CS2040C Data Structures and Algorithms

Sorting

Outline

- Iterative sort algorithms (comparison based)
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
- Recursive sort algorithms (comparison based)
  - Mergesort
  - Quicksort
- Radix sort (non-comparison based)
- In-place sort, stable sort
- Comparison of sort algorithms

Note: we only consider sorting data in ascending order

Why study sorting?

- When an input is sorted, many problems become easy (e.g. searching, min, max, k^{th} smallest, ...)
- Sorting has a variety of interesting algorithmic solutions that embody many ideas:
  - iterative
  - recursive
  - divide-and-conquer
  - best/worst/average-case bounds
  - randomized algorithms

Sorting applications

- Uniqueness testing
- deleting duplicates
- prioritizing events
- frequency counting
- reconstructing the original order
- set intersection/union
- finding a target pair x, y such that x+y = z
- efficient searching

Algo #1: Selection Sort

Given an array of *n* items:

1) Find the largest item

2) Swap it with the item at the end of the array

3) Go to step 1 by excluding the largest item from the array

(refer to VisuAlgo for demo; note: algorithm in VisuAlgo finds the smallest and swaps it with the first element)

Selection Sort of 5 integers

29	10	14	**37**	13

29	10	14	13	37

13	10	**14**	29	37

13	10	14	29	37

10	13	14	29	37

Code of Selection Sort

```
void selectionSort(int a[], int len) {
  for (int i=len-1; i>=1; --i) {
     int index = i;
     for (int j=0; j<i; ++j) {
          if (a[j] >= a[index])
               index = j;
     }
     int temp = a[index];
     a[index] = a[i];
     a[i] = temp;
   }
}
```

29	10	14	37	13

Analysis of Selection sort

```
void selectionSort (int a[], int len)
{
   for (int i=len-1; i>=1; --i) {
     int index = i;
     for (int j=0; j<i; ++j) {
       if (a[j] > a[index])
         index = j;
     }
     SWAP( ... )
   }
}
```

c_1 and c_2 = cost of stmts in outer and inner block.

Number of times executed:

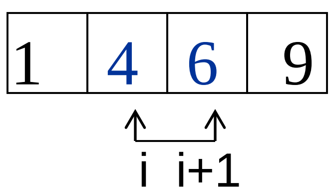
- n-1
- n-1
- $\left.\begin{array}{l}(n-1)+(n-2)+\ldots+1 \\ \\ \end{array}\right\} = n(n-1)/2$
- n-1

Total = $c_1(n-1)$
$+ c_2*n*(n-1)/2$
$= O(n^2)$

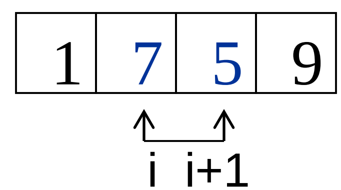
Algo #2: Bubble Sort

Idea:

- "bubble" the largest item to the end of the list in each iteration
- Examines items i and i+1 to see whether they need to be swapped.



