Divide and Conquer Algorithms: Advanced Sorting

Prichard Ch. 10.2: Advanced Sorting Algorithms

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 *(or (n-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

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 $n = 2^k$

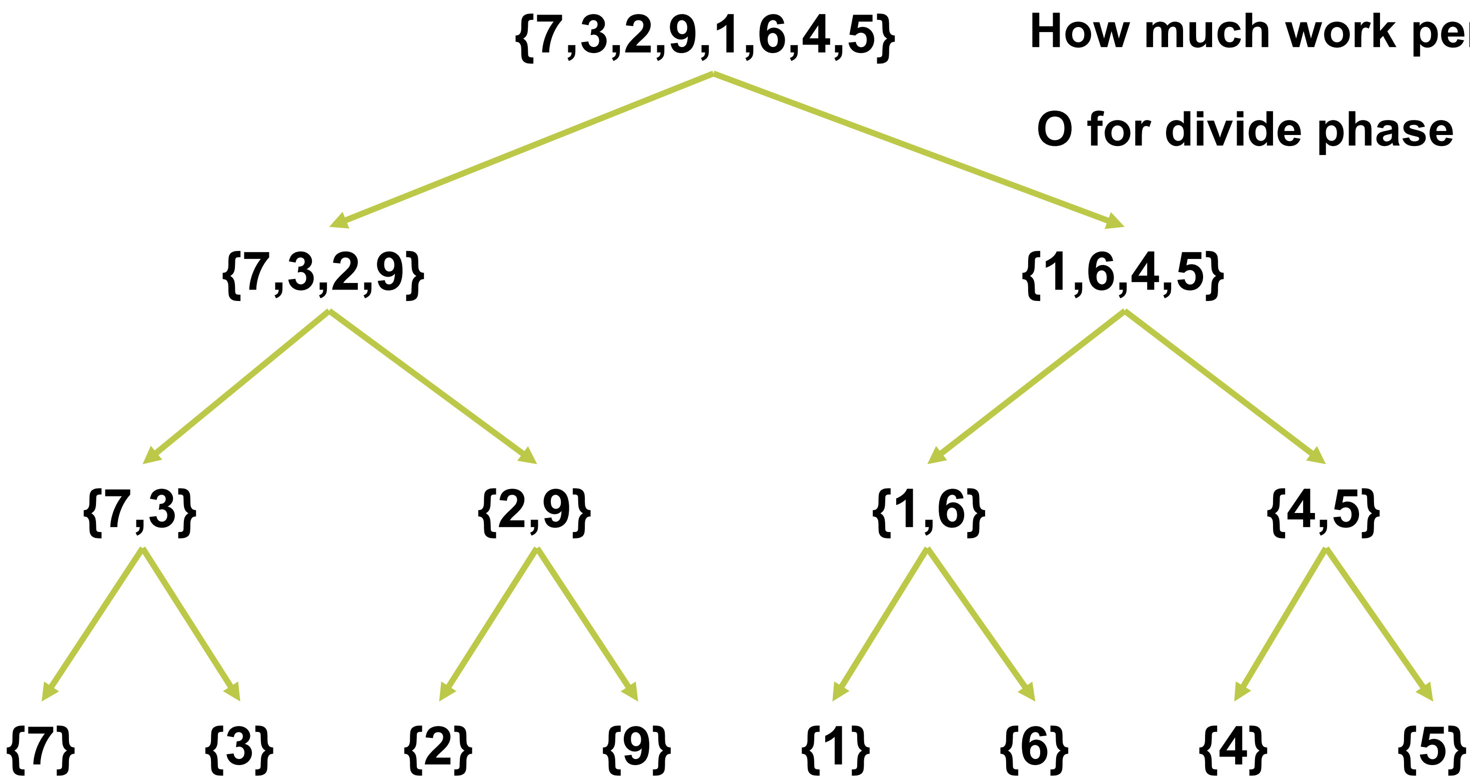
- for non powers of two we round to next 2^k
- same O

