Fu-Yin Cherng

National Taiwan University

Sorting Algorithms

Outline

- **Introduction to Sorting**
- Basic Sorting
- Faster Sorting
- Comparison of Sorting
- Summary

Sorting

- Organize data into ascending (1,2,3) or descending (3,2,1) order
- Why we need to do sorting?
  - sort data for report
  - sorting as an initialization step for certain algo (e.g., binary search)
- Internal sorting
  - data fit entirely in the computer’s main memory
  - we can see the entire data when sorting

SORT Function

Name	Score
Hannah	93
Edward	79
Miranda	85
William	64
Joanna	81
Collin	85
Mallory	81
Oscar	63
Arturo	79
Annie	72

Sort Key

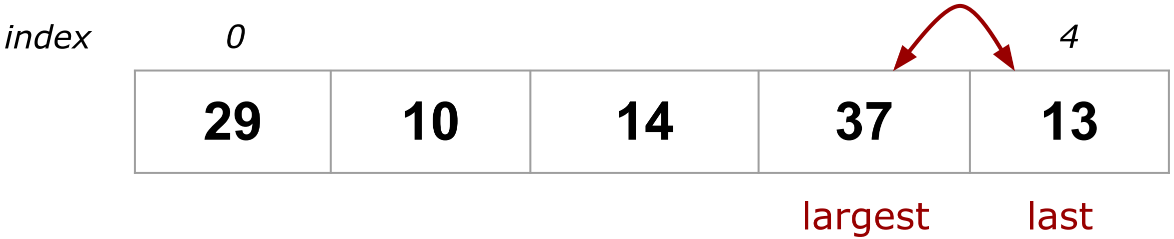
- Sort integers or character strings
- Sort object by its **sort key**
  - sort restaurants by **stars**, **distance**, or **price**
- For simplicity, all examples
  - sort quantities like numbers or strings
  - sort the data into ascending order (1,2,3; A,B,C)
  - assumes that the data resides in an array

Outline

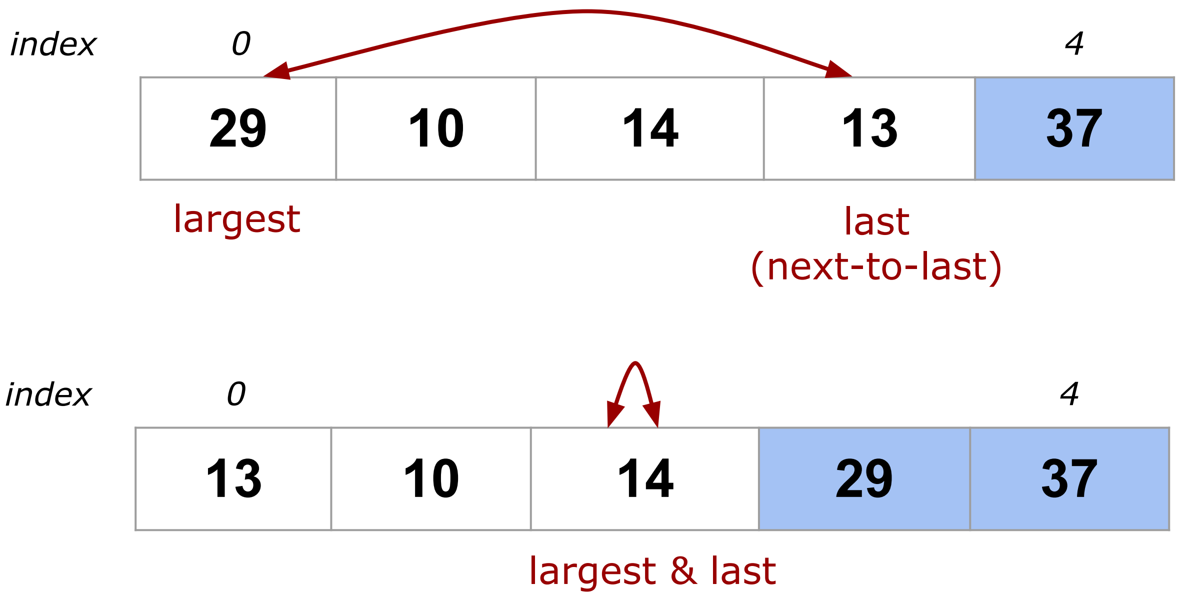
- Introduction to Sorting
- **Basic Sorting**
  - The selection sort
  - The bubble sort
  - The insertion sort
- Faster Sorting
- Comparison of Sorting
- Summary

The selection sort

- repeatedly select the **largest** item and put it at the **last** (swap it with the last item)



The selection sort



The selection sort

index	*0*				*4*
	13	**10**	**14**	**29**	**37**
	largest	last			

index	*0*				*4*
	10	**13**	**14**	**29**	**37**

Sorted into **ascending** order by the **selection sort**
(how about descending order?)

The selection sort in C++

```
int findIndexofLargest(const ItemType theArray[], int size) {
    int indexSoFar = 0;
    for (int currentIndex = 1; currentIndex < size; currentIndex++) {
        if (theArray[currentIndex] > theArray[indexSoFar])
            indexSoFar = currentIndex;
    }
    return indexSoFar; // Index of largest entry
}

void selectionSort(ItemType theArray[], int n){
    for (int last = n - 1; last >= 1; last--) {
        int largest = findIndexofLargest(theArray, last+1);
        std::swap(theArray[largest], theArray[last]);
} // end selectionSort
```

The selection sort - Analysis

```
int findIndexofLargest(const ItemType theArray[], int size) {
    int indexSoFar = 0;
    for (int currentIndex = 1; currentIndex < size; currentIndex++) {
        if (theArray[currentIndex] > theArray[indexSoFar])
            indexSoFar = currentIndex;
    }
    return indexSoFar; // Index of largest entry
}

void selectionSort(ItemType theArray[], int n){
    for (int last = n - 1; last >= 1; last--) {
        int largest = findIndexofLargest(theArray, last+1);
        std::swap(theArray[largest], theArray[last]);
} // end selectionSort
```

last (size-1) times

n-1 times

n-1 times

The selection sort - Analysis

- `findIndexofLargest: if (theArray[currentIndex] > theArray[indexSoFar])`
  - (n-1) + (n-2) + … + 1 = n*(n-1)/2 times comparisons
- `selectionSort: swap(theArray[largest], theArray[last])`
  - 3*(n - 1) data move

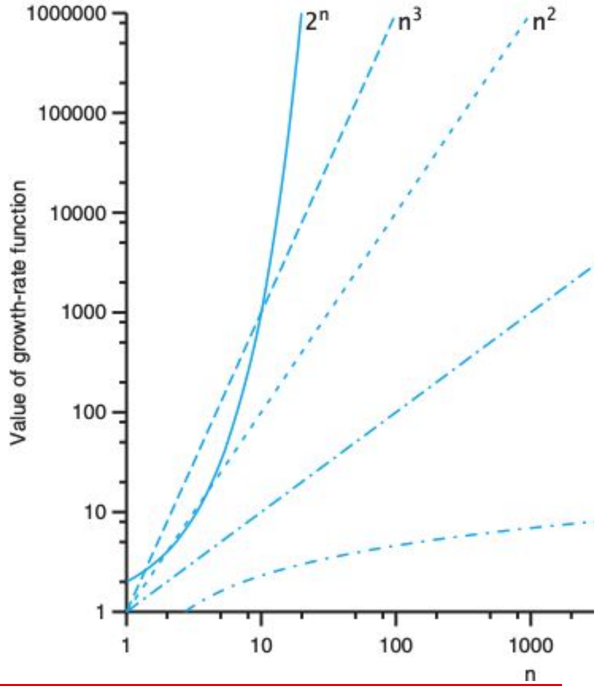
A selection sort of n items requires

$$n \times (n - 1)/2 + 3 \times (n - 1) = n^2/2 + 5 \times n/2 - 3 = O(n^2)$$

major operation *growth-rate functions*

The selection sort - summary

- not depend on initial arrangement of data
- only for small n since $O(n^2)$ grows rapidly
- What kind of data is suitable for using the selection sort? *(check CH p.309 for answer)*
  - *Hint: data move*



