Divide and Conquer Algorithms: Advanced Sorting

Prichard Ch. 10.2: Advanced Sorting Algorithms

Properties of orders of magnitude

1. You can ignore low-order terms in an algorithm's growth-rate function.
   - $O(n^3+4n^2+3n)$ is $O(n^3)$
2. You can ignore the multiplicative constant in the high-order term
   - $O(5n^3)$ is $O(n^3)$
3. You can combine growth-rate functions

Combining orders of magnitude

- Sequential
  - Big-O bound: Steepest growth dominates
    - copying of array, followed by binary search:

      O(n) + O(log(n)) is O(n)

- Embedded code
  - Big-O bound multiplicative
    - a for loop with n iterations and a body taking O(log n) is O(n log n)

Examples

Find the simplest and lowest $O(g(n))$ for:

$A.\ f(n) = 17n + 11$

$B.\ f(n) = n^2 + 1000$

$C.\ f(n) = nlogn + n$

$D.\ f(n) = n^4/2 + \mathrm{n}^3\mathrm{logn}$

$E.\ f(n) = 2^n$

$Is\ f(n)=log_2 n \quad O(log_3 n)$? $Is\ f(n)=log_3 n \quad O(log_2 n)$?

$Is\ f(n)=2^n \quad O(3^n)$? $\qquad Is\ f(n)=3^n \quad O(2^n)$?

Demonstrating Efficiency

- Computational complexity of the algorithm
  - Time complexity
  - Space complexity
    - Analysis of the computer memory required
    - Data structures used to implement the algorithm

Best, Average, and Worst Cases

- **Worst case**
  - Just how bad can it get:
    - The maximal number of steps
- **Average case**
  - Amount of time expected “usually”
- **Best case**
  - The smallest number of steps (not very useful, e.g. (linear or binary) search: O(1))

Sequential Search

10	9	3	8	5	6	7	4	1	45	90	22	2	0

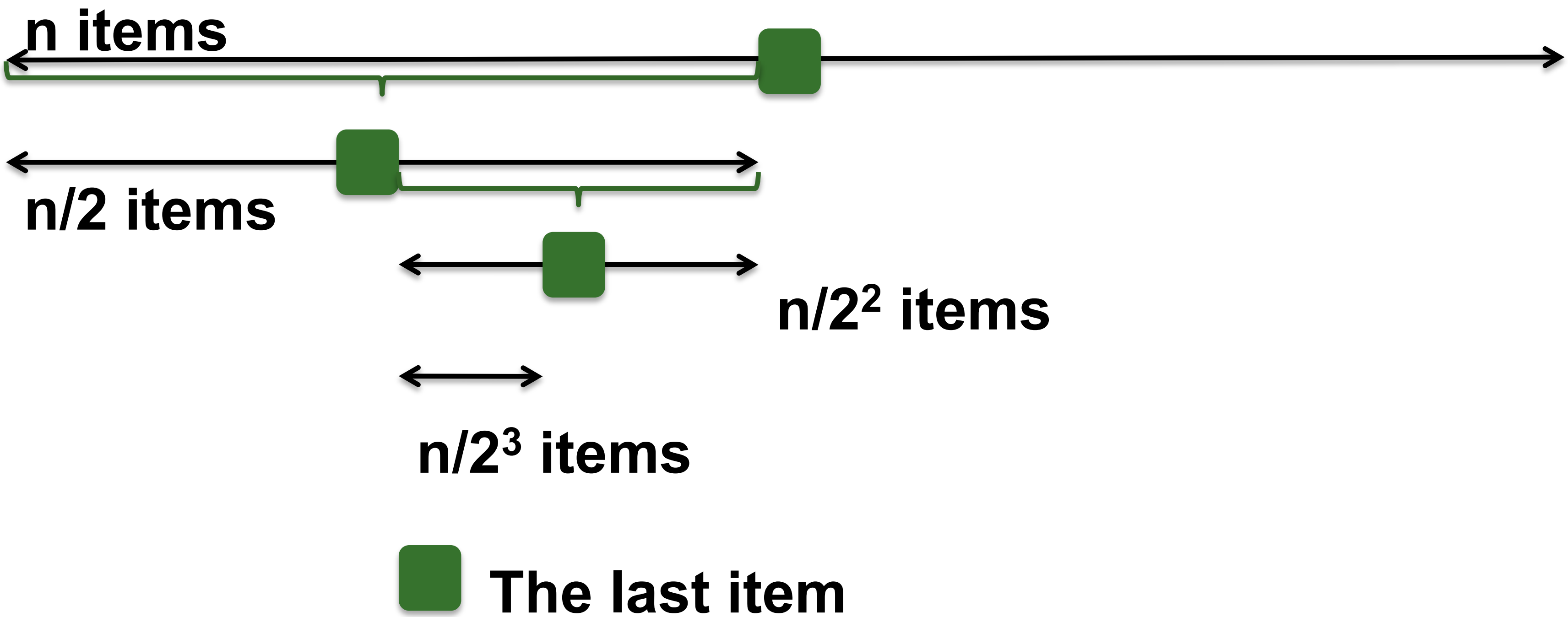
- Array of n items
  - From the first one until either you find the item or reach the end of the array.
  - Best case: $O(1)$
  - Worst case: $O(n)$ (n times of comparison)
  - Average case: $O(n)$ ($n/2$ comparison)

Binary Search (1/4)

10	11	13	15	16	20	22	39	40	45	90	92	93	94

- Searches a ***sorted array*** for a particular item by repeatedly dividing the array in half.
- Determines which half the item must be in and discards other half.
- Suppose that $n = 2^k$ for some k. (n=1,2,4,8,16,…)
  1. Inspect the middle item of size n
  2. Inspect the middle item of size $n/2$
  3. Inspect the middle item of size $n/2^2$
  4. .
  5. .
  6. .

Binary Search (2/4)



If we have $n = 2^k$, in worst case, it will repeat this k times therefore binary search size n takes O(?)