Merge Sort - Divide

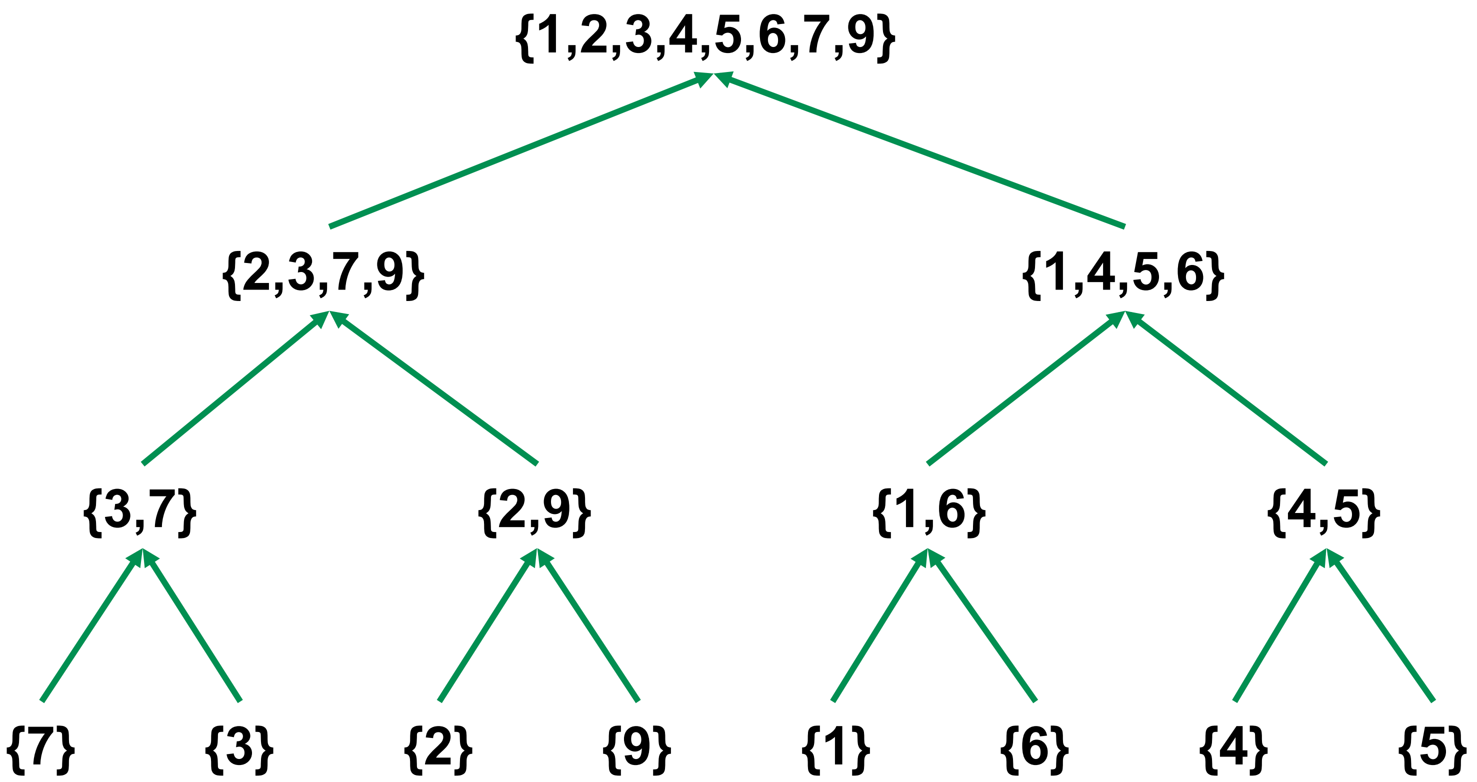
How many divides ?