The bubble sort

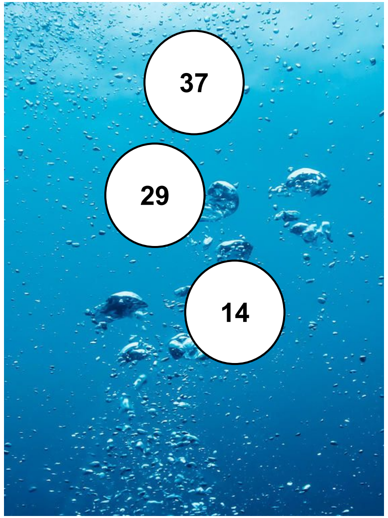
- Compare adjacent items and exchange them if they are out of order
- need several passes over the data

(a) Pass 1

Initial array:	29	10	14	37	13
	10	29	14	37	13
	10	14	29	37	13
	10	14	29	37	13
	10	14	29	13	**37**

(b) Pass 2

10	14	29	13	**37**
10	14	29	13	**37**
10	14	29	13	**37**
10	14	13	**29**	**37**



larget item bubbles to the top (end) of the array

The bubble sort in C++

```
void bubbleSort(ItemType theArray[], int n) {
    bool sorted = false; // False when swaps occur
    int pass = 1;
    while (!sorted && (pass < n)){
        sorted = true; // Assume sorted
        for (int index = 0; index < n - pass; index++){
            int nextIndex = index + 1;
            if (theArray[index] > theArray[nextIndex]){
                std::swap(theArray[index], theArray[nextIndex]);
                sorted = false;
            }
        }
        pass++;
    }
} // end bubbleSort
```

The bubble sort - Analysis

```
void bubbleSort(ItemType theArray[], int n) {
    bool sorted = false; // False when swaps occur
    int pass = 1;
    while (!sorted && (pass < n)){
        sorted = true; // Assume sorted
        for (int index = 0; index < n - pass; index++){
            int nextIndex = index + 1;
            if (theArray[index] > theArray[nextIndex]){
                std::swap(theArray[index], theArray[nextInd
                sorted = false;
            }
        }
        pass++;
    }
} // end bubbleSort
```

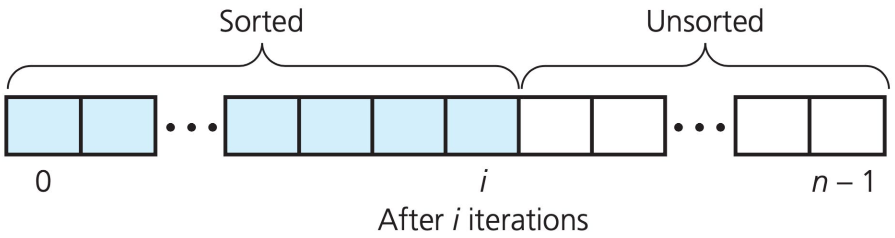
pass=1; n-1
pass=2; n-2
pass=3; n-3
…
pass=n-1; 1
comparisons/swap

The bubble sort - Analysis

- bubble sort will require a total
  - (n-1) + (n-2) + … + 1 = n*(n-1)/2 times comparisons/swaps
  - in worst case: $O(n^2)$
  - the best case (data is already sorted): only need 1 pass, so n-1 comparison: $O(n)$

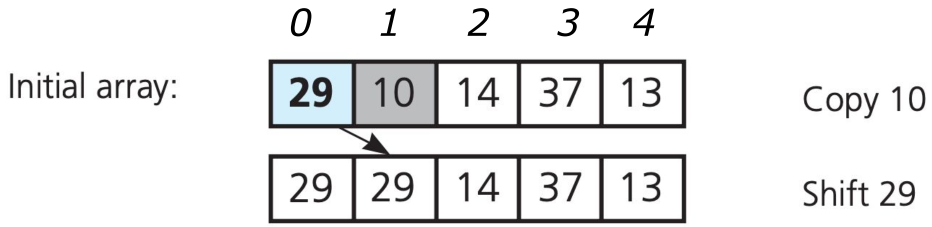
The insertion sort

- pick up an item and insert it into its proper position
- divide the array into two regions: unsorted & sorted

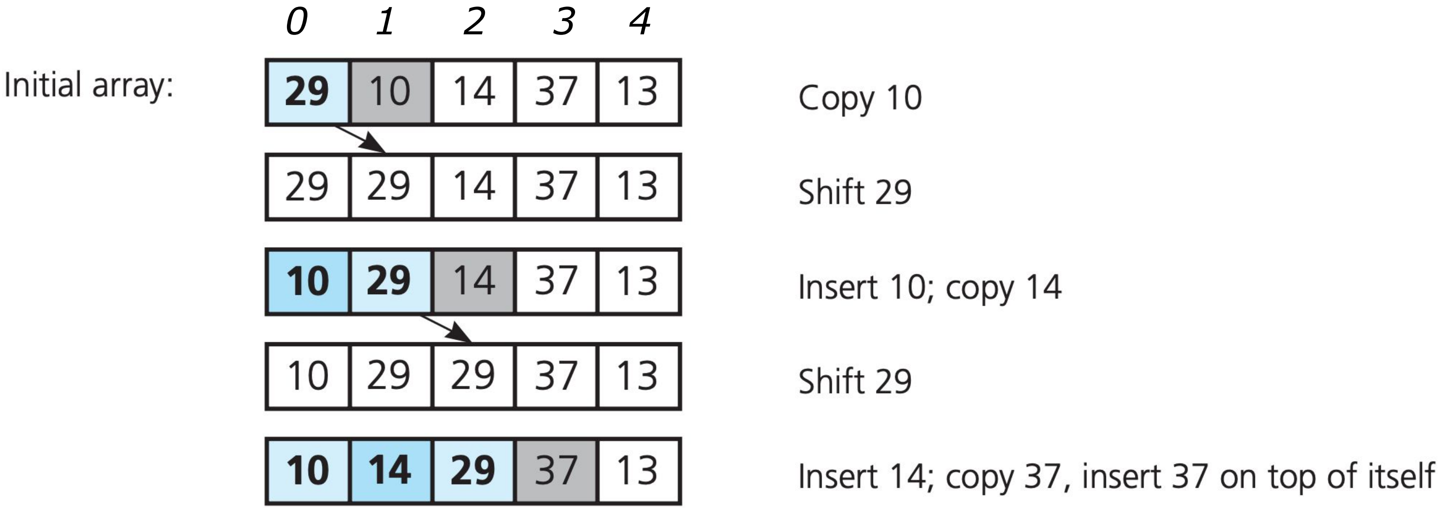


size of sorted region grows by 1 and the size of unsorted region shrinks by 1 in each step

