 1

 }

 Replace the items in theArray with all the items in group 0,
 followed by all the items in group 1, and so on.

}

Radix Sort -- Analysis

- The radix sort algorithm requires $2 \cdot n \cdot d$ moves to sort n strings of d characters each.
→ So, Radix Sort is **$O(n)$**
- Although the radix sort is $O(n)$, it is not appropriate as a general-purpose sorting algorithm.
 - Its memory requirement is **$d * \text{original size of data}$** (because each group should be big enough to hold the original data collection.)
 - For example, we need 26 groups to sort strings of uppercase letters in English.
 - The radix sort is more appropriate for a linked list than an array. (we will not need the huge memory in this case)

Counting sort

```
void countingSort(int arr[],int n, int RANGE){
    int count[RANGE]={0}, i, output[n];
    // store in count array the count of individual numbers
    for(i=0;i<n;i++)
        count[arr[i]]++;
    //change count[i] so that it contains the actual position
    //of numbers
    for(i=1; i < RANGE; i++)
        count[i]+=count[i-1];
    //build the output array
    for(i=0; i<n; i++){
        output[count[arr[i]]-1] = arr[i];
        count[arr[i]]--;
    }
    //copy the output array to array
    for(i=0;i<n;i++)
        arr[i]=output[i];
}
```


Example

- 1, 4, 1, 2, 7, 5, 2

Comparison of Sorting Algorithms

	<u>Worst case</u>	<u>Average case</u>
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Mergesort	$n * \log n$	$n * \log n$
Quicksort	n^2	$n * \log n$
Counting sort	n	n
Treesort	n^2	$n * \log n$
Heapsort	$n * \log n$	$n * \log n$