Binary Search (3/4)

- Dividing array in ***half*** k times.
- Worst case
  - Algorithm performs k divisions and k comparisons.
  - Since $n = 2^k$, $k = \log_2 n$

Repeatedly dividing a positive int n by any constant c until n=0 is O($\log_c$n) is O(log n)

Binary Search (4/4)

- What if n is not a power of 2?
- We can find the smallest k such that,

$2^{k-1} < n < 2^k$

$k - 1 < \log_2 n < k$

$k < 1 + \log_2 n < k+1$

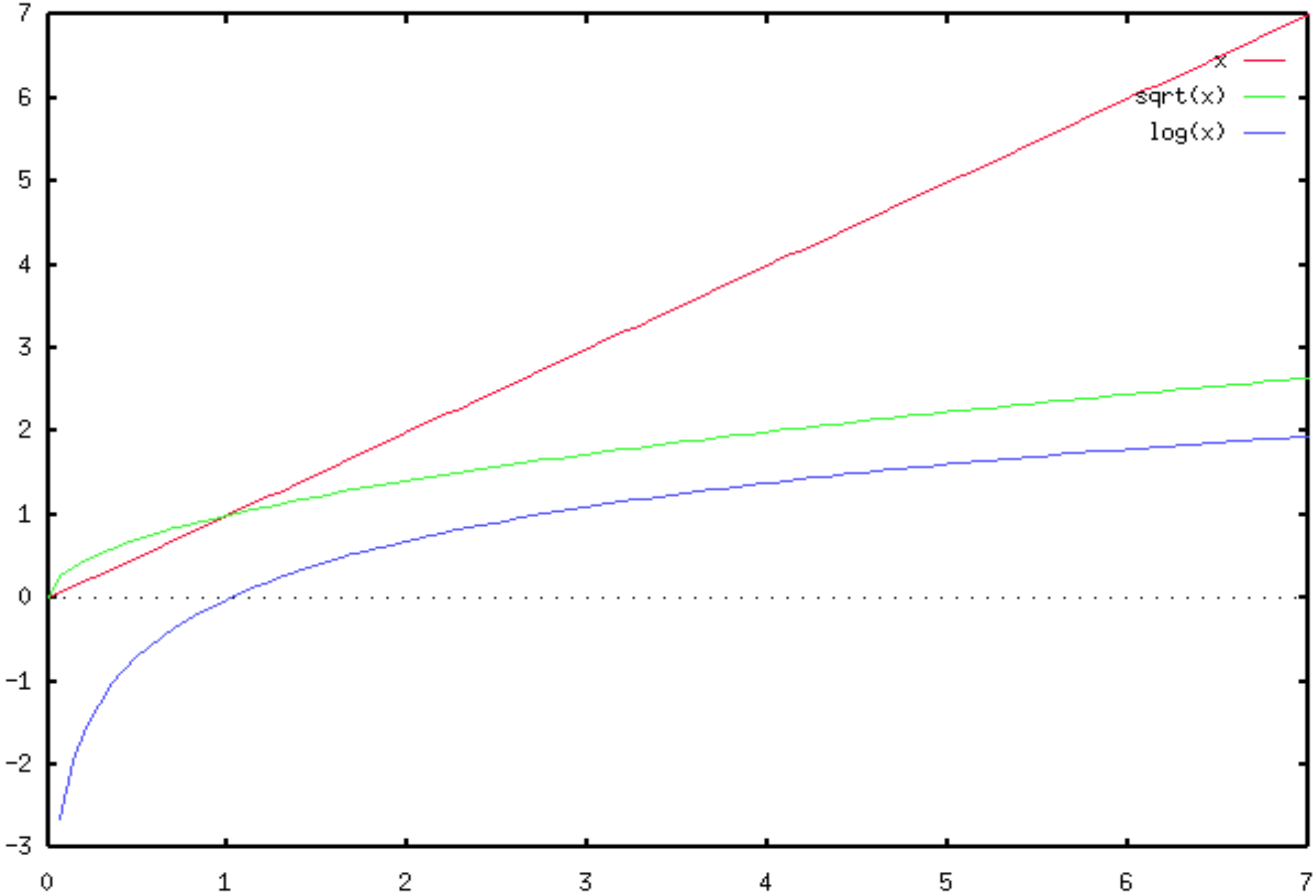
$k = 1 + \log_2 n$ rounded down

Therefore, the algorithm is still $O(\log_2 n)$. We only think in integers, because we are counting "the number of steps"

Is Binary Search is more Efficient than Linear Search? (1/2)

- For large number, $O(\log_2 n)$ requires significantly less time than $O(n)$
- For small numbers such as $n < 25$, does not show big difference.

Is Binary Search is more Efficient than Linear Search? (2/2)



Sorting Algorithm

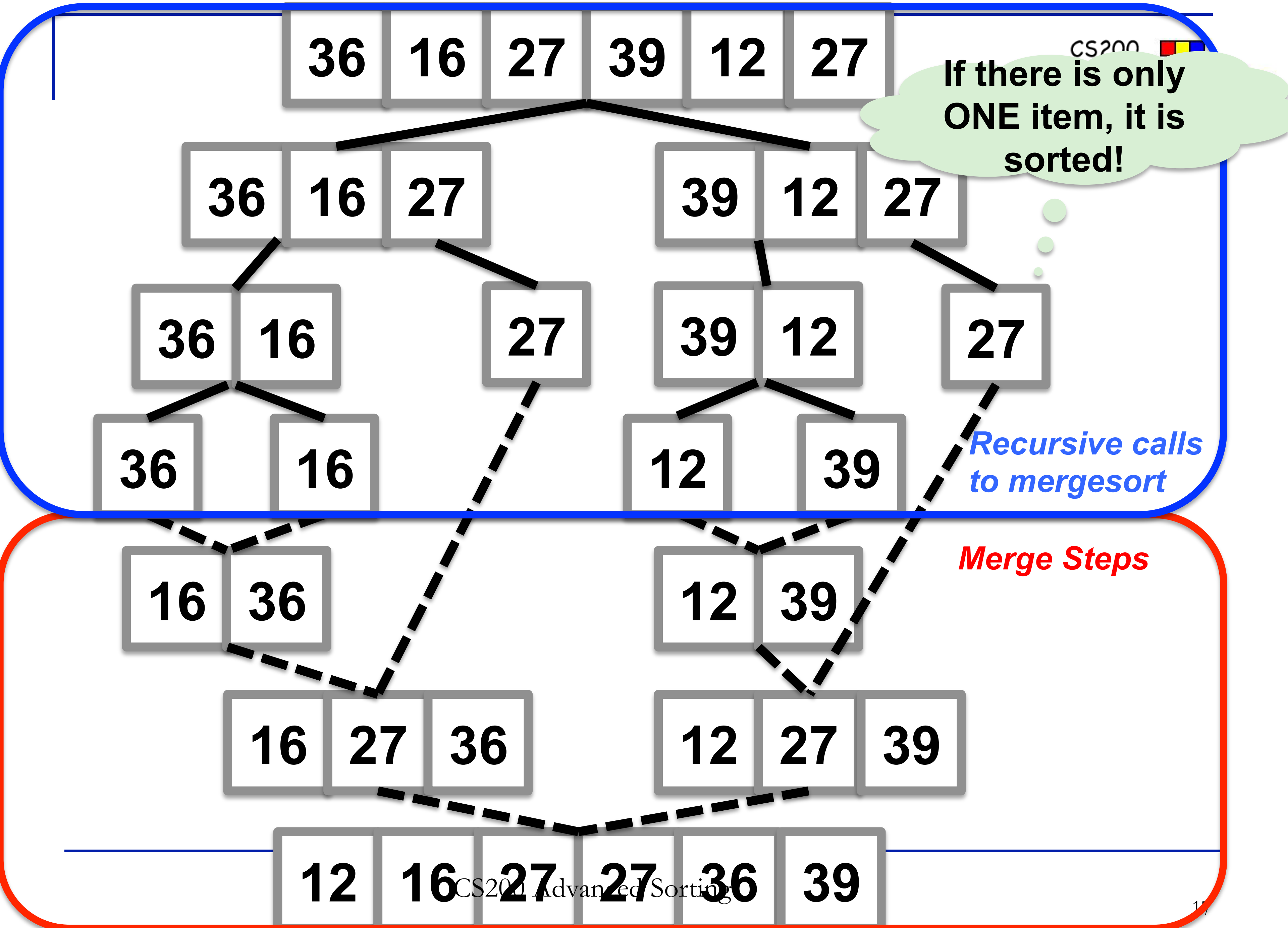
- Organize a collection of data into either ***ascending*** or ***descending*** order.
- *Internal sort*
  - Collection of data fits entirely in the computer's main memory
- *External sort*
  - Collection of data will not fit in the computer's main memory all at once.
- We will only discuss **internal sort**.