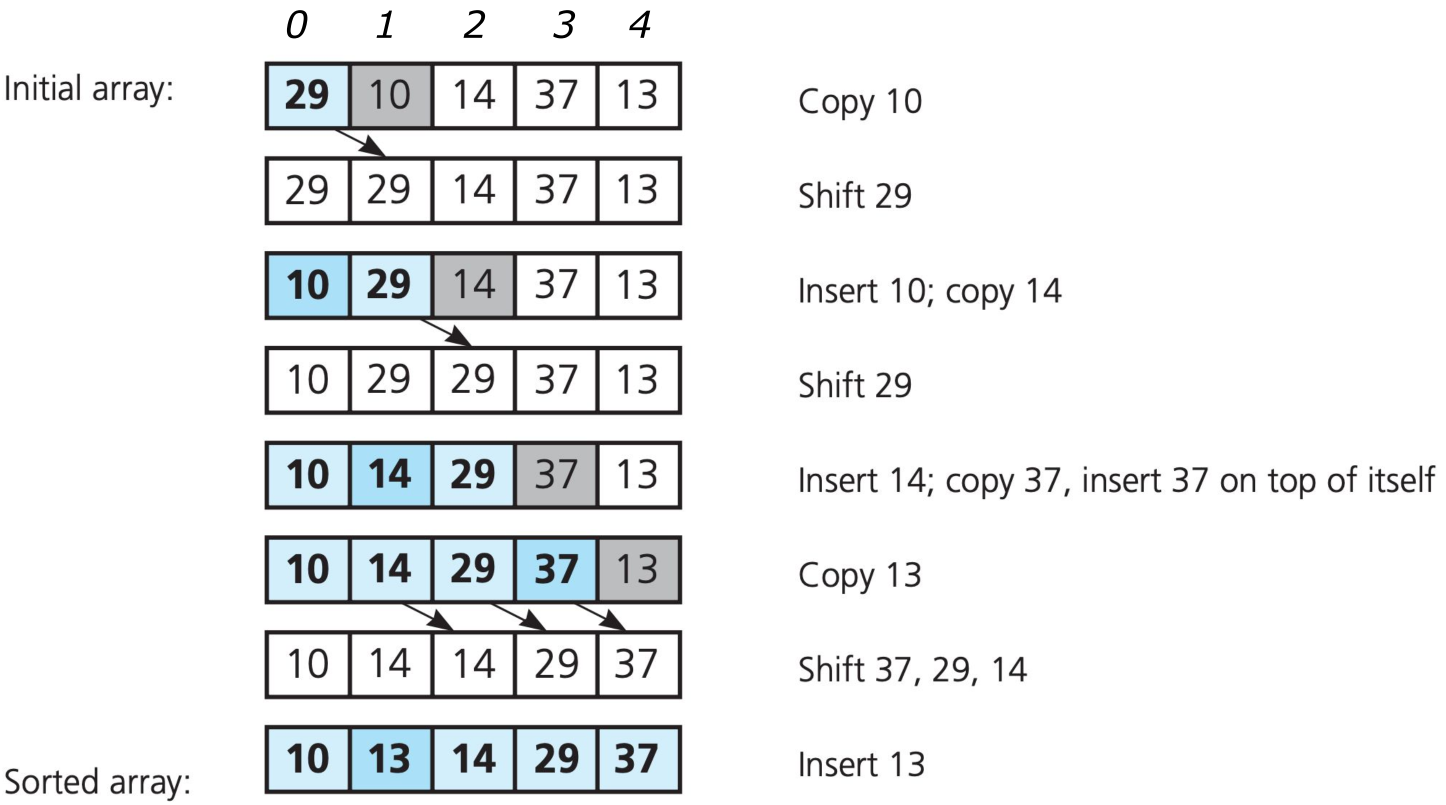
The insertion sort



The insertion sort



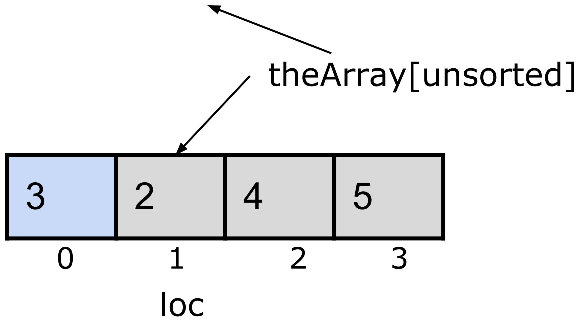
The insertion sort



The insertion sort in C++

```
void insertionSort(ItemType theArray[], int n) {
     for (int unsorted = 1; unsorted < n; unsorted++){
          //theArray[0..unsorted-1] is sorted
          //theArray[unsorted..n-1] is unsorted
          //1st item in unsorted
          ItemType nextItem = theArray[unsorted];

          //index of insertion in the sorted region
          int loc = unsorted;
          while ((loc > 0) && (theArray[loc - 1] > nextItem)){

               // Shift theArray[loc - 1] to the right
               theArray[loc] = theArray[loc - 1];
               loc--;
          }
          // Insert nextItem into sorted region
          theArray[loc] = nextItem;
     }
} // end insertionSort
```

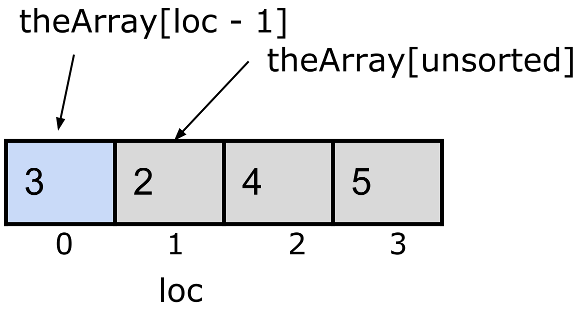
n=3
unsorted = 1
nextItem = 2



The insertion sort in C++

```
void insertionSort(ItemType theArray[], int n) {
     for (int unsorted = 1; unsorted < n; unsorted++){
          //theArray[0..unsorted-1] is sorted
          //theArray[unsorted..n-1] is unsorted
          //1st item in unsorted
          ItemType nextItem = theArray[unsorted];

          //index of insertion in the sorted region
          int loc = unsorted;
          while ((loc > 0) && (theArray[loc - 1] > nextItem)){

               // Shift theArray[loc - 1] to the right
               theArray[loc] = theArray[loc - 1];
               loc--;
          }
          // Insert nextItem into sorted region
          theArray[loc] = nextItem;
     }
} // end insertionSort
```

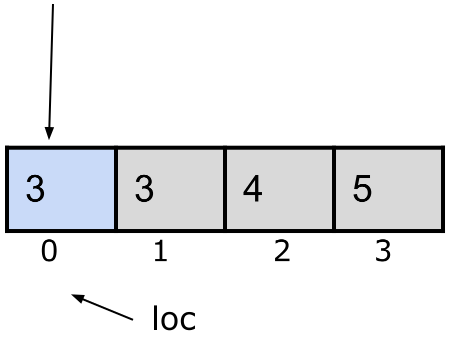
n=3
unsorted = 1
nextItem = 2



The insertion sort in C++

```
void insertionSort(ItemType theArray[], int n) {
     for (int unsorted = 1; unsorted < n; unsorted++){
          //theArray[0..unsorted-1] is sorted
          //theArray[unsorted..n-1] is unsorted
          //1st item in unsorted
          ItemType nextItem = theArray[unsorted];

          //index of insertion in the sorted region
          int loc = unsorted;
          while ((loc > 0) && (theArray[loc - 1] > nextItem)){

               // Shift theArray[loc - 1] to the right
               theArray[loc] = theArray[loc - 1];
               loc--;
          }
          // Insert nextItem into sorted region
          theArray[loc] = nextItem;
     }
} // end insertionSort
```

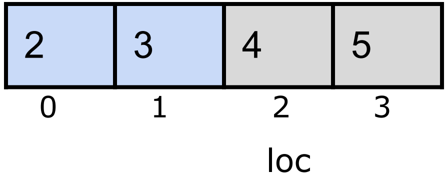
n=3
unsorted = 1
nextItem = 2



The insertion sort in C++

```
void insertionSort(ItemType theArray[], int n) {
     for (int unsorted = 1; unsorted < n; unsorted++){
          //theArray[0..unsorted-1] is sorted
          //theArray[unsorted..n-1] is unsorted
          //1st item in unsorted
          ItemType nextItem = theArray[unsorted];

          //index of insertion in the sorted region
          int loc = unsorted;
          while ((loc > 0) && (theArray[loc - 1] > nextItem)){

               // Shift theArray[loc - 1] to the right
               theArray[loc] = theArray[loc - 1];
               loc--;
          }
          // Insert nextItem into sorted region
          theArray[loc] = nextItem;
     }
} // end insertionSort
```

n=3
unsorted = 2
nextItem = 4



The insertion sort - Analysis

- in the worst case, required
  - $1 + 2 + \ldots + (n-1) = n*(n-1)/2$ times of comparisons $\longrightarrow O(n^2)$
- in the best case (data is already sorted): $O(n)$
- insertion sort only efficient for small array

Outline

- Introduction to Sorting
- Basic Sorting
- **Faster Sorting**
  - The Merge Sort
  - The Quick Sort
- Comparison of Sorting
- Summary