// no need to swap

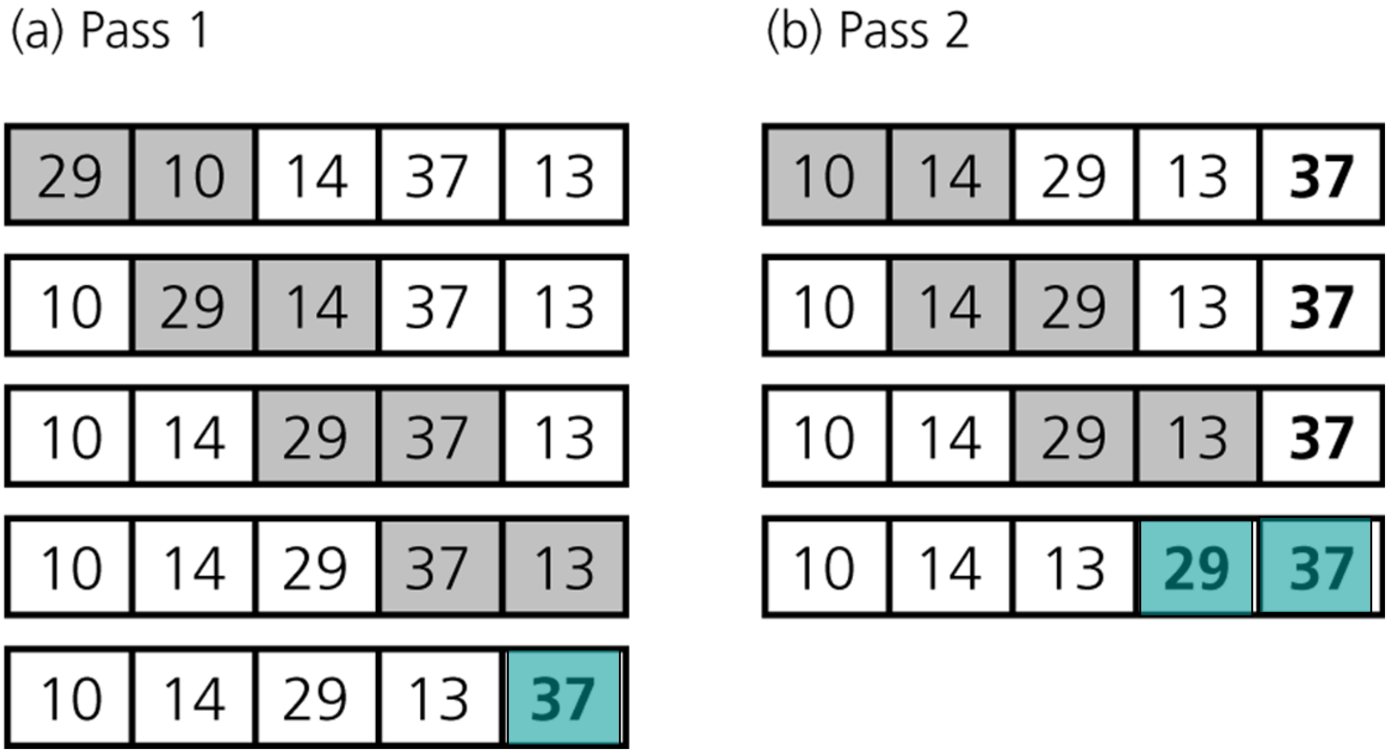


// out of order, need to swap

Code of Bubble Sort

```
void bubbleSort (int a[], int len){
  for (int i = 0; i < len ; ++i) {
      for (int j = 1; j < len -i; ++j) {
         // the largest item bubbles up
          if (a[j-1] > a[j]) {
               int temp = a[j-1];
               a[j-1] = a[j];
               a[j] = temp;
          }
      } // end for
  } // end outer for
}
```

The first two passes of a bubble sort of an array of five integers



Analysis of Bubble Sort

- 1 iteration of the inner loop (test and swap) requires time bounded by a constant c
- Two nested loops.
  - outer loop: exactly n iterations
  - inner loop:
    - when i=0, (n-1) iterations
    - when i=1, (n-2) iterations
    - …
    - when i=(n-1), 0 iterations
- Total number of iterations = 0+1+…+(n-1)

  = n(n-1)/2
- Total time is = c n(n-1)/2 = $O(n^2)$

Bubble Sort is inefficient

- Given a sorted input, bubble sort will still take $O(n^2)$ to sort.
- It does not make an effort to check whether the input has been sorted.
- Thus it can be improved as follows…

Code of Bubble Sort (Version 2)

```
void bubbleSort2 (int a[], int len) {
  for (int i = 0; i < len; ++i) {
     bool is_sorted = true;
     for (int j = 1; j < len -i; ++j) {
        if (a[j-1] > a[j]) {
            int temp = a[j-1] ;
            a[j-1] = a[j] ;
            a[j] = temp ;
            is_sorted = false;
         }
      } // end for
      if (is_sorted) return;
  } // end outer for
}
```

Can it be further improved?

