Comparison of Sorting Algorithms

Sorting Algorithm	Best Case time	Worst Case time	Avg. Case time
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$

Internal Sort	Yes
External Sort	No
Stable Sort	Yes
In Place	Yes

Divide and Conquer Sorting Algorithms

Divide and Conquer (D&C)

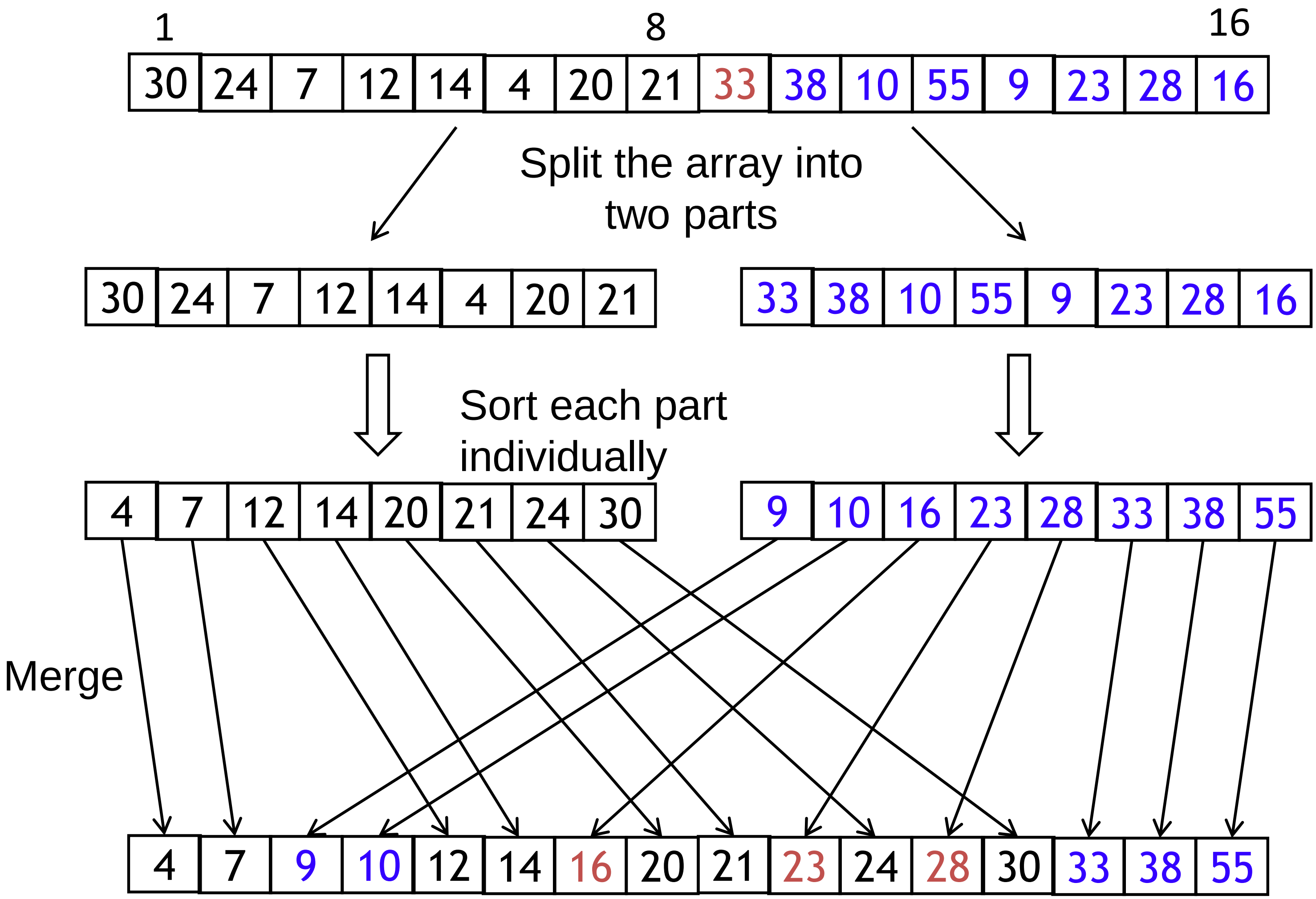
- Divide the problem into a number of subproblems
- Conquer (solve) each subproblem independently
  - Solve them **recursively**
  - There must be base case (to stop recursion)
- Combine (merge) solutions to subproblems into a solution to the original problem