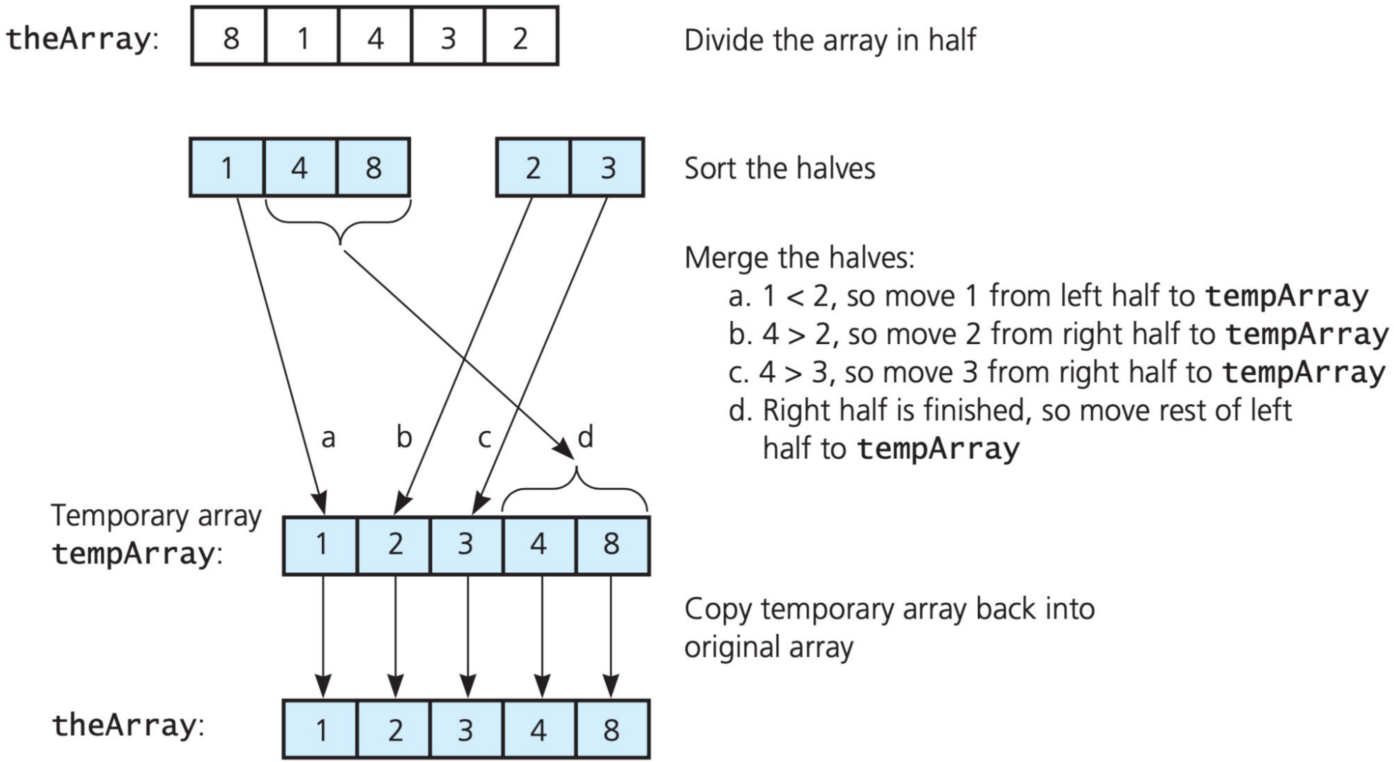
Faster Sorting

- basic sorting is sufficient for small array
- faster algorithms are needed for large arrays and data need to be updated and sorted repeatedly
- divide-and-conquer sorting algorithms
  - merge and quick sort