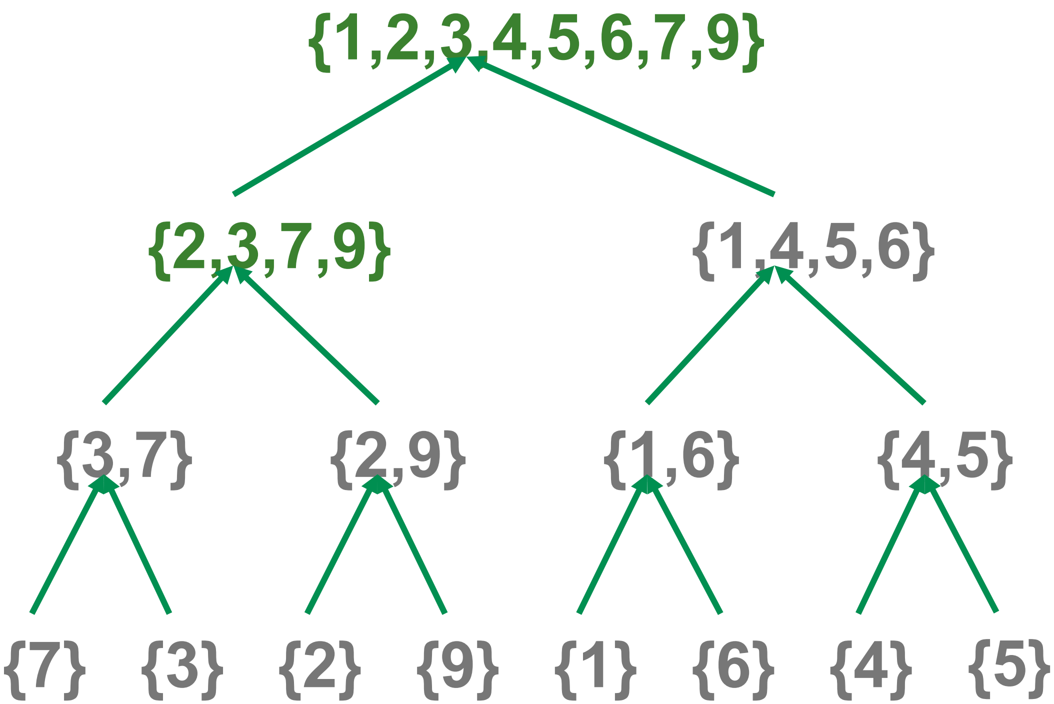
How much work per divide ?

O for divide phase ?



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				

TOP MERGE PHASE

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

TOP MERGE PHASE

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

$$\text{ceiling}(\log_2 n)$$

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.

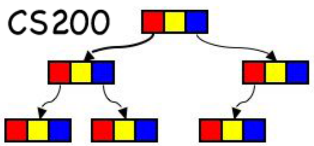
- **Significantly faster** than $O(n^2)$ (as in bubble, insertion, selection sorts)

Stable Sorting Algorithms

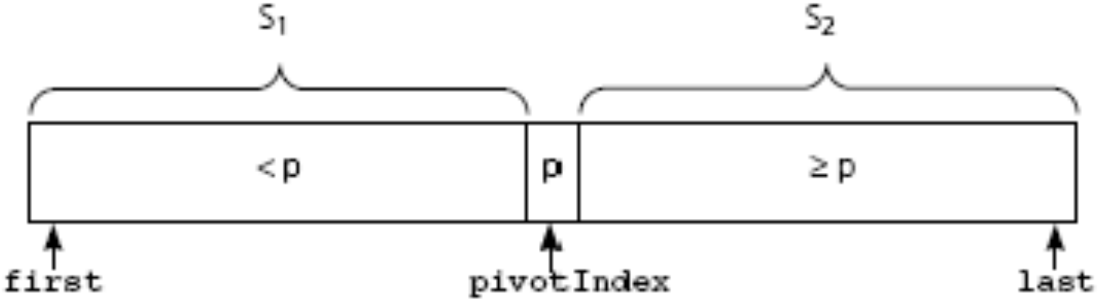
- Suppose we are sorting a database of users according to their name. Users can have identical names.

- *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?