Cocktail sort

23451

Analysis of Bubble Sort (Version 2)

- Worst-case
  - input is in descending order
  - running-time remains the same: $O(n^2)$
- Best-case
  - input is already in ascending order
  - the algorithm returns after a single outer-iteration
  - Running time: $O(n)$

Algo #3: Insertion Sort

Idea:

Arranging a hand of poker cards

- Start with one card in your hand
- Pick the next card and insert it into its proper sorted order
- Repeat previous step for all n cards

Example of Insertion Sort

n = 4	S1		S2		
Start	40	13	20	8	
i=1	13	40	20	8	
i=2	13	20	40	8	
i=3	8	13	20	40	

- S1 = Sorted so far
- S2 = Elements yet to be processed

Code of Insertion Sort

40 13 20 8

```
void insertionSort (int a[], int len){
   for (int i=1;i<len;++i) {
        // This is the next data to insert
        int next = a[i];
        // Scan backwards to find a place
        int j; // Why is j declared here?
        for (j=i-1; j>=0 && a[j]>next; --j)
            a[j+1] = a[j];    // right shift
        // Now insert the value after index j
        a[j+1] = next;
   } // outer for loop
}
```

Analysis of Insertion Sort

- Outer-loop executes (n – 1) times
- Number of times inner-loop executed depends on the input:
  - Best-case: the array is already sorted and (a[j] > next) is always false.
    - No shifting of data is necessary.
  - Worst-case: the array is reversely sorted and (a[j] > next) is always true.
    - Insertion always occur at the front.
- Therefore, the best-case time is O(n).
- And the worst-case time is $O(n^2)$.

Algo #4: Merge Sort

Suppose we only know how to merge two sorted sets of elements into one.

Given an unsorted set of n elements,

- merge each pair of elements into sets of 2.
- merge each pair of sets of 2 into sets of 4.
- Repeat previous step for sets of 4 …
- The final step merges 2 sets of n/2 elements to obtain a sorted set.

Divide-and-Conquer Method

Divide-and-conquer method solves problem in three steps:

- Divide Step: divide the large problem into smaller problems.
- Recursively solve the smaller problems
- Conquer Step: combine the results of the smaller problems to produce the result of the larger problem.

MergeSort Idea

- MergeSort is a divide-and-conquer sorting algorithm
- Divide Step: Divide the array into two (equal) halves
- Recursively sort the two halves
- Conquer Step: Merge the two halves to form a sorted array