Sorting Refresher from cs161

- Simple Sorts: Bubble, Insertion, Selection
- Doubly nested loop
- Outer loop puts one element in its place
- It takes i steps to put element i in place
  - n-1 + n-2 + n-3 + … + 3 + 2 + 1
  - $O(n^2)$ complexity
  - In place: O(n) space

Mergesort

- ***Recursive sorting algorithm***
- ***Divide-and-conquer***
  - Step 1. Divide the array into halves
  - Step 2. Sort each half
  - Step 3. Merge the sorted halves into one sorted array


36 16 27 39 12 27
36 16 27
39 12 27
If there is only ONE item, it is sorted!
36 16
27
39 12
27
36
16
12
39
Recursive calls to mergesort
16 36
12 39
Merge Steps
16 27 36
12 27 39
12 16 27 27 36 39

MergeSort code

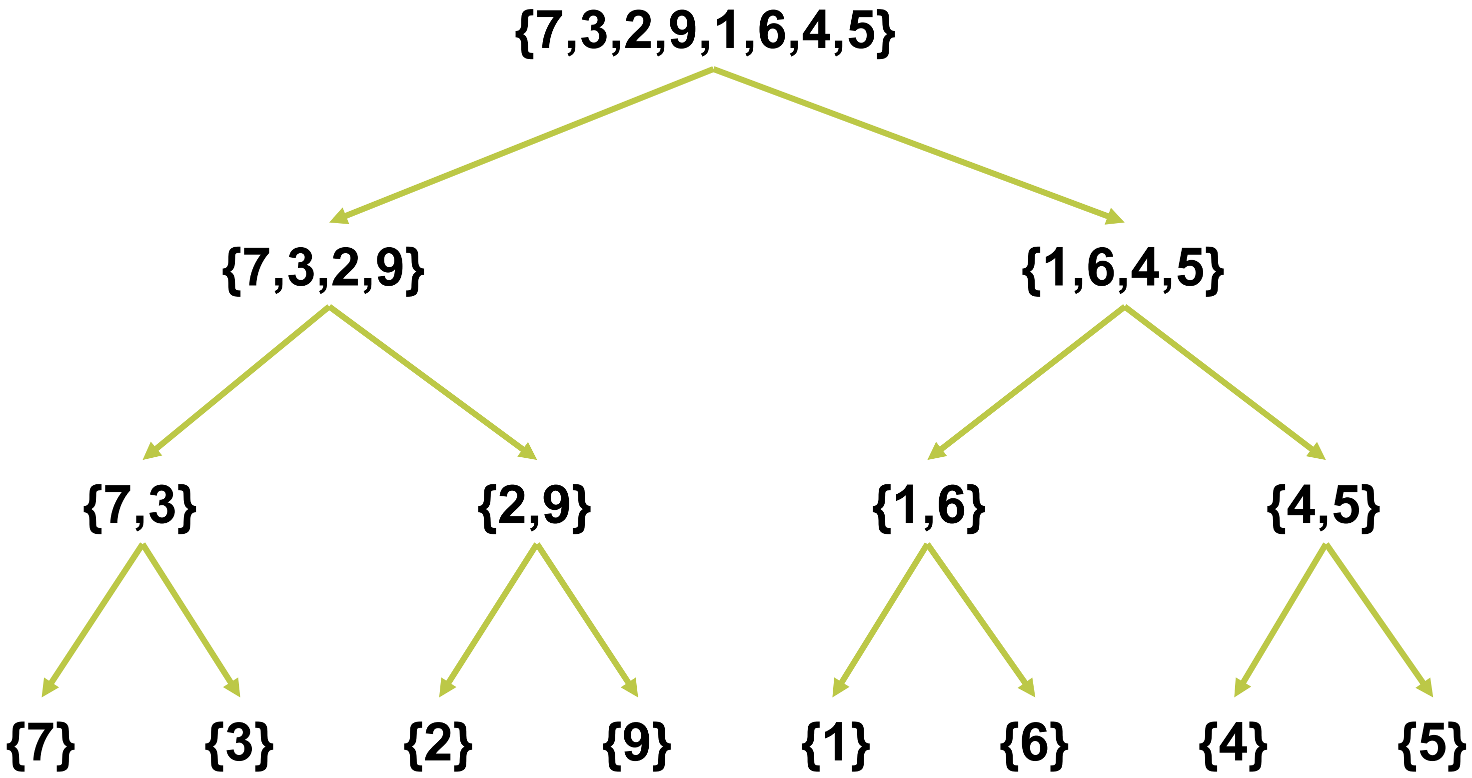
```
public void mergesort(Comparable[] theArray, int first, int last){
   // Sorts the items in an array into ascending order.
   // Precondition: theArray[first..last] is an array.
   // Postcondition: theArray[first..last] is a sorted permutation
   if (first < last) {
          int mid = (first + last) / 2;  // midpoint of the array
          mergesort(theArray, first, mid);
          mergesort(theArray, mid + 1, last);
          merge(theArray, first, mid, last);
    }// if first >= last, there is nothing to do
}
```