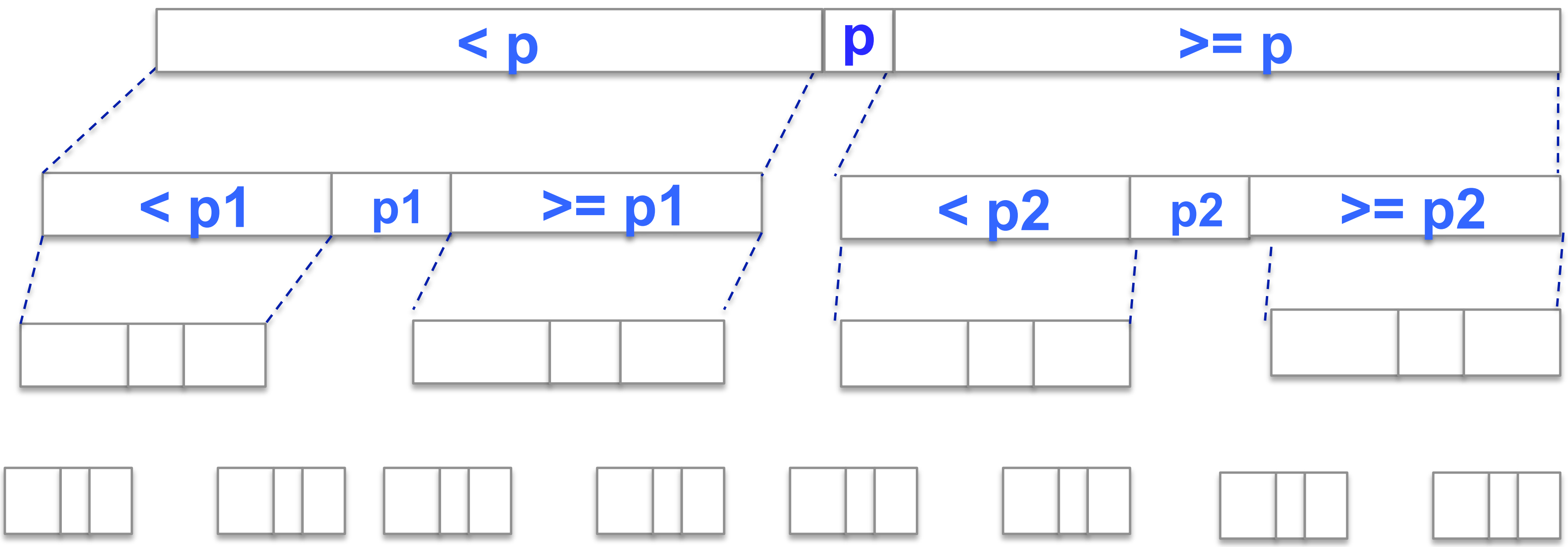
Quicksort



1. Select a **pivot** element from the array. Often just the first element
2. Partition array into 3 parts
   - **Pivot in its "sorted" position**
   - Subarray with **elements < pivot**
   - Subarray with **elements >= pivot**
3. **Recursively** sort 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.

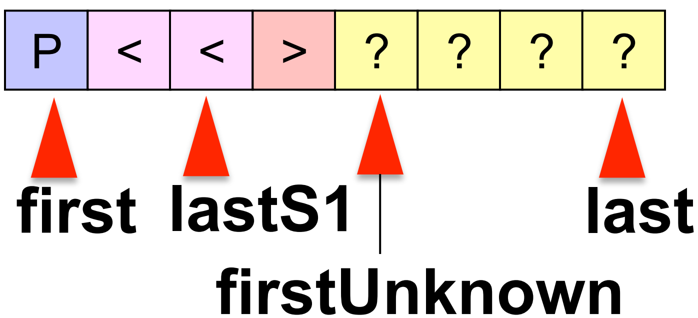
C. After each pass, the pivot is in the correct position.