Example of MergeSort

7	2	6	3	8	4	5

Divide into two halves

7	2	6	3

8	4	5

Recursively sort the halves

2	3	6	7

4	5	8

Merge them

2	3	4	5	6	7	8

Code of MergeSort

Base case: When (i>=j)
it is an array of size 1

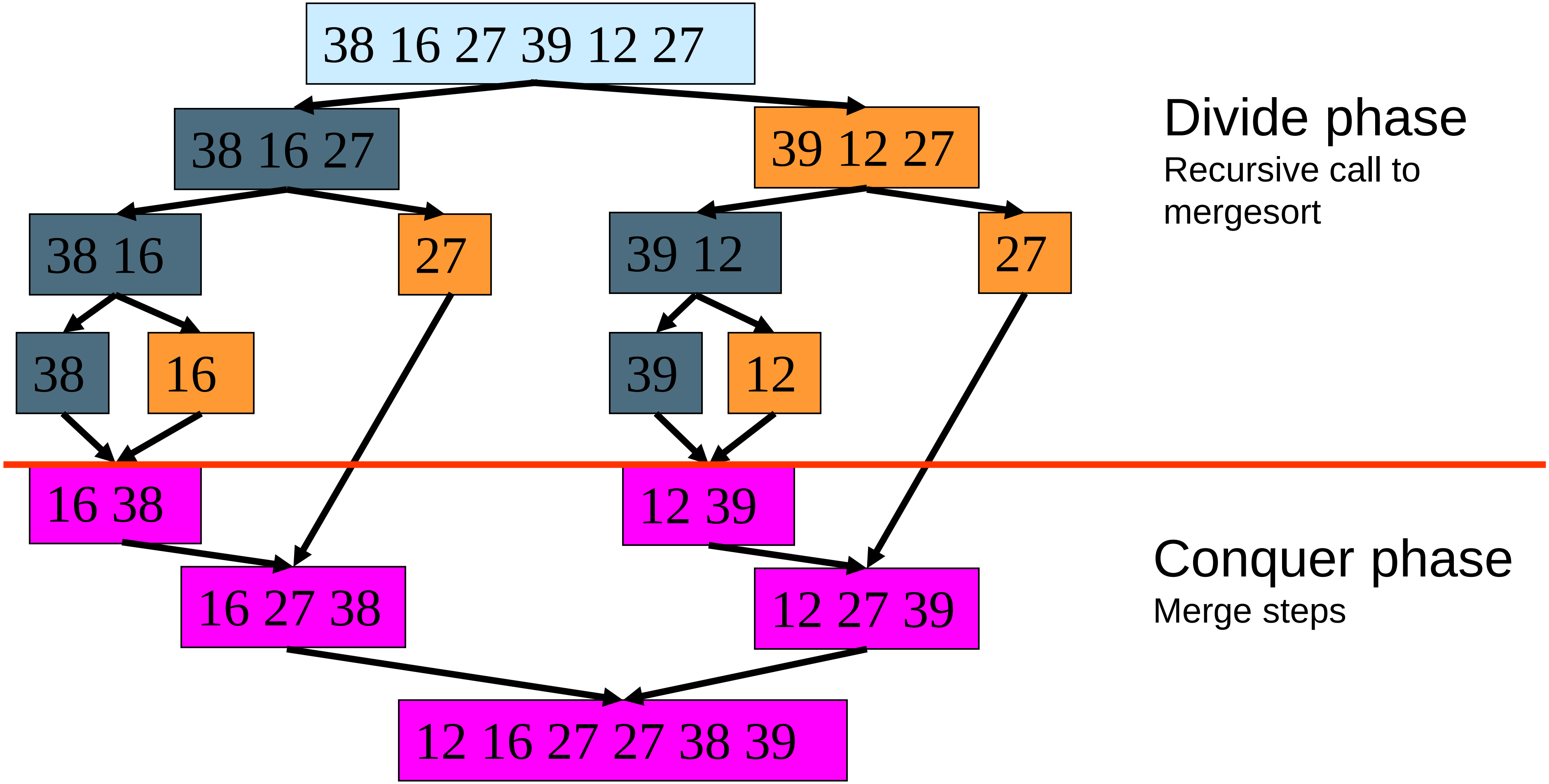
```
void mergeSort (int a[], int i, int j){

    if (i < j) {
        int mid = (i+j)/2;      // divide
        mergeSort(a,i,mid);     // recursion
        mergeSort(a,mid+1,j);  //
        merge(a, i, mid, j);   //conquer
    }
}
```