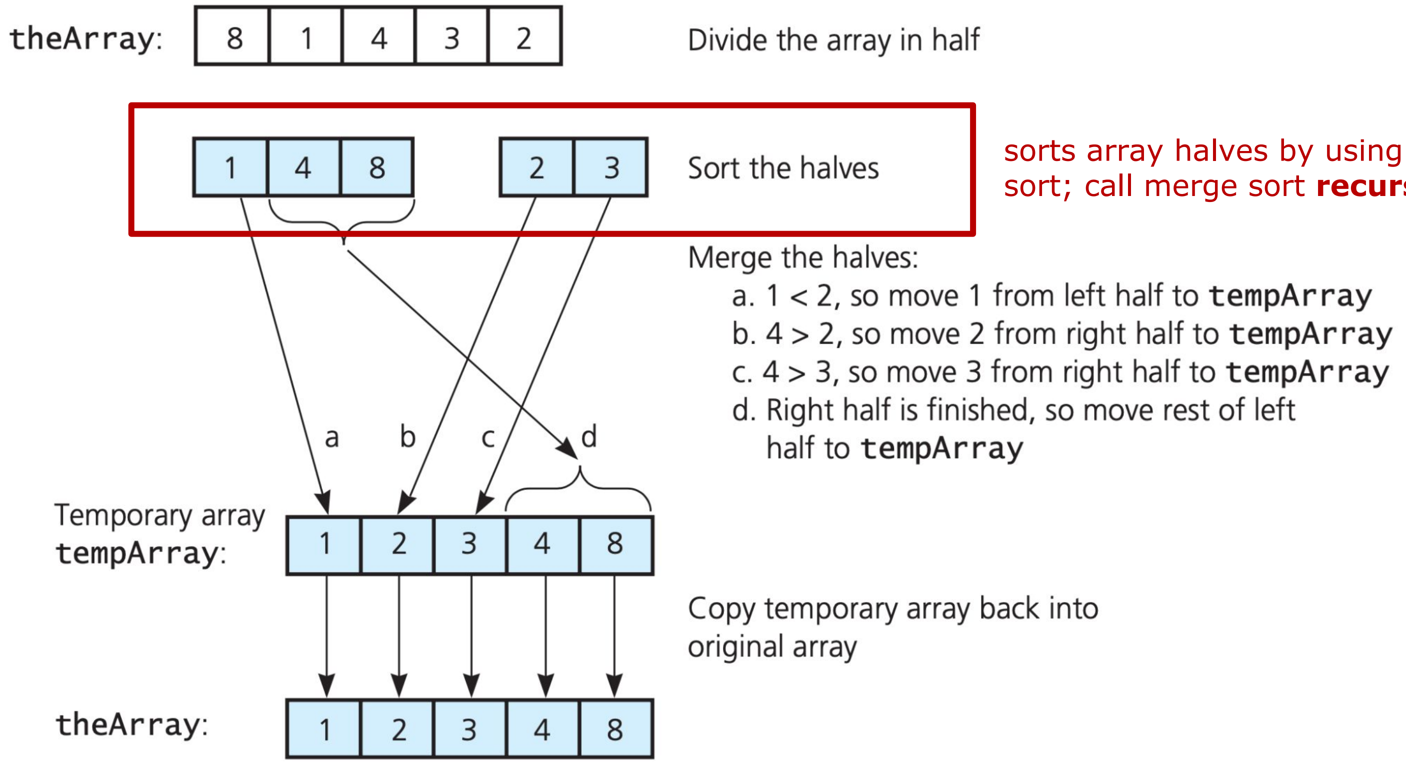
The Merge Sort

- Halve the array, recursively sort its halves, and then merge the halves

The Merge Sort



The Merge Sort

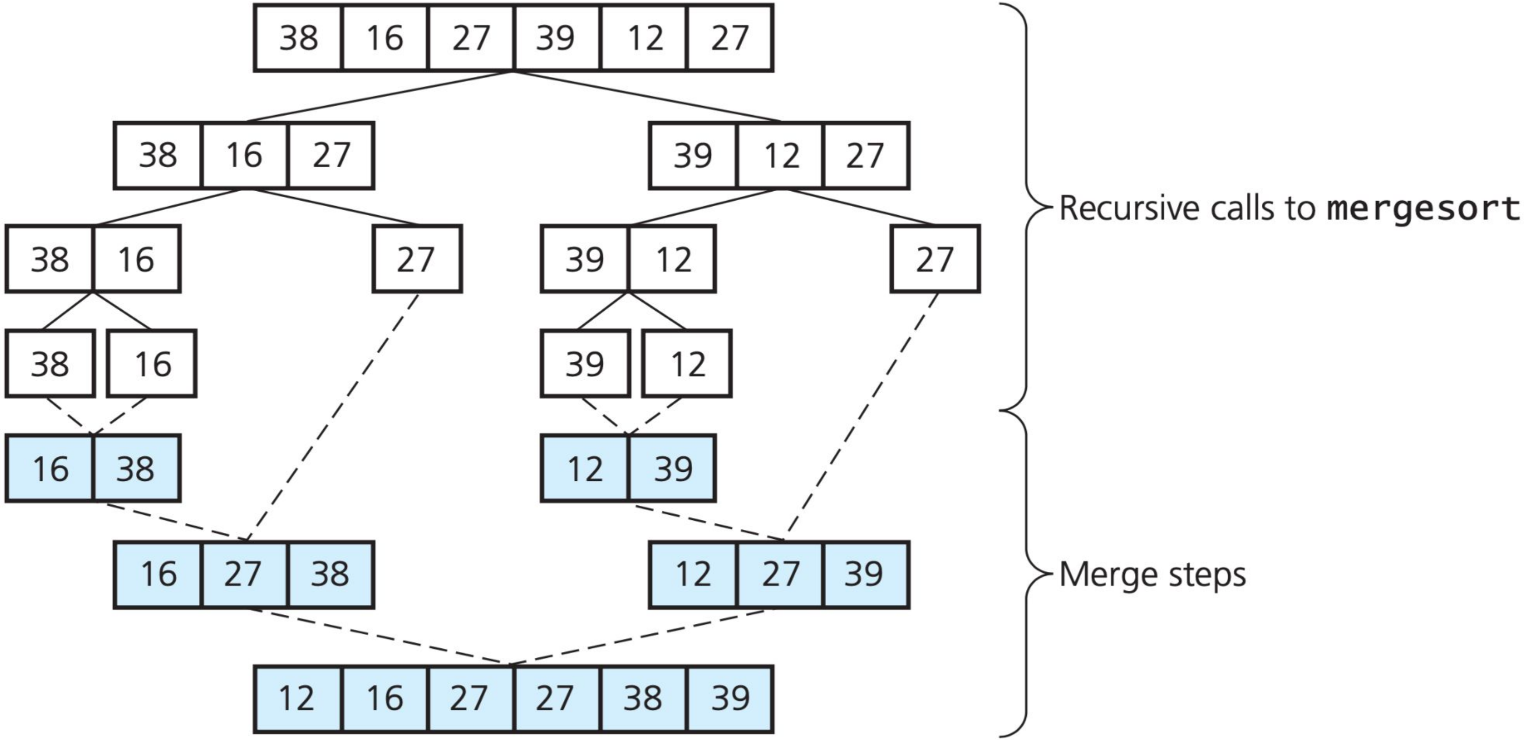


sorts array halves by using a merge sort; call merge sort **recursively**

The Merge Sort - pseudocode

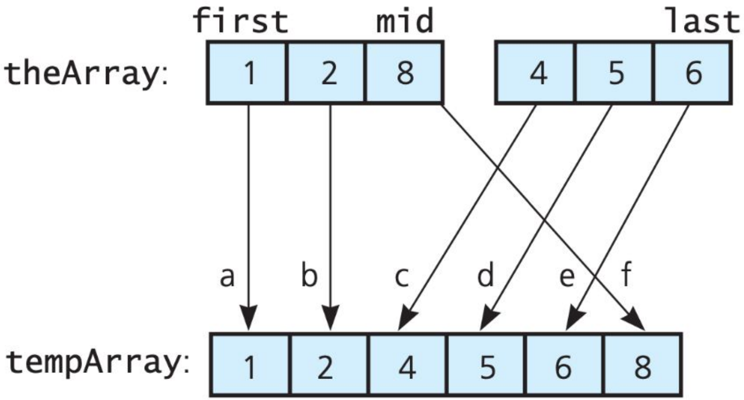
```
mergeSort(theArray: ItemArray, first: integer, last: integer)
if (first < last) {

    // Get midpoint
    mid = (first + last) / 2

    //sort 1st half
    mergeSort(theArray, first, mid)

    //sort 2nd half
    mergeSort(theArray, mid + 1, last)

    merge(theArray, first, mid, last)
}
```

The Merge Sort - pseudocode



The Merge Sort - Analysis

- n-1 times comparisons
- n *(origianl to tmp)* + n *(tmp to original)* + n-1 = 3*n-1 major operations

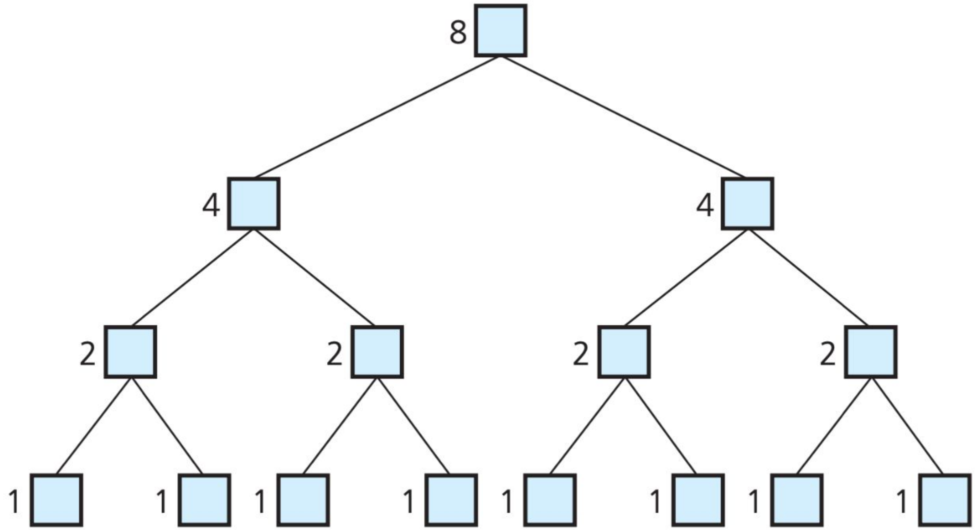


Merge the halves:

a. $1 < 4$, so move 1 from `theArray[first..mid]` to `tempArray`
b. $2 < 4$, so move 2 from `theArray[first..mid]` to `tempArray`
c. $8 > 4$, so move 4 from `theArray[mid+1..last]` to `tempArray`
d. $8 > 5$, so move 5 from `theArray[mid+1..last]` to `tempArray`
e. $8 > 6$, so move 6 from `theArray[mid+1..last]` to `tempArray`
f. `theArray[mid+1..last]` is finished, so move 8 to `tempArray`