O time complexity of MergeSort

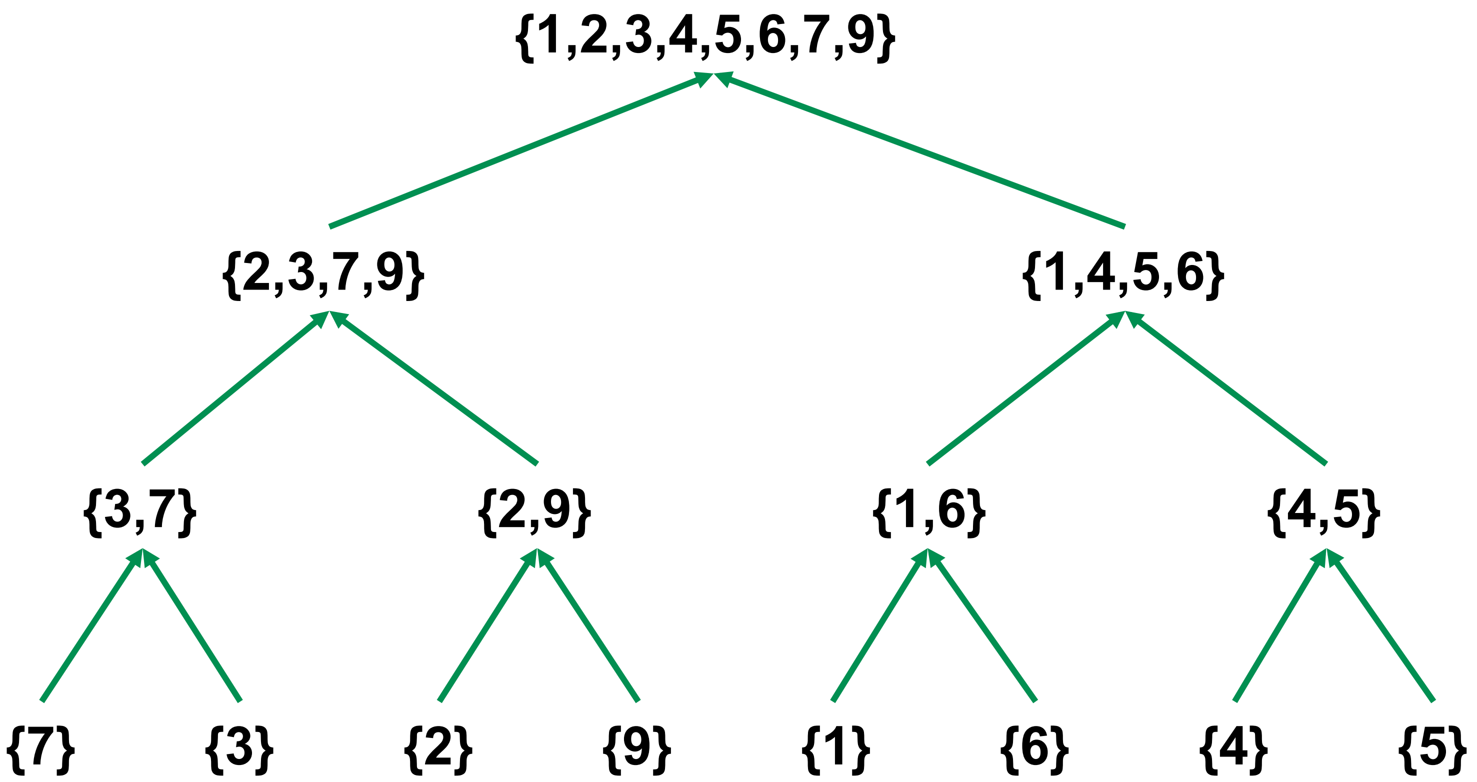
Think of the call tree for $n = 2^k$

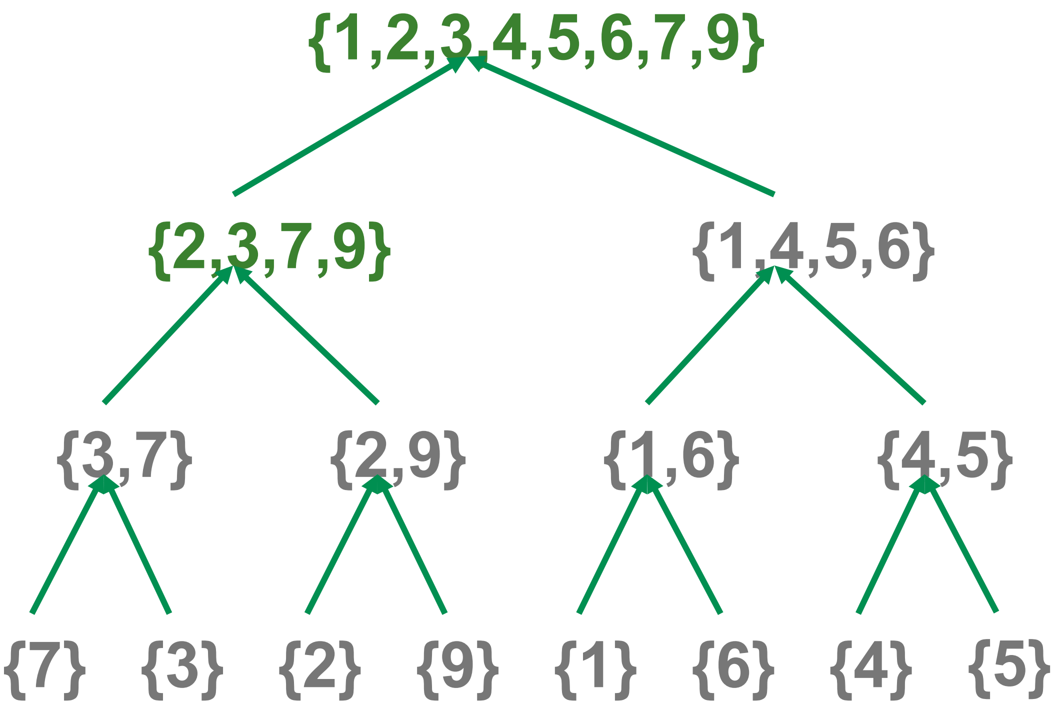
- for non powers of two we do the rounding trick

Merge Sort - Divide



Merge Sort - Merge





At depth i

- work done?