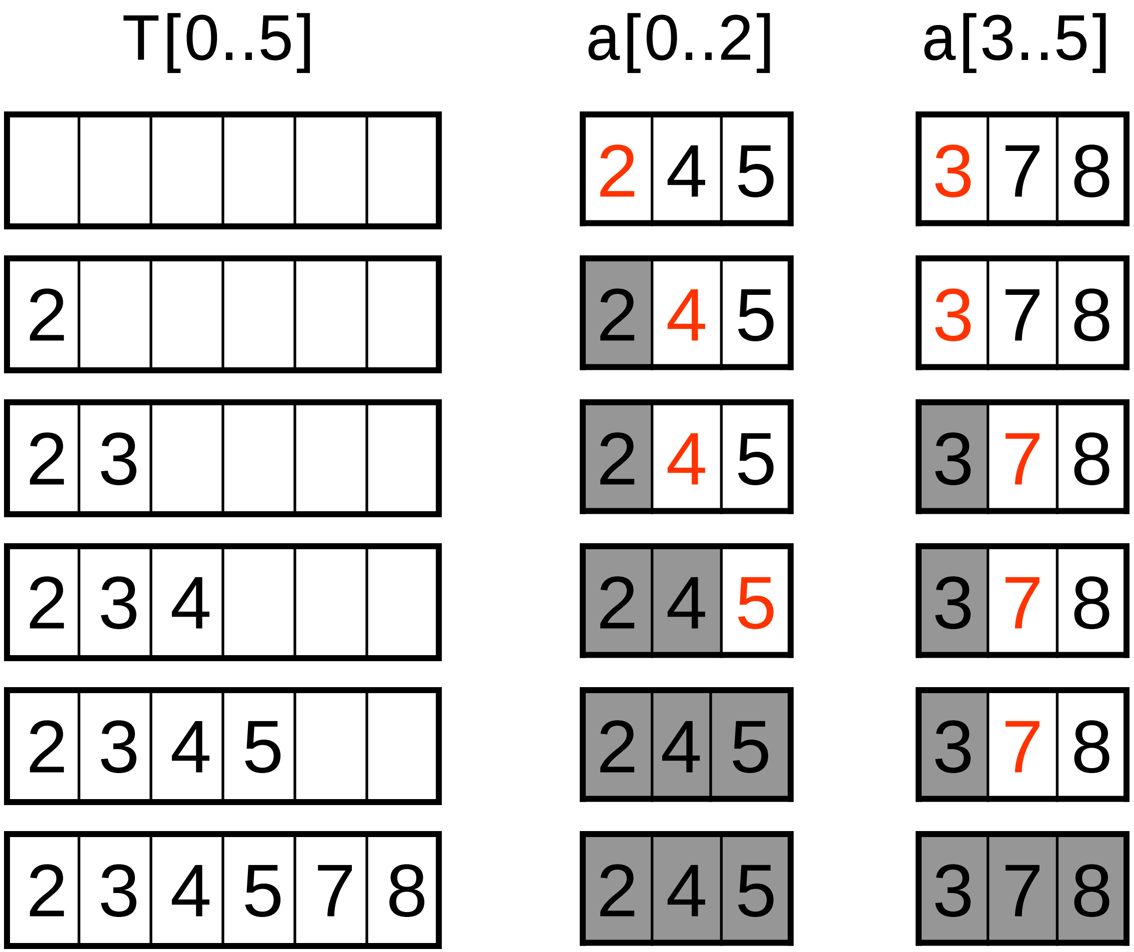
merge() must store
the result back in a[i..j].

MergeSort of an array of six integers

```
mergeSort(a,i,mid);
mergeSort(a,mid+1,j);
merge(a[i..mid], a[mid+1..j]);
```



How to merge two sorted subarrays?



Merge Algorithm

```
void merge(int a[], int i, int mid, int j) {
// Merges a[i..mid] a[mid+1..j] into a[i..j]
   int n = j-i+1;
   int* b = new int[n]; //temp. storage
   int left=i, right=mid+1, ib=0;
   while (left<=mid && right<=j) {
       if (a[left] <= a[right])
           b[ib++] = a[left++];
       else
           b[ib++] = a[right++];
   }
```

Merge Algorithm (cont'd)

```
    // Copy the remaining elements into b
    while (left<=mid) b[ib++] = a[left++];
    while (right<=j) b[ib++] = a[right++];
    // Copy the result back into array a
    for (int k=0;k<n;++k)
      a[i+k] = b[k];
    delete [] b;
}
```

Time analysis for Merge

In mergeSort, the bulk of work is done in the merge step.

merge(a, i, mid, j)

Total items = k = (j – i + 1)

- Number of comparisons ≤ k-1
- Number of moves from original array to temporary array = k
- Number of moves from temporary array back to original array = k

In total, no. of operations ≤ 3k-1 = O(k)

How many times is merge() called?

```
void mergeSort (int a[], int i, int j){
    if (i < j)
        int mid = (i+j)/2;// divide
        mergeSort(a,i,mid);// recursion
        mergeSort(a,mid+1,j);
        merge(a, i, mid, j);
                //conquer
    }
}
```