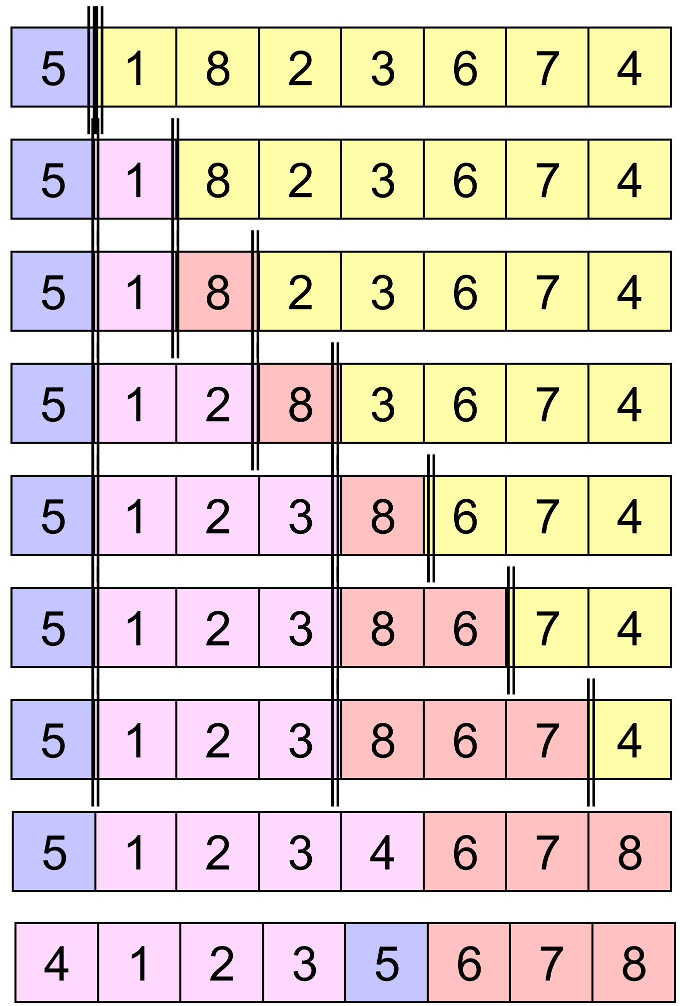
D. It has no invariant.