Divide and Conquer Examples

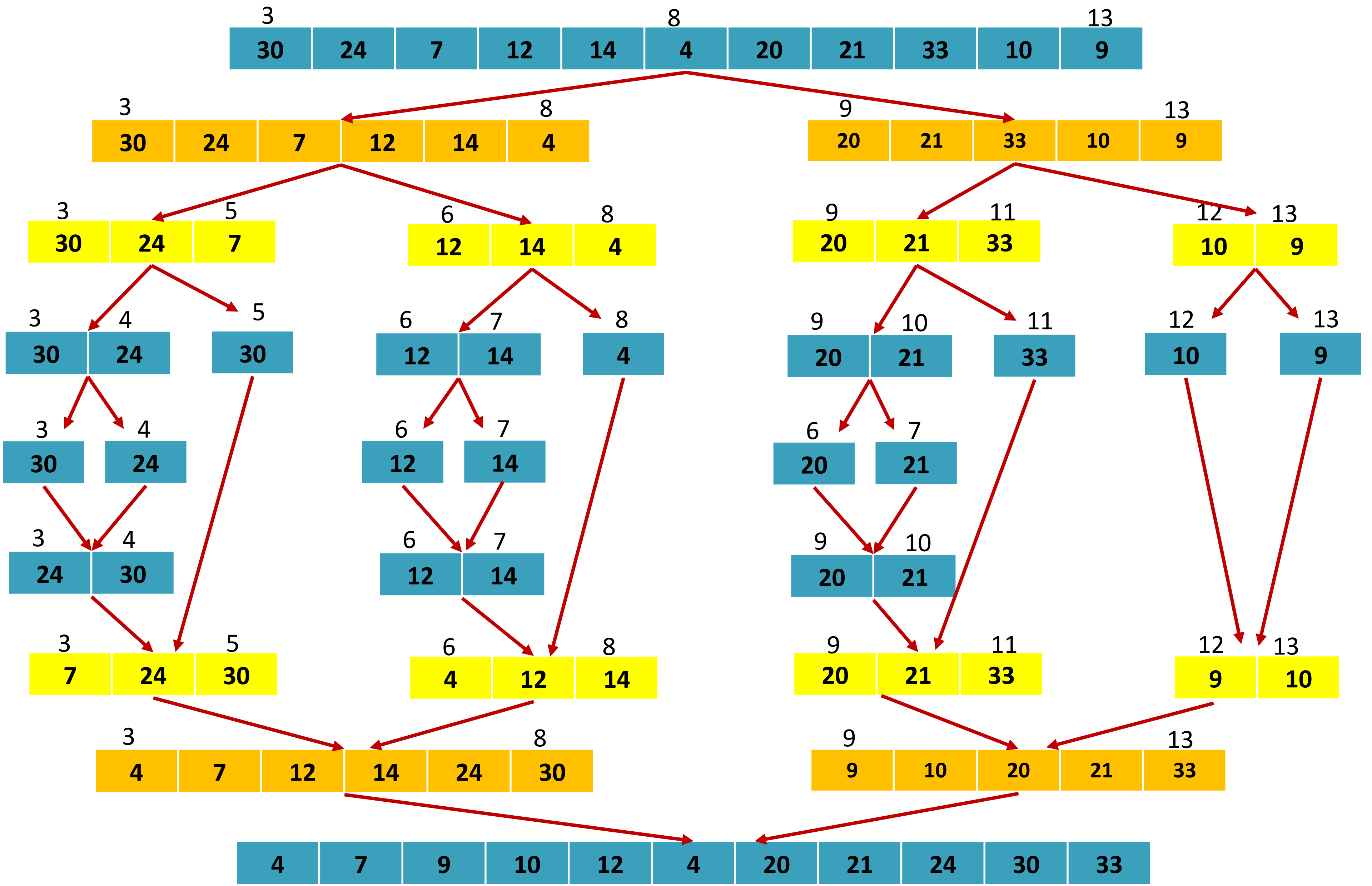
- **E.g. 1**: Divide array into two halves, *recursively* sort both halves, then *merge* the two halves → Merge sort

- **E.g. 2**: Partition array into items that are "small" and items that are "large", then recursively sort the two sets → Quick sort

Merge sort


1
8
16
30 24 7 12 14 4 20 21 33 38 10 55 9 23 28 16
Split the array into two parts
30 24 7 12 14 4 20 21
33 38 10 55 9 23 28 16
Sort each part individually
4 7 12 14 20 21 24 30
9 10 16 23 28 33 38 55
Merge
4 7 9 10 12 14 16 20 21 23 24 28 30 33 38 55

Merge sort: Conquer



Merge Sort code

	low															high
input_array	30	24	7	12	14	4	20	21	33	38	10	55	9	23	28	16
temp_array																
									low					high		

```
sort(input_array, temp_array, First_index, Last_index);
```

```
void sort(int list[], int temp[], int low, int high)
{
     if( low < high)
       {
              int center = (low + high) / 2;      Divide
              sort(list, temp, low, center);
              sort(list, temp, center + 1, high);  Conquer
              merge(list, temp, low, center + 1, high);   Combine
        }
}
```

Merge Function

```
void merge(int list[], int temp[], int low, int mid, int high)
 {
    int leftPos = low, leftEnd = mid - 1;
    int rightPos = mid, rightEnd = high;
    int tempPos = 0;
    int numElements = high - low + 1;

    while( leftPos <= leftEnd && rightPos <= rightEnd)
     {
       if( list[leftPos].compareTo(list[rightPos]) <= 0)
         {
               temp[ tempPos ] = list[ leftPos ];
               leftPos++;
          }
       else{
               temp[ tempPos ] = list[ rightPos ];
               rightPos++;
             }
       tempPos++;
      }
```

Continue…….