The Merge Sort - Analysis

- recurision level: $k = \log_2 n$ or $1 + \log_2 n$



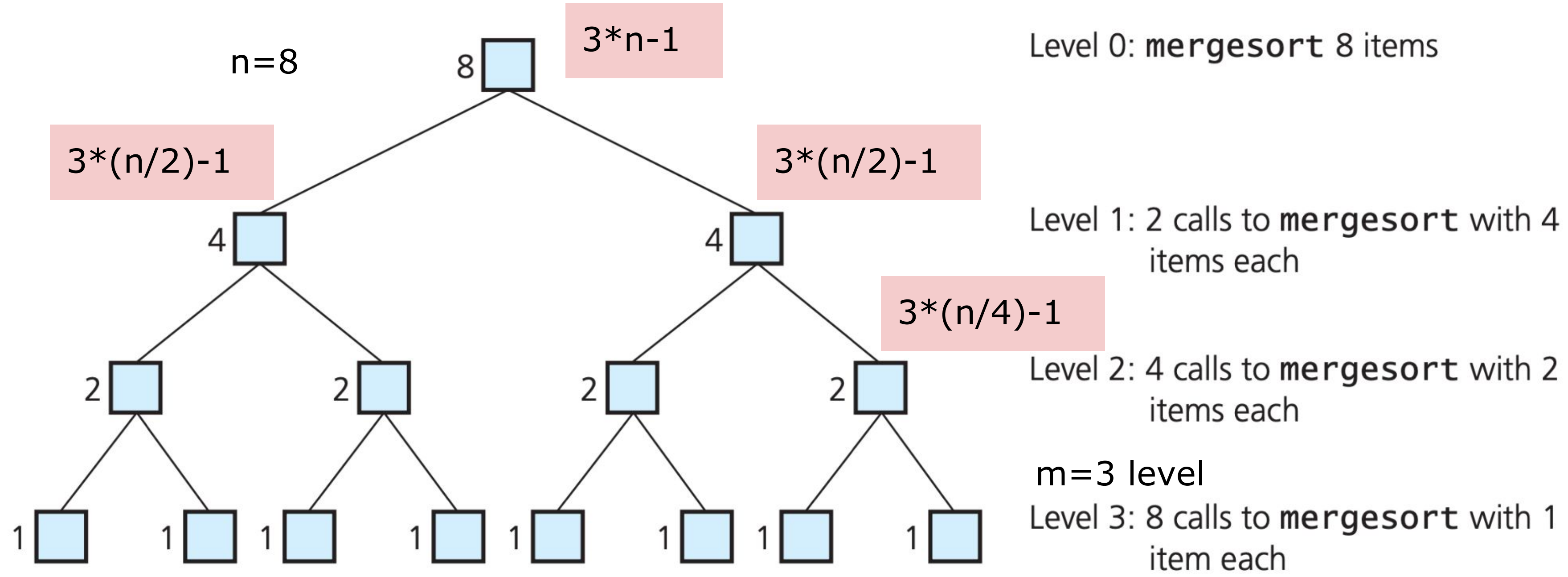
Level 0: **mergesort** 8 items

Level 1: 2 calls to **mergesort** with 4 items each

Level 2: 4 calls to **mergesort** with 2 items each

Level 3: 8 calls to **mergesort** with 1 item each

The Merge Sort - Analysis

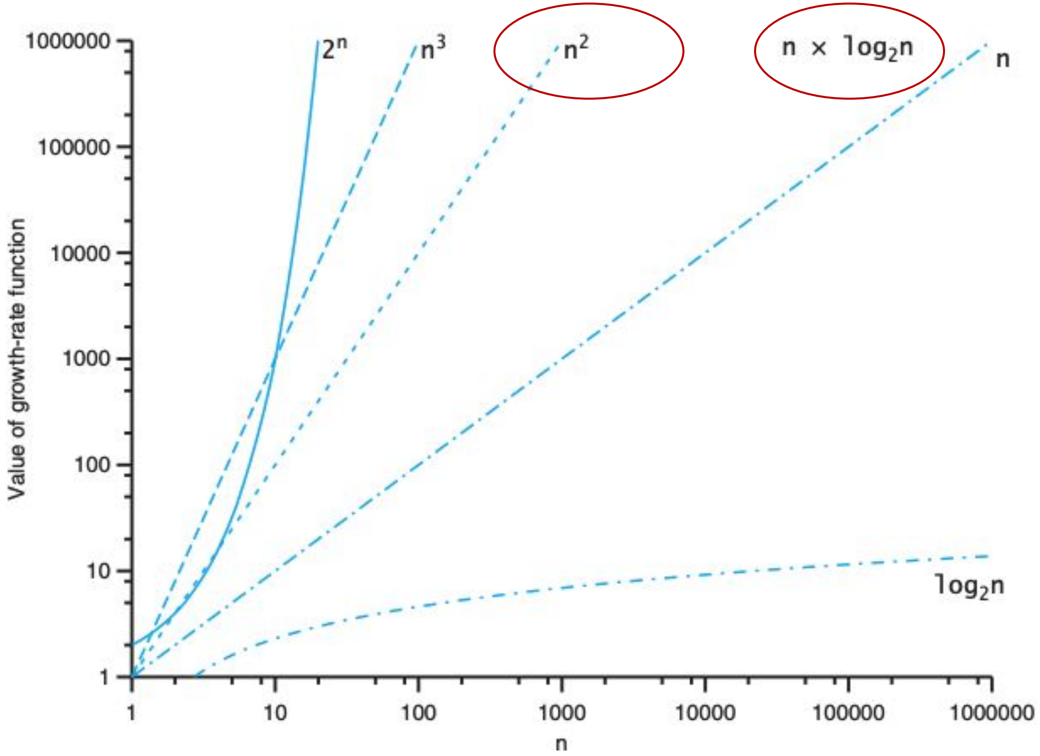


3*(n/2^m)-1 = 3*(n/2^3)-1

The Merge Sort - Analysis

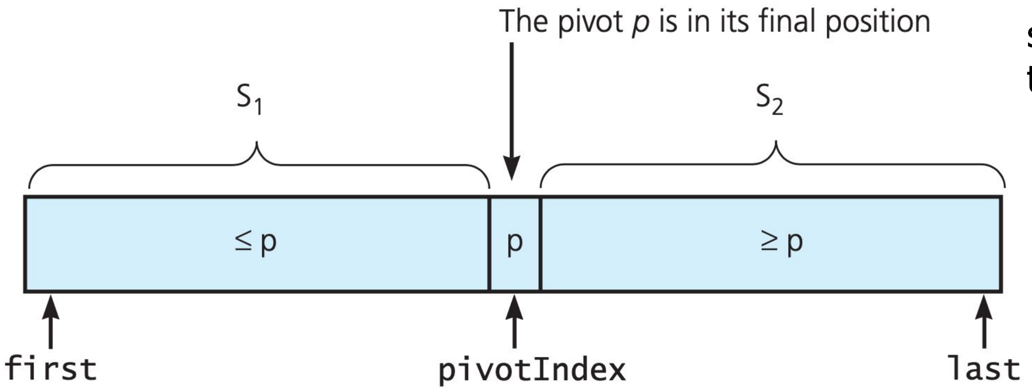
- each level recursion: $\mathrm{O}(n)$
- recurision level: $k = \log_2 n$ or $1 + \log_2 n$
- Merge sort: $\mathrm{O}(n \times \log n)$

The Merge Sort - Analysis



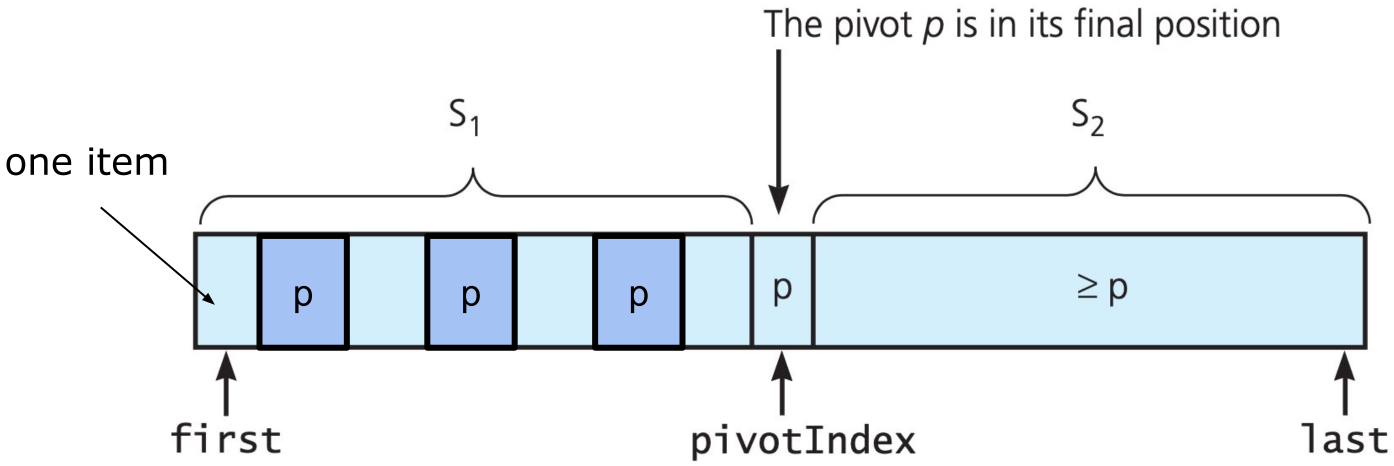
The Quick Sort

- drawback of merge sort: need temp Array cause extra storage and copy operations
- quick sort: partitions an array into items that are less than or equal to the pivot and those that are greater than or equal to the pivot