QuickSort Code

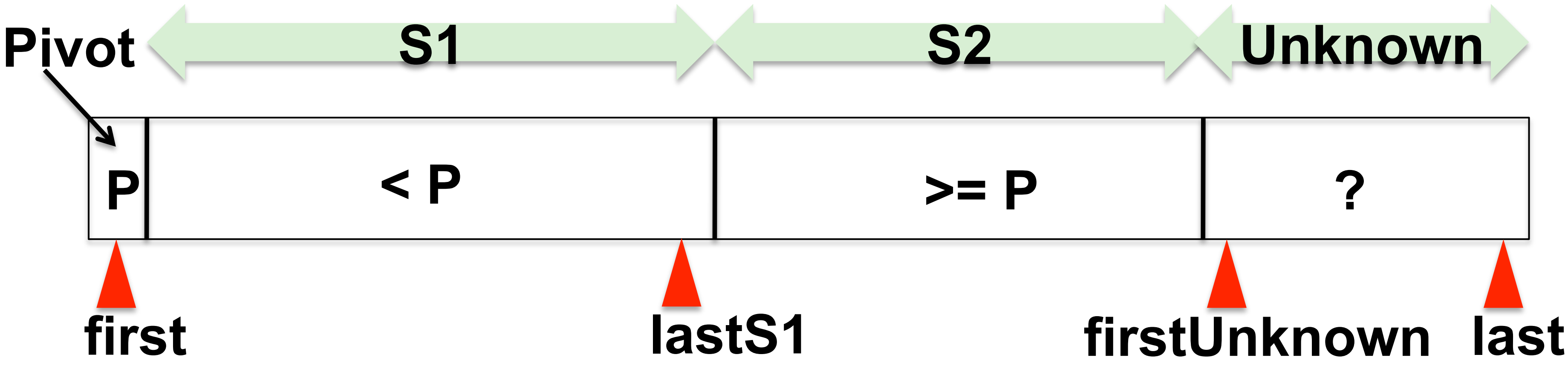
```
public void quickSort(Comparable[] theArray, int first,  int last) {
      int pivotIndex;
      if (first < last) {
      // create the partition: S1, Pivot, S2
        pivotIndex = partition(theArray, first, last);
      // sort regions S1 and S2
        quickSort(theArray, first, pivotIndex-1);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

Quick Sort - Partitioning

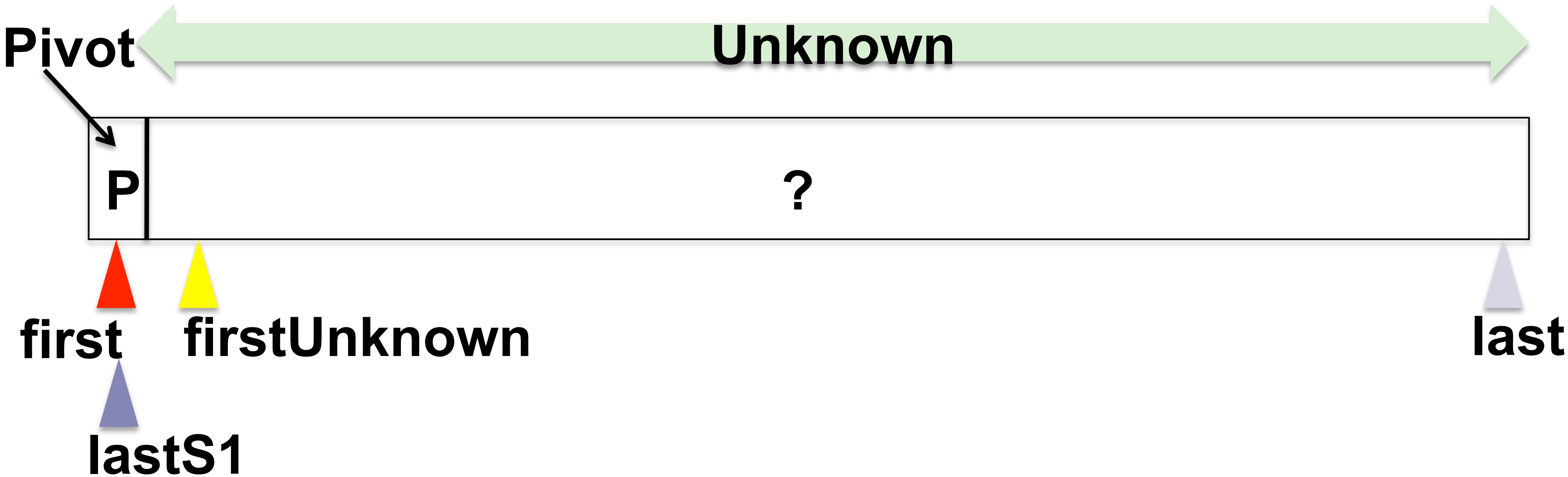




Invariant for partition



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 item to S1 by incrementing S1 last position and swapping item into end of S1
   2. If item >= pivot, leave where it is making it part of S2
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

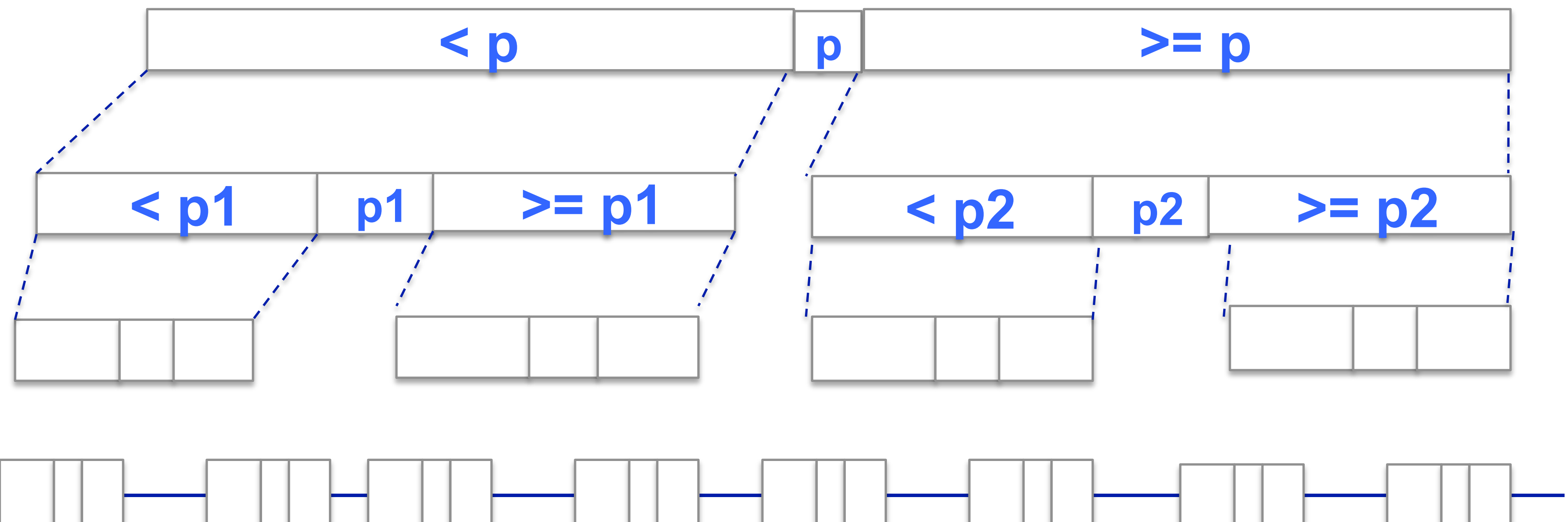
```
// move one item at a time until unknown region is empty