Merge Function (Cont…)

```
//copy rest of left half
while( leftPos <= leftEnd)
 {      temp[ tempPos ] = list[ leftPos ];
        tempPos++;
        leftPos++;
  }

//copy rest of right half
while( rightPos <= rightEnd)
{       temp[ tempPos ] = list[ rightPos ];
        tempPos++;
        rightPos++;
 }

//Copy temp elements back into the list
 k = low;
for(int i = 0; i < numElements; i++, k++)
        list[ k ] = temp[ i ];
}
```

Analyzing Merge Sort

1. **Divide**: divide the given n-element array A into 2 subarrays of n/2 elements each
2. **Conquer**: recursively sort the two subarrays
3. **Combine**: merge 2 sorted subarrays into 1 sorted array

- Analysis: $T(n) = \boldsymbol{\Theta(1)} + \mathbf{2T(n/2)} + \boldsymbol{\Theta(n)}$

$$= \Theta(n \log n)$$

Analyzing Merge Sort

- T(n) = Divide(n) + Combine (n) + Conquer (n)
  $= O(1) \qquad + O(n) \qquad\qquad + 2T(n/2)$

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

$= n + 2\,(n/2 + 2(T(n/4)))$

$= n + n + 2^2\, T(n/2^2)$

$= n + n + 2^2\,(n/\,2^2 + 2(T(n/\,2^3)))$

$= n + n + n + 2^3\, T(n/2^3)$

…

$= n + n + \ldots .\ i \text{ times} + 2^i\, T(n/2^i) \qquad n/2^i = 1 \Rightarrow i = \lg n$

$= n + n + \ldots . + 2^{\lg n}\, T(1)$

$= n + n + \ldots . + n\ (\lg n \text{ times}) \quad + \quad n\,T(1)$

$= n\lg n + n$

$= O(n\lg n)$

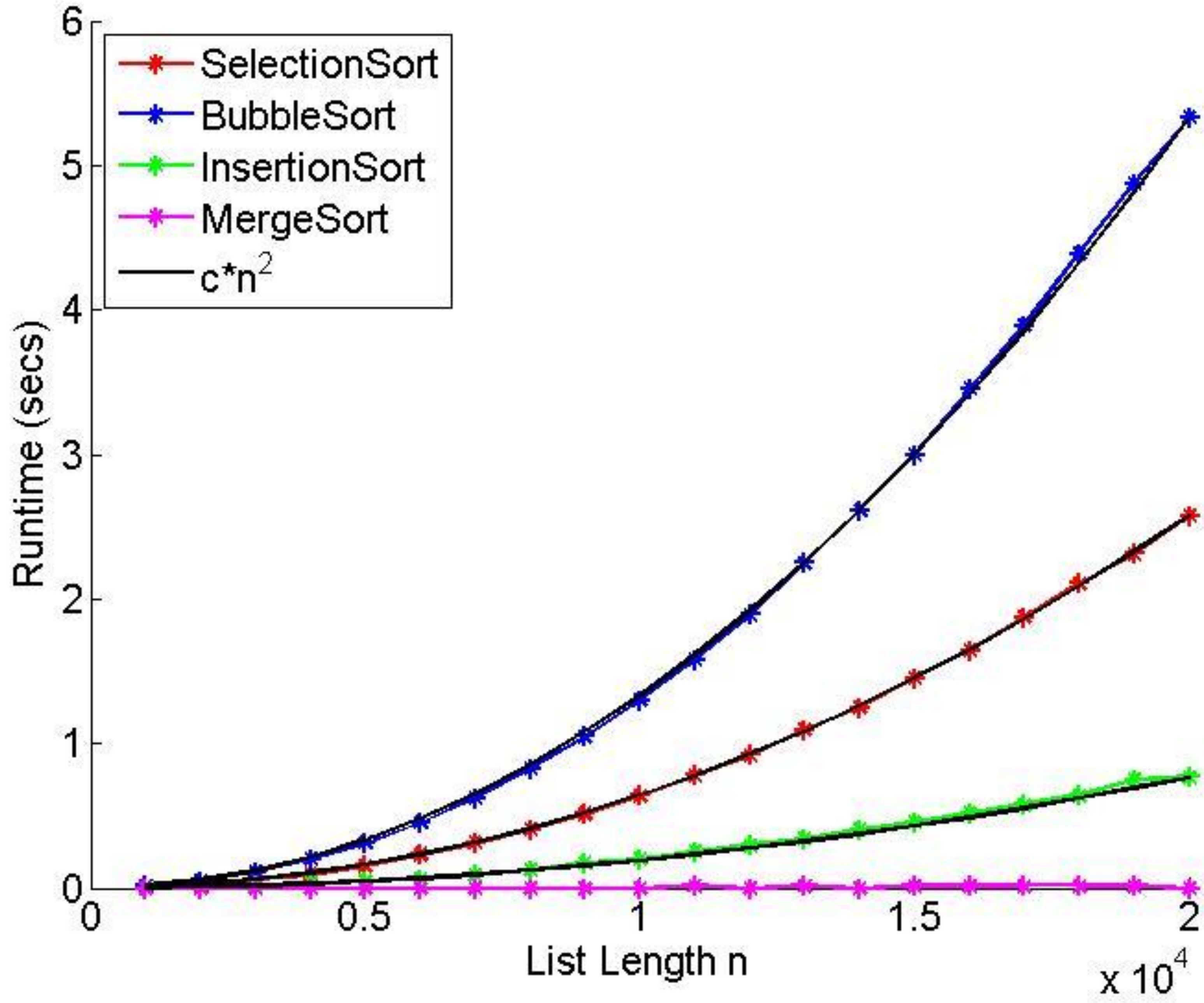
Comparison of Sorting Algorithms

Sorting Algorithm	Best Case time	Worst Case time	Avg. Case time
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Merge Sort	$\Theta(nlgn)$	$\Theta(nlgn)$	$\Theta(nlgn)$