  O(n)

Total depth?

O(log n)

Total work?

O(n log n)

Data:	**2**	3	7	9	**1**	4	5	6
Temp:								

Step 1:

2	3	7	9	1	**4**	5	6
1							

Step 2:

2	**3**	7	9	1	**4**	5	6
1	2						

Step 3:

2	3	**7**	9	1	**4**	5	6
1	2	3					

Step 4:

2	3	**7**	9	1	4	**5**	6
1	2	3	4				

2	3	**7**	9	1	4	**5**	6
1	2	3	4				

Step 5:

2	3	**7**	9	1	4	5	**6**
1	2	3	4	5			

Step 6:

2	3	**7**	9	1	4	5	6
1	2	3	4	5	6		

Step 7:

2	3	7	**9**	1	4	5	6
1	2	3	4	5	6	7	

Step 8:

2	3	7	9	1	4	5	6
1	2	3	4	5	6	7	9

Merge code I

```
private void merge (Comparable[] theArray, Comparable[]
   tempArray, int first, int mid, int last({

  int first1 = first;
  int last1 = mid;
  int first2 = mid+1;
  int last2 = last;
  int index = first1;    // incrementally creates sorted array

  while ((first1 <= last1) && (first2 <= last2)){
    if( theArray[first1].compareTo(theArray[first2])<=0) {
      tempArray[index] = theArray[first1];
      first1++;
    }
    else{
      tempArray[index] = theArray[first2];
      first2++;
    }
    index++;
  }
```

Merge code II

```
// finish off the two subarrays, if necessary
while (first1 <= last1){
  tempArray[index] = theArray[first];
  first1++;
  index++;  }
while(first2 <= last2)
  tempArray[index] = theArray[first2];
  first2++;
  index++;  }
// copy back
for (index = first; index <= last: ++index){
  theArray[index ] = tempArray[index];
}
```

Mergesort Complexity

- Analysis
  - Merging:
    - for total of n items in the two array segments, at most $n - 1$ comparisons are required.
    - n moves from original array to the temporary array.
    - n moves from temporary array to the original array.
    - Each merge step requires $O(n)$ steps

Mergesort: More complexity

- Each call to `mergesort` recursively calls itself **twice**.
- Each call to `mergesort` **divides** the array into two.
  - First time: divide the array into 2 pieces
  - Second time: divide the array into 4 pieces
  - Third time: divide the array into 8 pieces
- How many times can you divide n into 2 before it gets to 1?

Mergesort Levels

- If n is a power of 2 (i.e. $n = 2^k$), then the recursion goes $k = \log_2 n$ levels deep.
- If n is not a power of 2, there are $1 + \log_2 n$ (rounded down) levels of recursive calls to `mergesort`.

Mergesort Operations

- At level 0, the original call to `mergesort` calls merge once. (O(n) steps) At level 1, two calls to `mergesort` and each of them will call `merge`, total O(n) steps
- At level m, 2^m <= n calls to `merge`
  - Each of them will call merge with $n/2^m$ items and each of them requires $O(n/2^m)$ operations. Together, $O(n) + O(2^m)$ steps, where 2^m<=n, hence O(n) work at each level
- Because there are O($log_2 n$) *levels* , total ***O(n log n)*** work

Mergesort Computational Cost

- `mergesort` is $O(n*\log_2 n)$ in both the **worst** and **average** cases (it always does n comparisons and 3 n moves at each level).

- **Significantly faster** than $O(n^2)$ (as in bubble, insertion, selection sorts)

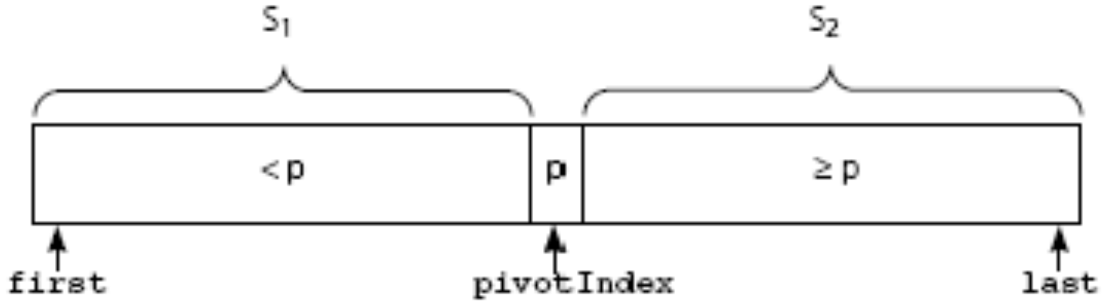
Stable Sorting Algorithms

- Suppose we are sorting a database of users according to their name. Users can have identical names.

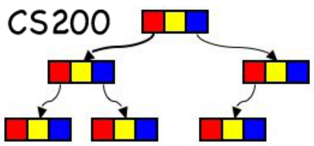
- *A* stable sorting algorithm maintains the relative order of records with equal keys (i.e., sort key values). Stability: whenever there are two records *R* and *S* with the same key and *R* appears before *S* in the original list, *R* will appear before *S* in the sorted list.
- Is mergeSort stable? What do we need to check?