for (int firstUnknown = first + 1; firstUnknown <= last; ++firstUnknown)
{// move item from unknown to proper region
    if (theArray[firstUnknown].compareTo(pivot) < 0) {
        // item from unknown belongs in S1
        ++lastS1;   // figure 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
```

Average Case

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



Question

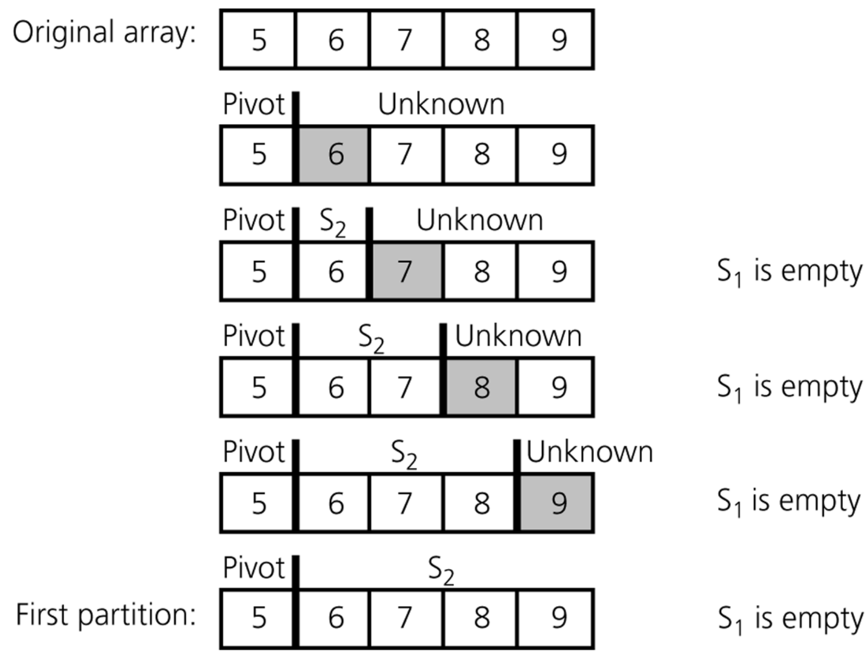
- Is QuickSort like MergeSort in that it is always O(nlogn) complexity?

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

Question

- Why would someone pick QuickSort over MergeSort?

A. Less space

B. Better worst case complexity

C. Better average complexity

D. Lower multiplicative constant in average 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)

- Why not use it for every application?