Internal Sort	Yes
External Sort	-
Stable Sort	Yes
In Place	No

Can we write external merge sort?

Can we write in place merge sort?


SelectionSort
BubbleSort
InsertionSort
MergeSort
c*n^2
Runtime (secs)
List Length n
x 10^4
0
1
2
3
4
5
6
0.5
1.5

Why merge sort?

- It isn’t an “in place” sort - requires extra storage
- However, it doesn’t require this storage “all at once”
- This means you can use merge sort to sort something that doesn’t fit in memory—say, 300 million census records—then much of the data must be kept on backup media, such as a HDD
- Merge sort is a good way to do this

Using merge sort for large data sets

- Repeat:
  - Read in as much data as fits in memory
  - Sort it, using a fast sorting algorithm (may be quicksort)
  - Write out the sorted data to a new file
- After all the data has been written into smaller, individually sorted files:
  - Read in the initial portion of each sorted file into individual arrays
  - Start merging the arrays
  - Whenever an array becomes empty, read in more data from its file
  - Every so often, write the destination array to the (one) final output file

- When you are done, you will have one (large) sorted file

Quicksort

- Like Mergesort, Quicksort is also based on the *Divide-and-Conquer* paradigm

- But uses it in a somewhat opposite manner, as all the hard work is done *during division operation*

Quicksort (cont.)

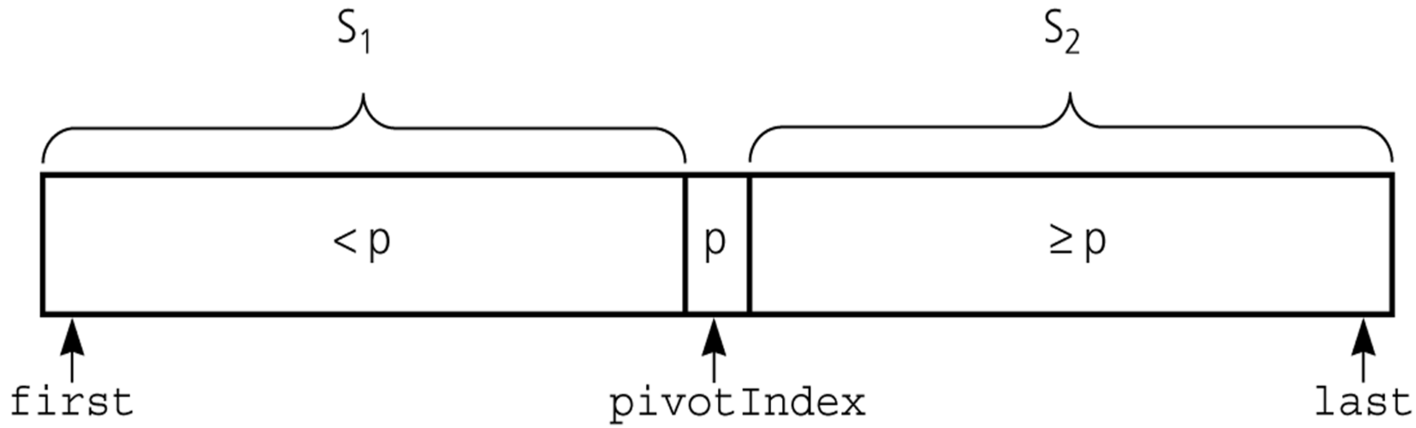
It consists of the following three steps:

1. ***Divide***: Partition the list.
   - To partition the list, we first choose an element from the list for which we hope about half the elements will come before and half after. Call this element the ***pivot***.
   - Then we partition the elements so that all those with values less than the pivot come in one sublist and all those with greater values come in another.
2. ***Conquer/Recursion***: Recursively sort the sublists separately.
3. ***Combine***: Put the sorted sublists together.

What you get after Partition?

Partition

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



- Elements around the pivot p generates two smaller sorting problems.
  - sort both of the subproblems independently
  - when these two smaller sorting problems are solved recursively, our bigger sorting problem is solved.