sort S1 and S2
then sort the array

The Quick Sort

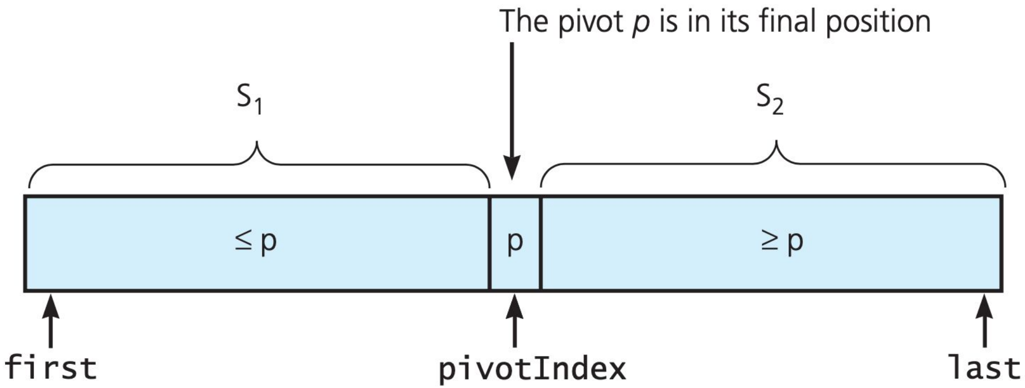


The Quick Sort - pseudocode 1

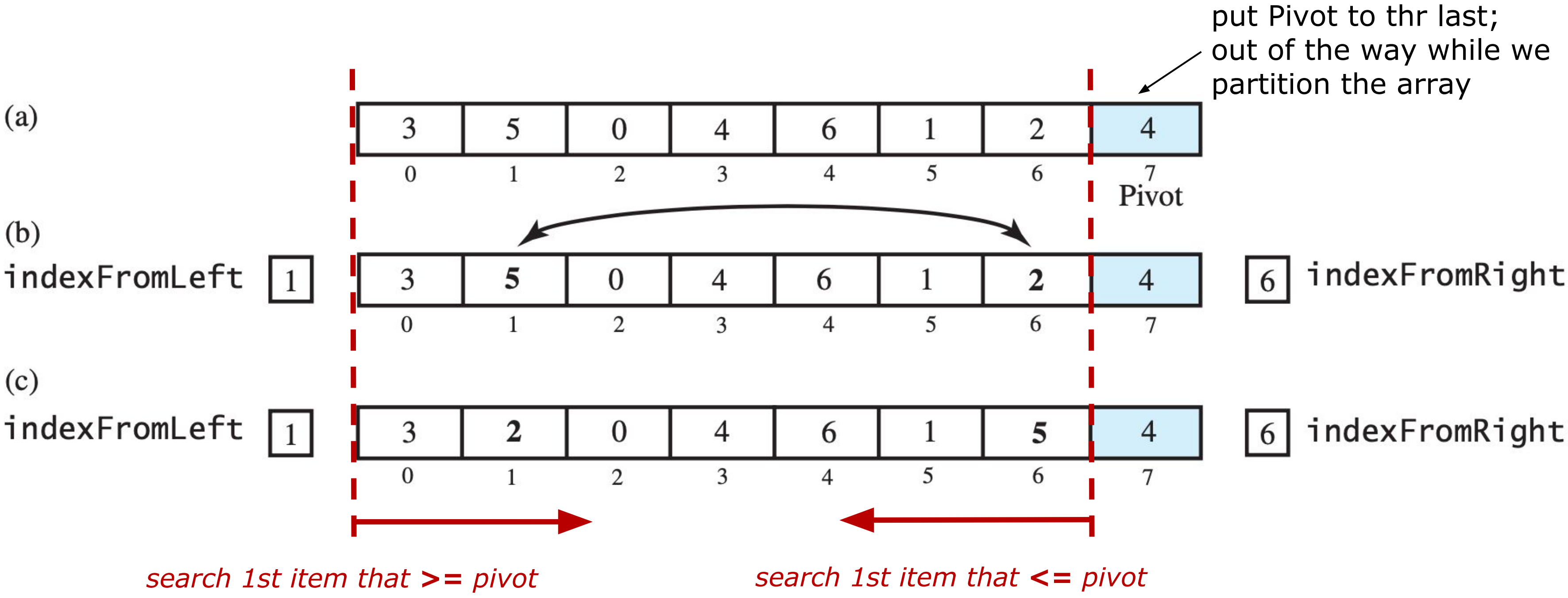
```
// Sorts theArray[first..last]
quickSort(theArray: ItemArray, first: integer, last: integer): void

if (first < last) {
    Choose a pivot item p from theArray[first..last]
    Partition the items of theArray[first..last] about p
    // The partition is theArray[first..pivotIndex..last]

    quickSort(theArray, first, pivotIndex - 1) // Sort S1
    quickSort(theArray, pivotIndex + 1, last) // Sort S2
}
// If first >= last, there is nothing to do
```

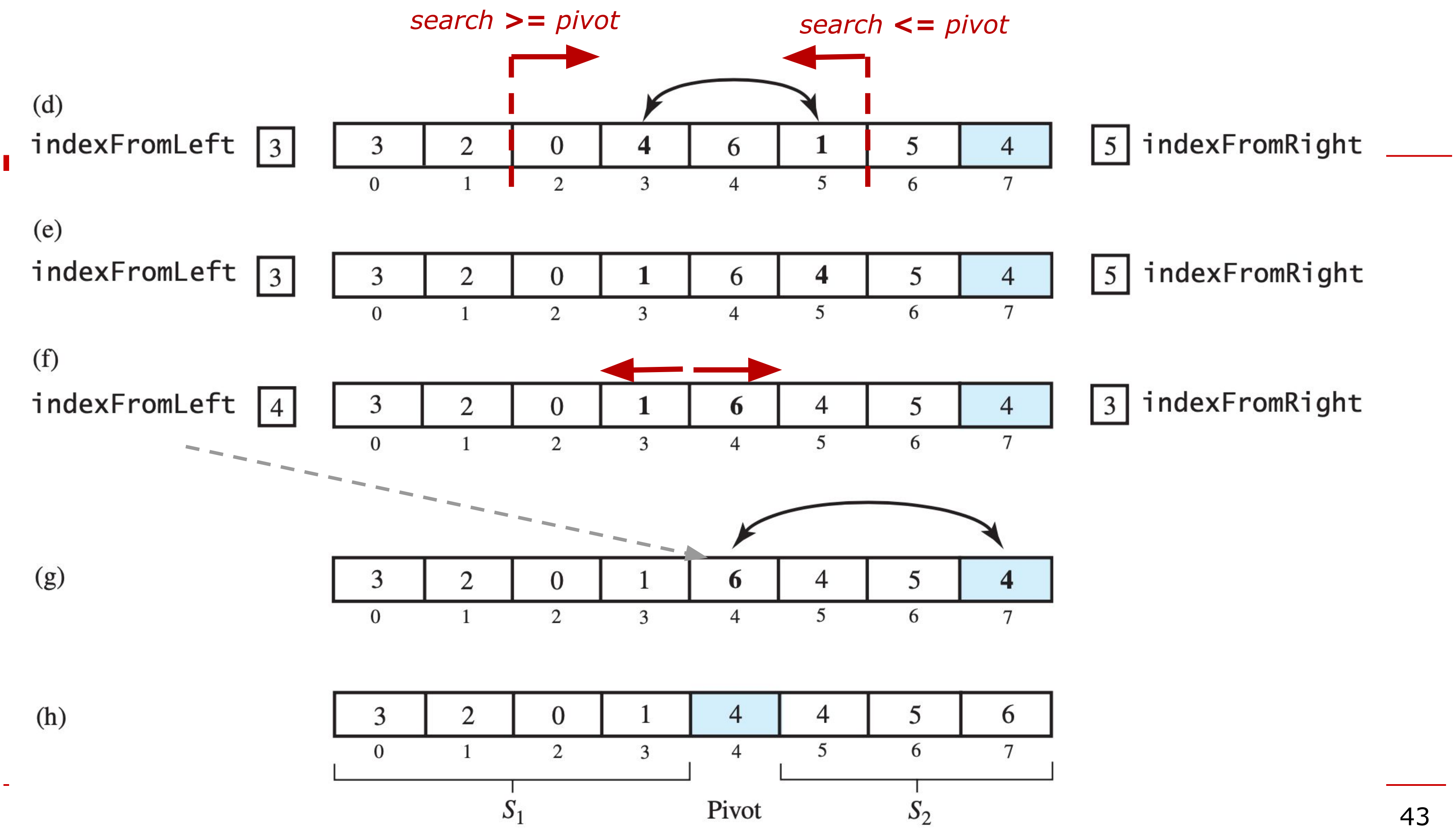
The Quick Sort - Partitioning the array

- assume we chosen a pivot, segment array into two regions
  - S1 region contains items <= pivot
  - S2 contains items >= pivot