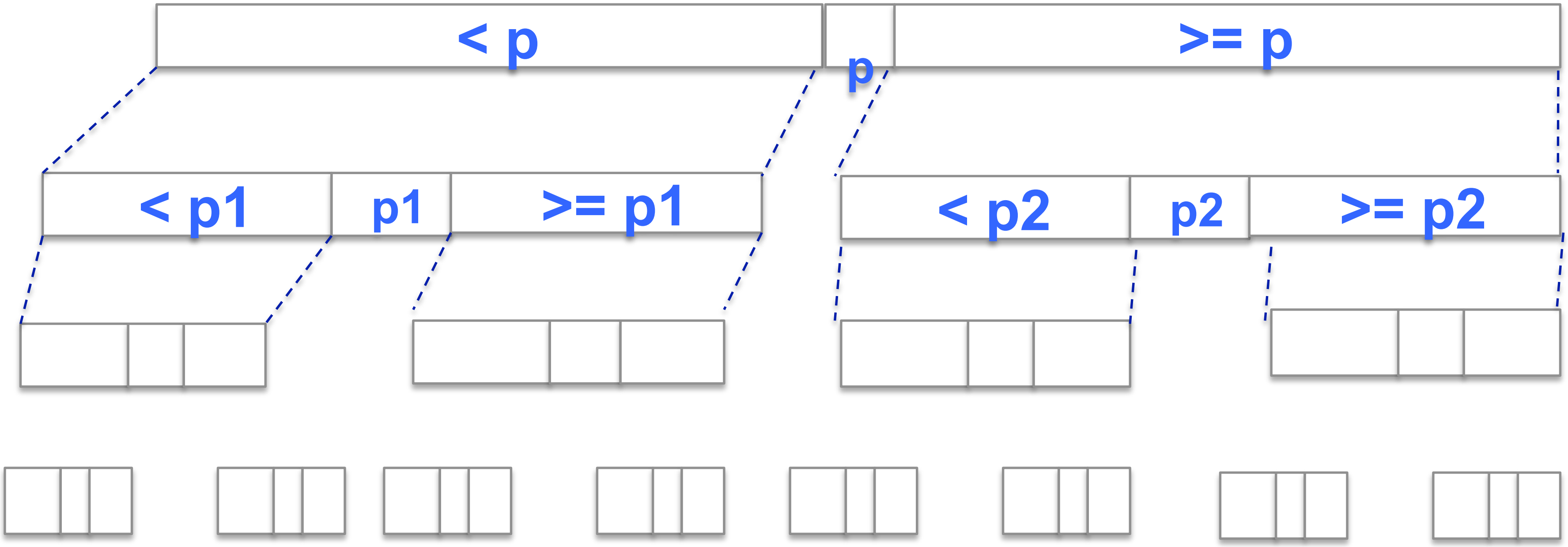
Quicksort

1. Select a **pivot** item.
2. Partition array into 3 parts
   - **Pivot in its "sorted" position**
   - Subarray with **elements < pivot**
   - Subarray with **elements >= pivot**
3. **Recursively** apply to each sub-array



Quicksort Key Idea: Pivot





Question

- An invariant for the QuickSort code is:

A. After the first pass, the P< partition is fully sorted.

B. After the first pass, the P>= partition is fully sorted.

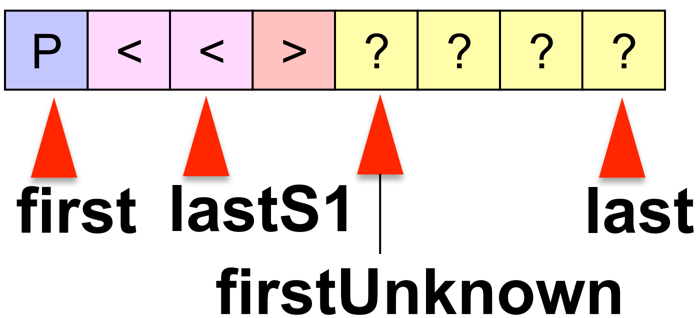
C. After each pass, the pivot is in the correct position.

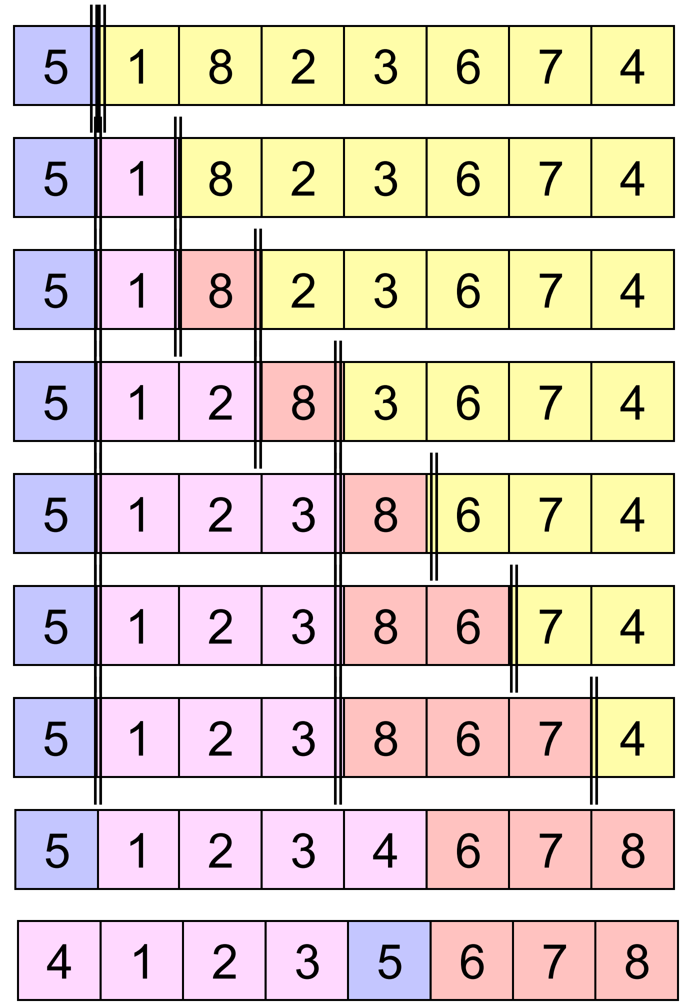
D. It has no invariant.