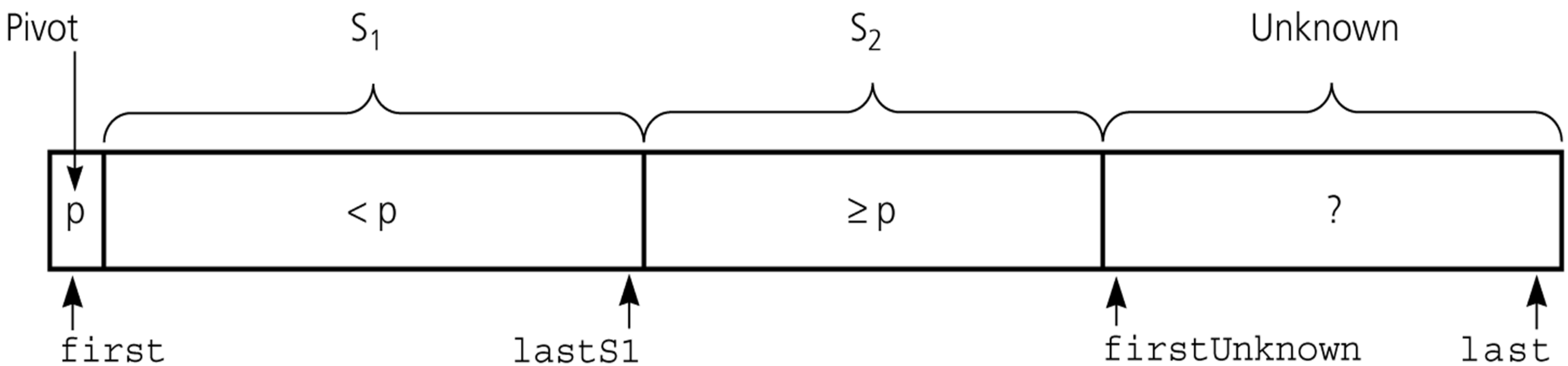
Partition – Choosing the pivot

- First, select a pivot element from the given array, and store it into the first location of the 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.
  - If the items in the array arranged randomly, we choose a pivot randomly.
  - We can choose the first or last element as a pivot (it may not give a good partitioning).
  - We can use different techniques to select the pivot.

Partition Function (cont.)

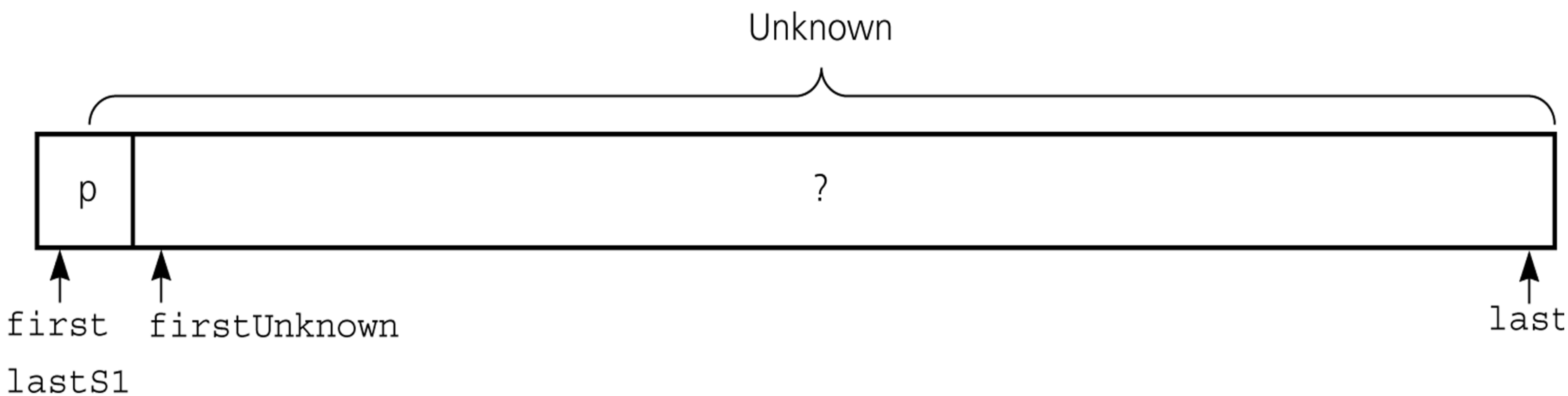
Invariant for the partition algorithm



Work with two extra variables: **lastS1** and **firstUnknown**

Partition Function (cont.)