The Quick Sort - Partitioning the array



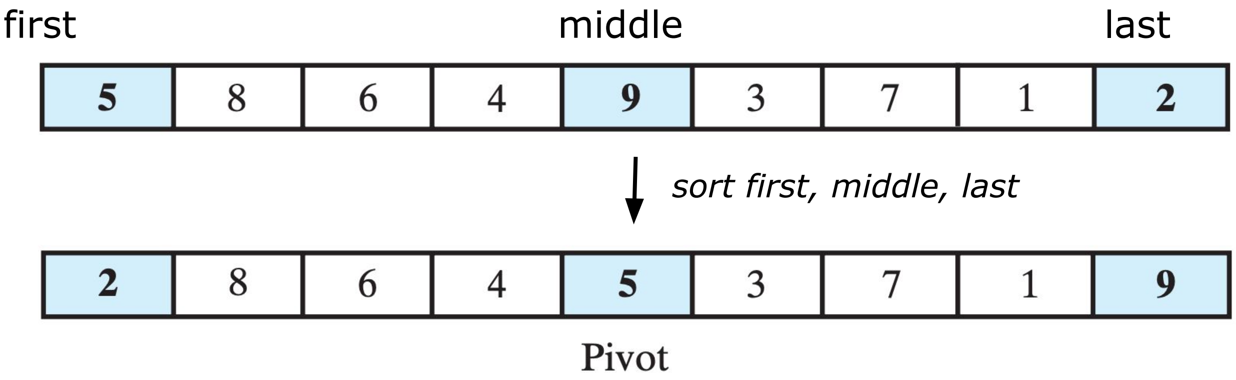

search >= pivot
search <= pivot
(d)
indexFromLeft 3
3 2 0 4 6 1 5 4
0 1 2 3 4 5 6 7
5 indexFromRight
(e)
indexFromLeft 3
3 2 0 1 6 4 5 4
0 1 2 3 4 5 6 7
5 indexFromRight
(f)
indexFromLeft 4
3 2 0 1 6 4 5 4
0 1 2 3 4 5 6 7
3 indexFromRight
(g)
3 2 0 1 6 4 5 4
0 1 2 3 4 5 6 7
(h)
3 2 0 1 4 4 5 6
0 1 2 3 4 5 6 7
S1
Pivot
S2

The Quick Sort - selecting a pivot

- the best pivot should be the median value in the array, so S1 and S2 have nearly the same number of items/entries
- but fing median value require sorting, which is the original problem
- so, we try to avoid a bad pivot

The Quick Sort - selecting a pivot

- use selection strategy: median-of-three pivot selection
  - sort the **first**, **middle**, and **last** entry (three numbers)
  - after sorted find the one in the **middle** among three



select the middel as pivot

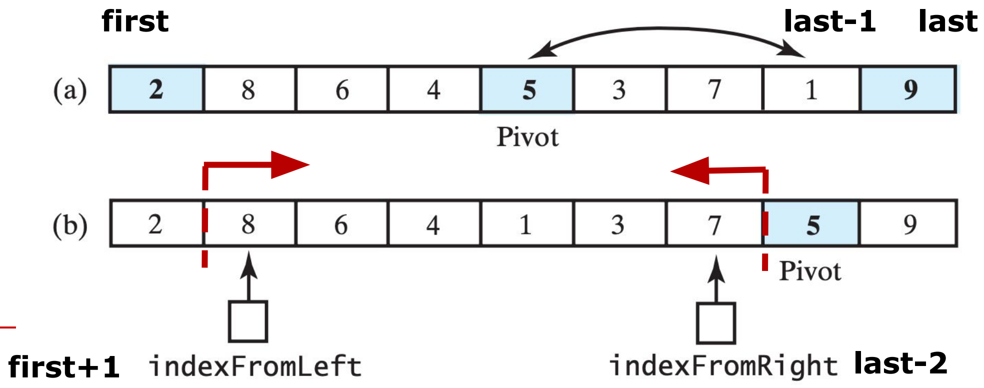
The Quick Sort - selecting a pivot

- sort the first, middle, and last entries

```
// Arranges the first, middle, and last entries in an array into ascending order.
sortFirstMiddleLast(theArray: ItemArray, first: integer, mid: integer,
last: integer): void

    if (theArray[first] > theArray[mid])
        Interchange theArray[first] and theArray[mid]

    if (theArray[mid] > theArray[last])
        Interchange theArray[mid] and theArray[last]

    if (theArray[first] > theArray[mid])
        Interchange theArray[first] and theArray[mid]
```

The Quick Sort - adjusting the partition algorithm

- Why? after median-of-three pivot, we know
  - **last** >= pivot, so belongs to S2
  - **first** <= pivot, so belogs to S1
- put pivot to [last -1]
- start searching from first+1 & last-2



The Quick Sort - partition pseudocode

```
partition(theArray: ItemArray, first: integer, last:
integer): integer
      // Choose pivot and reposition it
      mid = first + (last - first) / 2
      sortFirstMiddleLast(theArray, first, mid, last)
      Interchange theArray[mid] and theArray[last – 1]
      pivotIndex = last - 1
      pivot = theArray[pivotIndex]

      // Determine the regions S1 and S2
      indexFromLeft = first + 1
      indexFromRight = last - 2

      done = false
      while (not done){
            // Locate first entry on left that is ≥ pivot
            while (theArray[indexFromLeft]< pivot)
                  indexFromLeft = indexFromLeft + 1

            // Locate first entry on right that is ≤ pivot
            while (theArray[indexFromRight]> pivot)
                  indexFromRight = indexFromRight - 1
```

```
if (indexFromLeft < indexFromRight) {

      Interchange theArray[indexFromLeft] and
theArray[indexFromRight]

      indexFromLeft = indexFromLeft + 1
      indexFromRight = indexFromRight - 1
}

else //indexFromLeft > indexFromRight
      done = true
}//end of while(not done)


// Place pivot in proper position between S1 and S2,
and mark its new location
Interchange theArray[pivotIndex] and
theArray[indexFromLeft]
pivotIndex = indexFromLeft

return pivotIndex
```

read with figures at 42 & 43 in this slides or Figure11-10 in textbook

The Quick Sort - small array

- if the array has small number of entries (<=10), then consider use basic sorting like selection sort instead of quick sort
  - no need to do recursion, pivot selection, partition...
- check array length before use quick sort
- select the suitable sorting algorithm based on your data

The Quick Sort - Analysis

- major effort in partitioning step
  - require no more than n comparisons, so $O(n)$
- Like merge sort, there are $\log_2 n$ or $1+\log_2 n$ levels of recursive calls to quick sort
- Quick sort: $O(n \times \log n)$ in average case
  - worst case (each partition has one empty subarray)
    - $O(n^2)$

The Quick Sort - compare to merge sort

- merge sort is always $O(n \times \log n)$
- quick sort normally $O(n \times \log n)$
  - but worst case: $O(n^2)$
  - but no need extra storage
  - worst case rarely occured in practice
  - quick sort is often faster than merge sort
- chose the one that fit your need

Comparison of Sorting

- growth rates of time: big O

	Worst case	Average case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Merge sort	$n \times \log n$	$n \times \log n$
Quick sort	n^2	$n \times \log n$
Radix sort	n	n
Tree sort	n^2	$n \times \log n$
Heap sort	$n \times \log n$	$n \times \log n$

Summary

- Basic sort vs faster sort
  - best case and worst case
- Standard Template Library (STL) provides several sort functions in the library header <algorithm>
- Knowing how to choose sorting method based on the problem and data