QuickSort Code

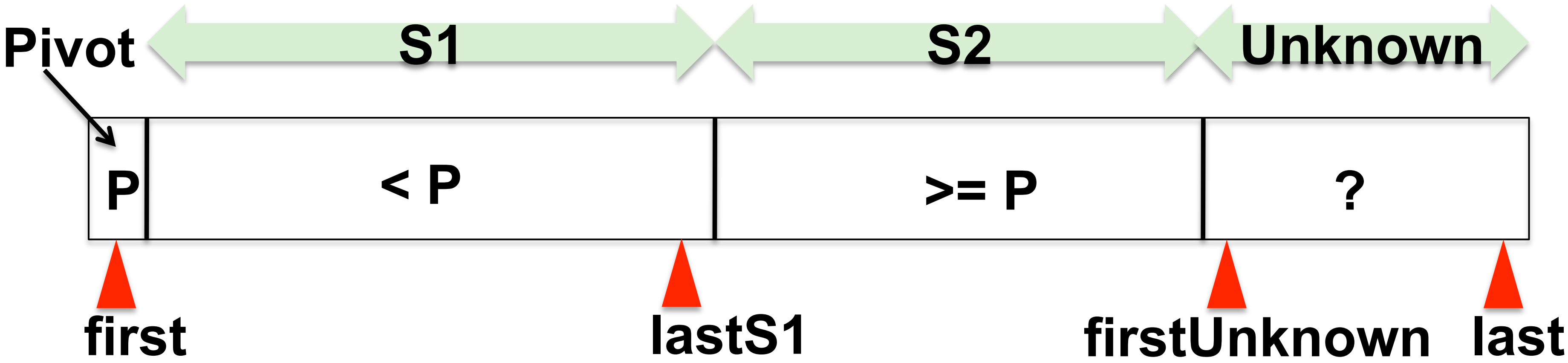
```
public void quickSort(Comparable[] theArray, int first,  int last) {
     int pivotIndex;
     if (first < last) {
     // create the partition: S1, Pivot, S2
        pivotIndex = partition(theArray, first, last);
     // sort regions S1 and S2
        quickSort(theArray, first, pivotIndex-1);
        quickSort(theArray, pivotIndex+1, last);
      }
   }
```

Quick Sort - Partitioning

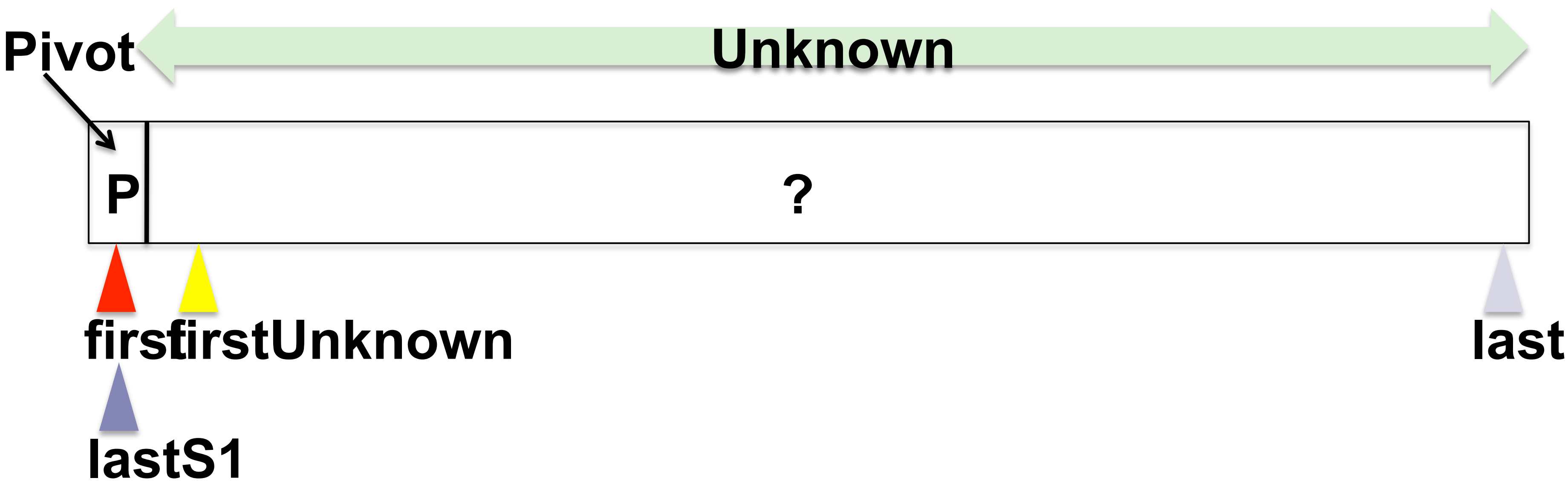




Invariant for partition



Initial state of the array



Partition Overview

1. Choose and position pivot
2. Take a pass over the current part of the array
    1. If item < pivot, move to S1 by incrementing S1 last position and swapping item into beginning of S2
    2. If item >= pivot, leave where it is
3. Place pivot in between S1 and S2

Partition Code: the Pivot

```
private int partition(Comparable[] theArray, int first, int last) {
        Comparable tempItem;
    // place pivot in theArray[first]
    // by default, it is what is in first position
      choosePivot(theArray, first, last);
      Comparable pivot = theArray[first]; // reference pivot
    // initially, everything but pivot is in unknown
      int lastS1 = first; // index of last item in S1
```

Partition Code: Segmenting

```
// move one item at a time until unknown region is empty
    for (int firstUnknown = first + 1; firstUnknown <= last; ++firstUnknown)
    {// move item from unknown to proper region
        if (theArray[firstUnknown].compareTo(pivot) < 0) {
            // item from unknown belongs in S1
            ++lastS1;   // figure out where it goes
            tempItem = theArray[firstUnknown];  // swap it with first unknown
            theArray[firstUnknown] = theArray[lastS1];
            theArray[lastS1] = tempItem;
            } // end if
       // else item from unknown belongs in S2 – which is where it is!
    } // end for
```