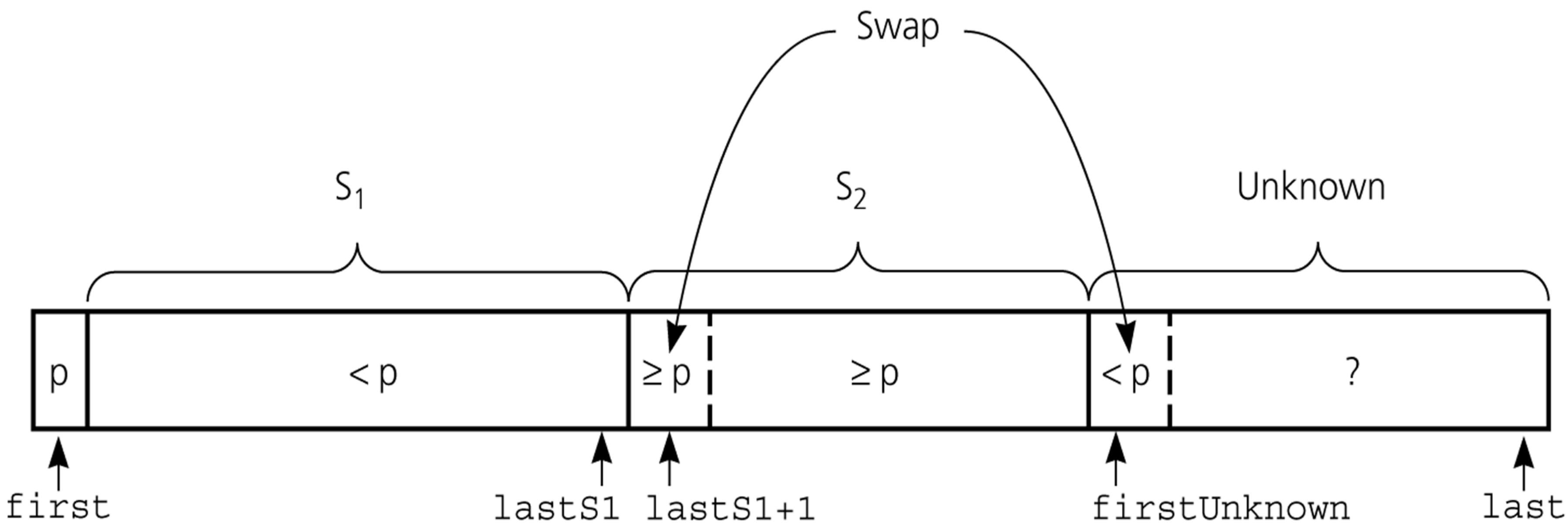
Initial state of the array



Partition Function (cont.)

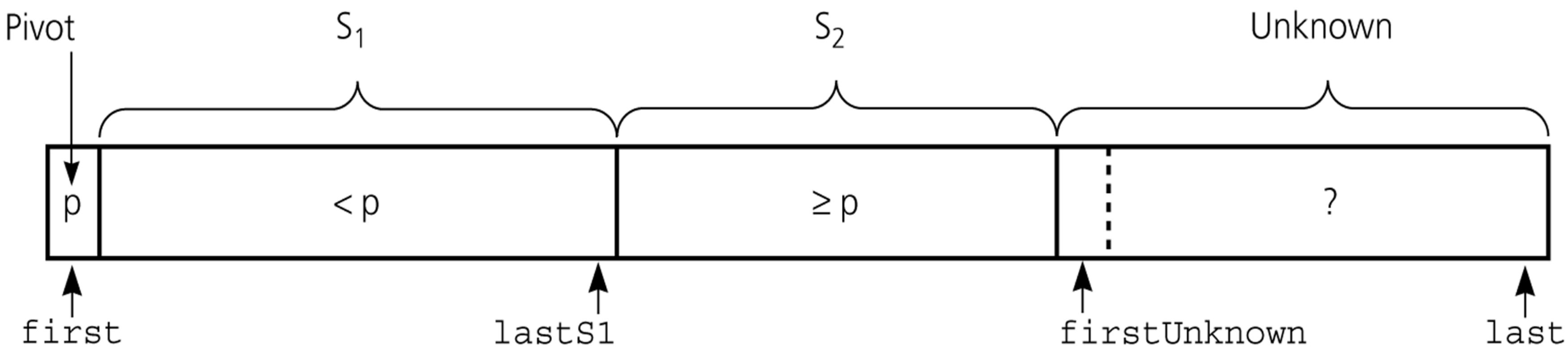
Moving theArray[firstUnknown] into S_1

by swapping it with theArray[lastS1+1]
then increment both lastS1 and firstUnknown.

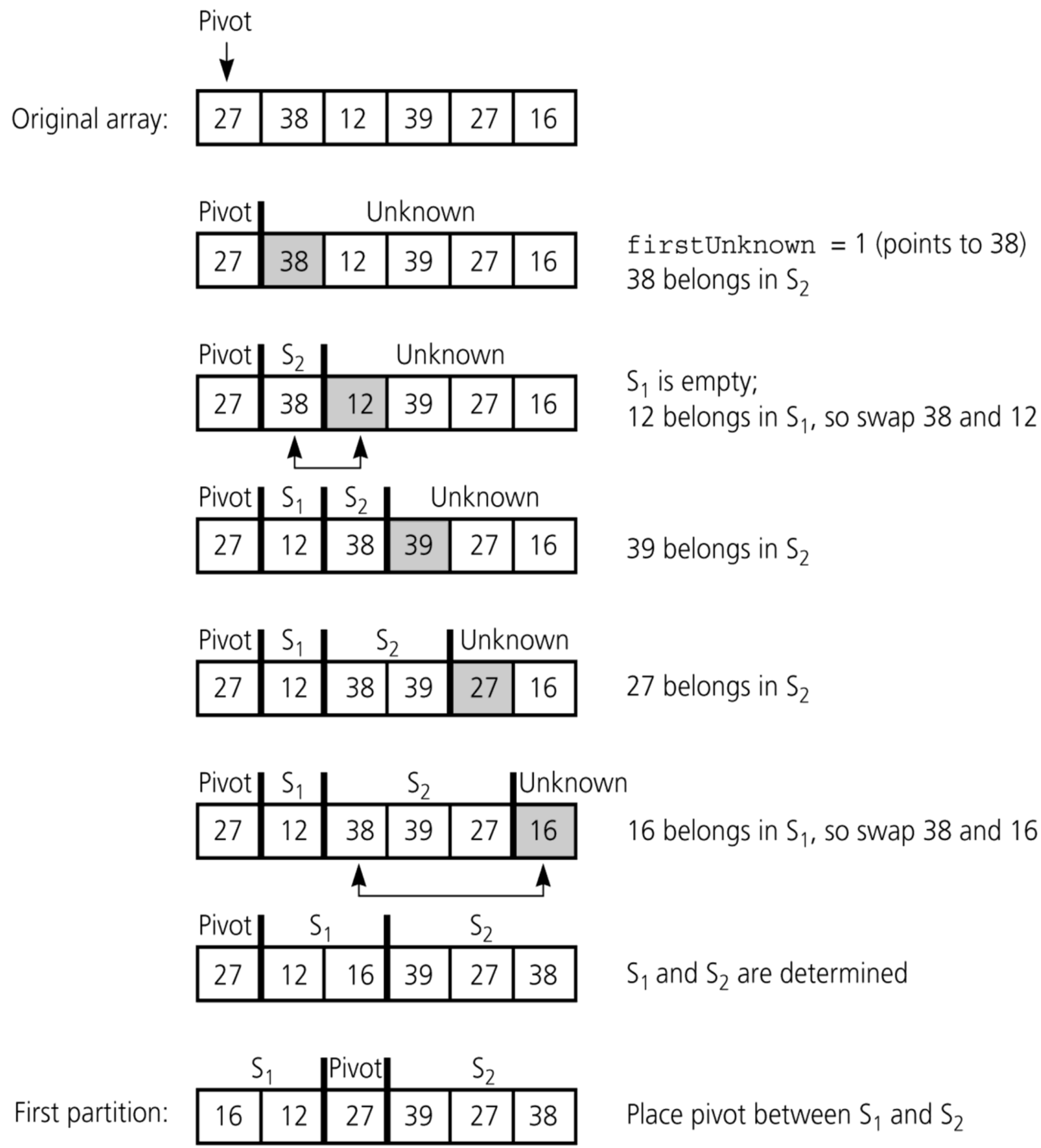


Partition Function (cont.)

Moving theArray[firstUnknown] into S_2 by incrementing firstUnknown.



Developing the first partition of an array when the pivot is the first item



Quicksort Function

```
void quicksort(int theArray[], int first, int last)
{
   // Sorts the items in an array into ascending order.

   int pivotIndex;
   if (first < last)
   {
      // create the partition: S1, pivot, S2
      partition(theArray, first, last, &pivotIndex);

      // sort subarrays S1 and S2
      quicksort(theArray, first, pivotIndex-1);
      quicksort(theArray, pivotIndex+1, last);
   }
}
```

Divide

Conquer

Partition Function

```
void partition(int theArray[], int first, int last,
              int &pivotIndex)
{
// Precondition: first < last.
// Postcondition: Partitions theArray[first..last] such that:
//    S1 = theArray[first..pivotIndex-1]
//    S2 = theArray[pivotIndex+1..last]

// place pivot in theArray[first]
   choosePivot(theArray, first, last);

   int pivot = theArray[first];
```

Partition Function (cont.)

```
   int lastS1 = first;
   int firstUnknown = first + 1;

   for (  ; firstUnknown <= last; ++firstUnknown)
   {
     if (theArray[firstUnknown] < pivot)
     {
         ++lastS1;
         swap(theArray[lastS1],theArray[firstUnknown]);
      }
    }

   swap(theArray[first], theArray[lastS1]);
   pivotIndex = lastS1;
}
```