Partition Code: Replace Pivot

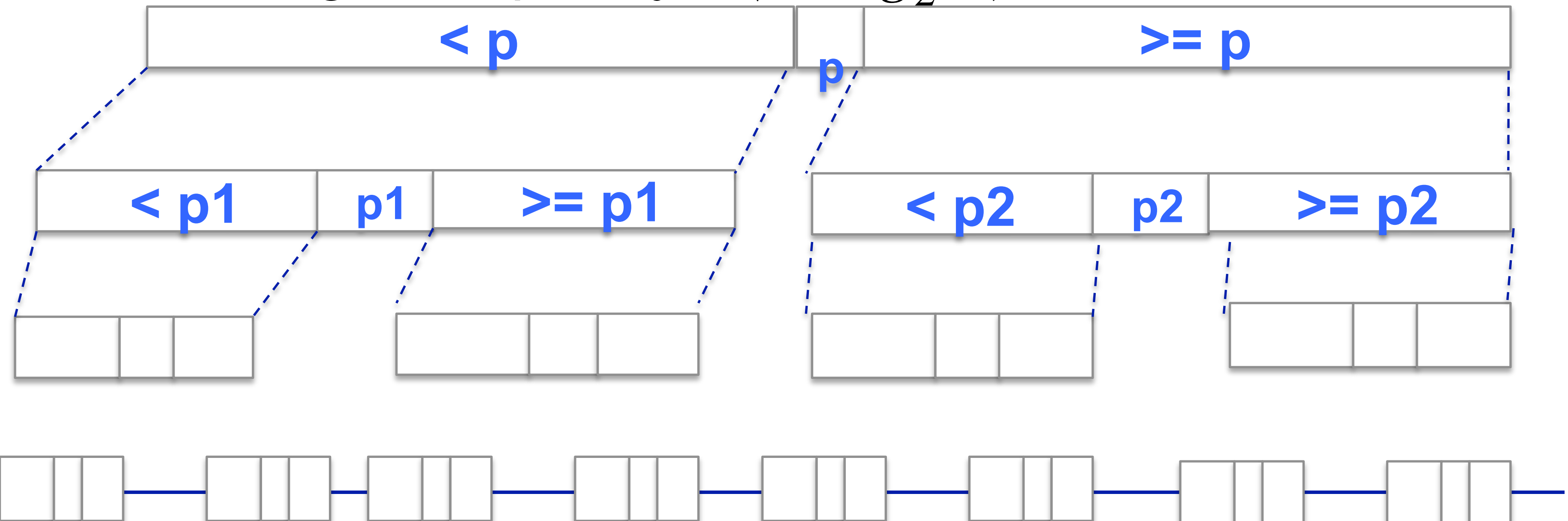
```
// place pivot in proper position and mark its location
   tempItem = theArray[first];
   theArray[first] = theArray[lastS1];
   theArray[lastS1] = tempItem;
   return lastS1;
 } // end partition
```

Quicksort Visualizations

- http://en.wikipedia.org/wiki/Quicksort
- http://www.sorting-algorithms.com
- Hungarian Dancers via YouTube

Average Case

- Each **level** involves,
  - Maximum $(n-1)$ comparisons.
  - Maximum $(n-1)$ swaps. $(3(n-1)$ data movements)
  - $log_2\, n$ levels are required.
- Average complexity $O(n\ log_2\ n)$



Question

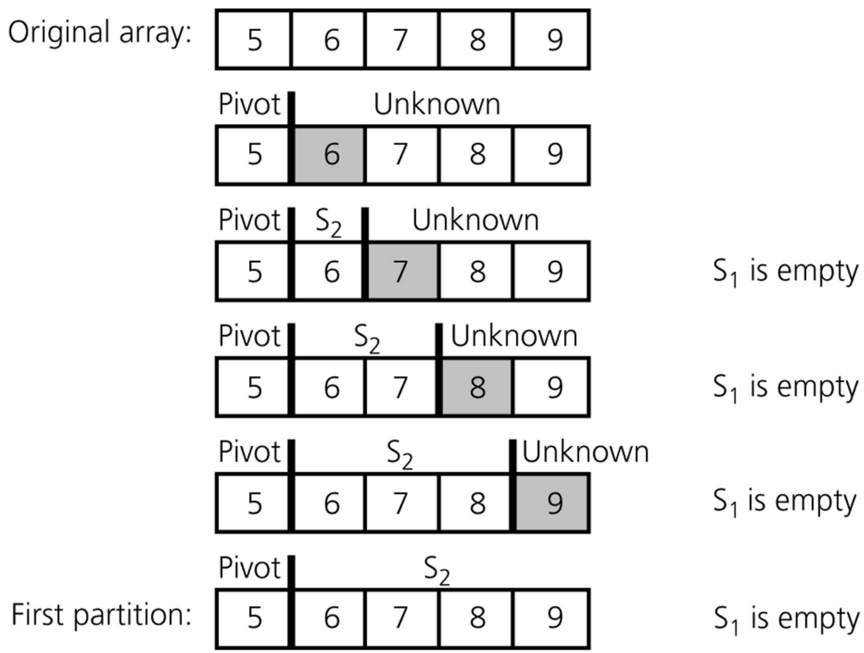
- Is QuickSort like MergeSort in that it is always O(nlogn) complexity?

A. Yes

B. No

When things go bad…

- Worst case
  - `quicksort` is $O(n^2)$ when every time the smallest item is chosen as the pivot (e.g. when it is sorted)



Worst case analysis

- This case involves $(n\text{-}1)+(n\text{-}2)+(n\text{-}3)+\ldots+1+0 = n(n\text{-}1)/2$ comparisons

- `Quicksort` is $O(n^2)$ for the **worst-case**.

Strategies for Selecting **pivot**

- First value: worst case if the array is **sorted**.
- If we look at only one value, whatever value we pick, we can and up in the worst case (if it is the minimum).
- Median of 3 sample values
  - Worst case $O(n^2)$ can still happen
  - but less likely

quickSort – Algorithm Complexity

- Depth of call tree?
    - O(log n) *split roughly in half,* best case
    - O(n) worst case
- Work done at each depth
    - O(n)
- Total Work
    - O(n log n) best case
    - $O(n^2)$ worst case

Clicker Q

- Why would someone pick QuickSort over MergeSort?

A. Less space

B. Better worst case complexity

C. Better average complexity

D. Lower multiplicative constant in ava=erage complexity

How fast can we sort?

- Observation: all the sorting algorithms so far are *comparison sorts*
  - A comparison sort must do at least O(n) comparisons (*why?*)
  - We have an algorithm that works in O(n log n)
  - What about the gap between O(n) and O(n log n)
- **Theorem (cs 420):**

  all comparison sorts are Ω(n log n)

- MergeSort is therefore an “optimal” algorithm

Radix Sort (by MSD)

0. Represent all numbers with the same number of digits
1. Take the most significant digit (MSD) of each number.
2. Sort the numbers based on that digit, grouping elements with the same digit into one bucket.
3. Recursively sort each bucket, starting with the next digit to the right.
4. Concatenate the buckets together in order.

80 24 62 40 68 20 26

	24, 20, 26	40	62, 68	80

20		24	26		40						62			68	80				

Radix sort

- Analysis
  - n moves each time it forms groups
  - n moves to combine them again into one group.
  - Total $2n*d$ (for the strings of d characters)

Radix Sort

- Radix sort is
  - Fast
  - Asymptotically fast
  - Simple to code
  - A good choice
- Can we use it for strings?
- So why not use it for every application?