Quicksort – Worst Case Analysis

The pivot divides the list of size n into two sublists of sizes 0 and n-1

- The number of key comparisons

  $= n-1 + n-2 + \ldots + 1$

  $= \mathbf{n^2/2 - n/2}$ ➔ $\mathbf{O(n^2)}$

- The number of swaps =

  $= \quad n-1 \quad + \quad n-1 + n-2 + \ldots + 1$

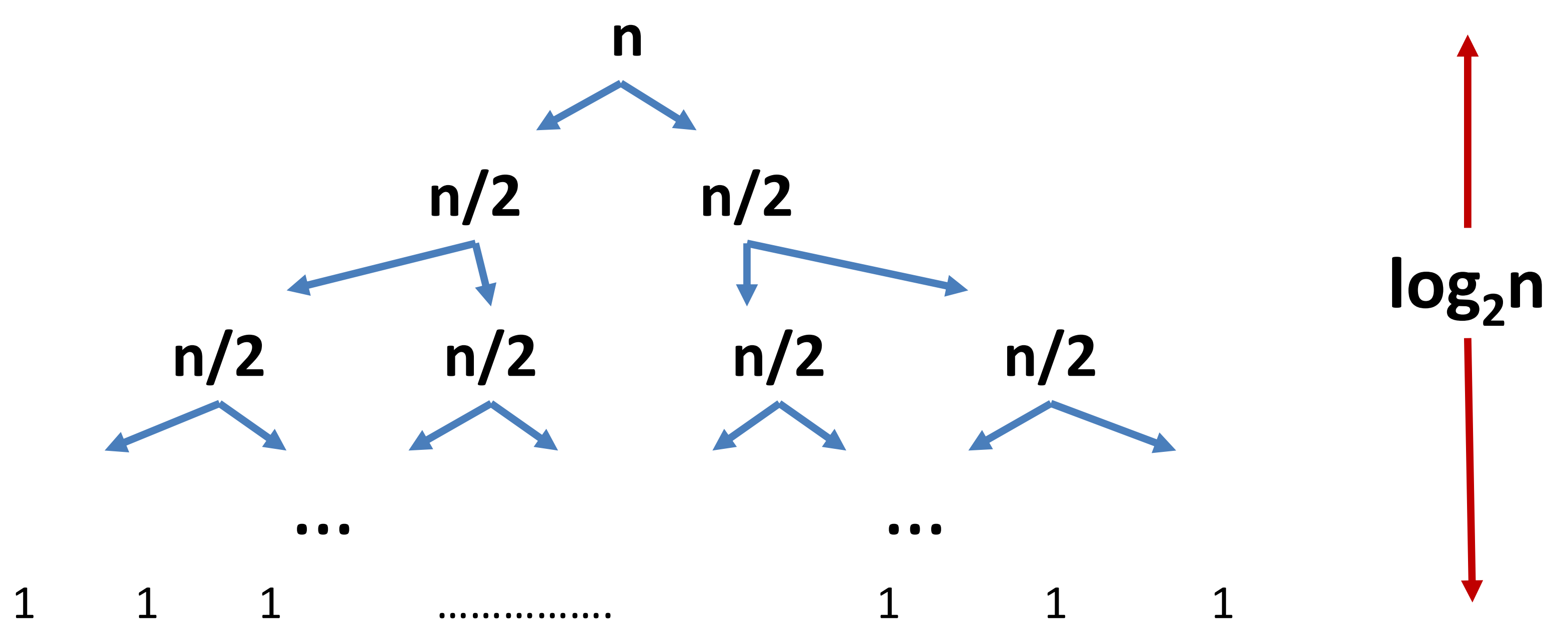
  swaps outside of the for loop   swaps inside of the for loop

  $= \mathbf{n^2/2 + n/2 - 1}$ ➔ $\mathbf{O(n^2)}$

- So, Quicksort is **$O(n^2)$** in worst case

Quicksort – Best and Avg. Analysis

- It takes **O(n*log$_2$n)** in the best and avg. case



Comparison of Sorting Algorithms

Sorting Algorithm	Best Case time	Worst Case time	Avg. Case time
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Merge Sort	$\Theta(nlgn)$	$\Theta(nlgn)$	$\Theta(nlgn)$
Quick Sort	$\Theta(nlgn)$	$\boldsymbol{\Theta(n^2)}$	$\Theta(nlgn)$

Internal Sort	Yes
External Sort	-
Stable Sort	No
In Place	Yes

Randomized Quick Sort

Comparison of Sorting Algorithms

Sorting Algorithm	**Best Case time**	**Worst Case time**	**Avg. Case time**
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Merge Sort	$\Theta(nlgn)$	$\Theta(nlgn)$	$\Theta(nlgn)$
Quick Sort	$\Theta(nlgn)$	$\Theta(n^2)$	$\Theta(nlgn)$
Randomized Quick Sort	$\Theta(nlgn)$	$\boldsymbol{\Theta}\mathbf{(nlgn)}$	$\Theta(nlgn)$

Internal Sort	Yes
External Sort	-
Stable Sort	No
In Place	Yes

Quicksort – Analysis

- Quicksort is slow when the array is sorted and we choose the first element as the pivot.

- Although the worst case behavior is not so good, its average case behavior is much better than its worst case.
  - So, Quicksort is one of best sorting algorithms using key comparisons.

Quick Sort Faster than Merge Sort

- Both quick and merge sort take O(n log n) in the average case
- But quick sort is faster in the average case:
  - The inner loop consists of an increment/decrement (by 1, which is fast), a test and a jump.
  - There is no extra juggling as in merge sort.

```
int i = left, j = right - 1;
for( ; ; )
{
    while( a[ ++i ] < pivot ) { }
    while( pivot < a[ --j ] ) { }
    if( i < j )
        swap( a[ i ], a[ j ] );
    else
        break;
}
```

inner loop

Summary

- Most of the sorting techniques we have discussed are $O(n^2)$
- We can do much better than this with somewhat more complicated sorting algorithms
- Within $O(n^2)$,
  - Bubble sort is very slow, and should probably never be used for anything
  - Selection sort is intermediate in speed
  - Insertion sort is usually faster than selection sort—in fact, for small arrays (say, 10 or 20 elements), insertion sort is faster than more complicated sorting algorithms
- Merge sort, if done in memory, is $O(n \log n)$. But not In Place
- Quick sort is good in general for average case inputs
- Merge sort is good for sorting data that doesn't fit in main memory