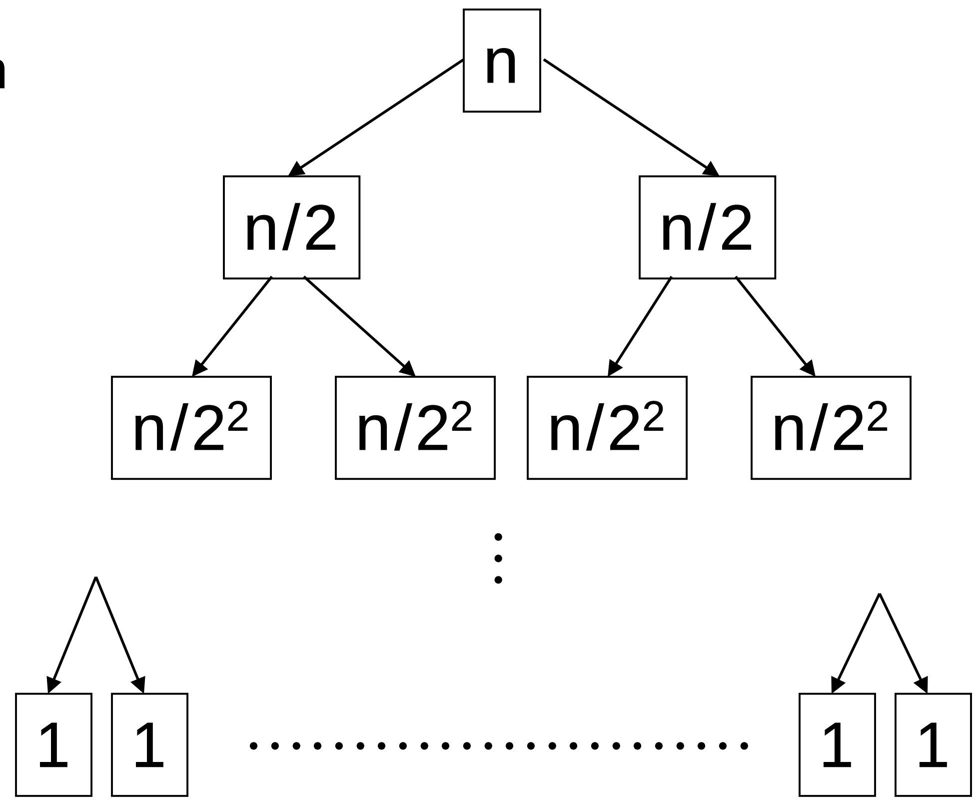
Time analysis for MergeSort

Level 0: Mergesort n items

Level 1: Mergesort n/2 items

Level 2: Mergesort $n/2^2$ items

Level (log n): Mergesort 1 item



Level 0: 1 call to Mergesort

Level 1: 2 calls to Mergesort

Level 2: 2^2 calls to Mergesort

Level (log n): $2^{\log n}$(= n) calls to Mergesort

$$n/(2^k) = 1 \quad \Rightarrow \quad n = 2^k \quad \Rightarrow \quad k = \lg n$$

Time analysis for MergeSort

- Level 0: 0 call to merge
- Level 1: 1 call to merge with n/2 items each, O(1 x 2 x n/2) = O(n) time
- Level 2: 2 calls to merge with $n/2^2$ items each, O(2 x 2 x $n/2^2$) = O(n) time
- Level 3: 2^2 calls to merge with $n/2^3$ items each, O(2^2 x 2 x $n/2^3$) = O(n) time
- …

- Level (lg n): $2^{\lg n-1}$(= n/2) calls to merge with $n/2^{\lg n}$ (= 1) item each, O(n) time
- In total, running time = O(n lg n)
- Optimal comparison based sort method

Drawbacks of MergeSort

1. Not as easy to implement
2. Requires additional storage to copy the merged set

Algo #5: Quicksort

Quicksort is a divide-and-conquer algorithm

- Divide Step: Choose an item *p* and partition the items of a[i..j] into two parts such that
  - the items in one part are smaller than *p* while
  - those in the other part are greater than or equal to *p*
- Recursively sort the two parts
- Conquer Step: Do nothing!

Note:

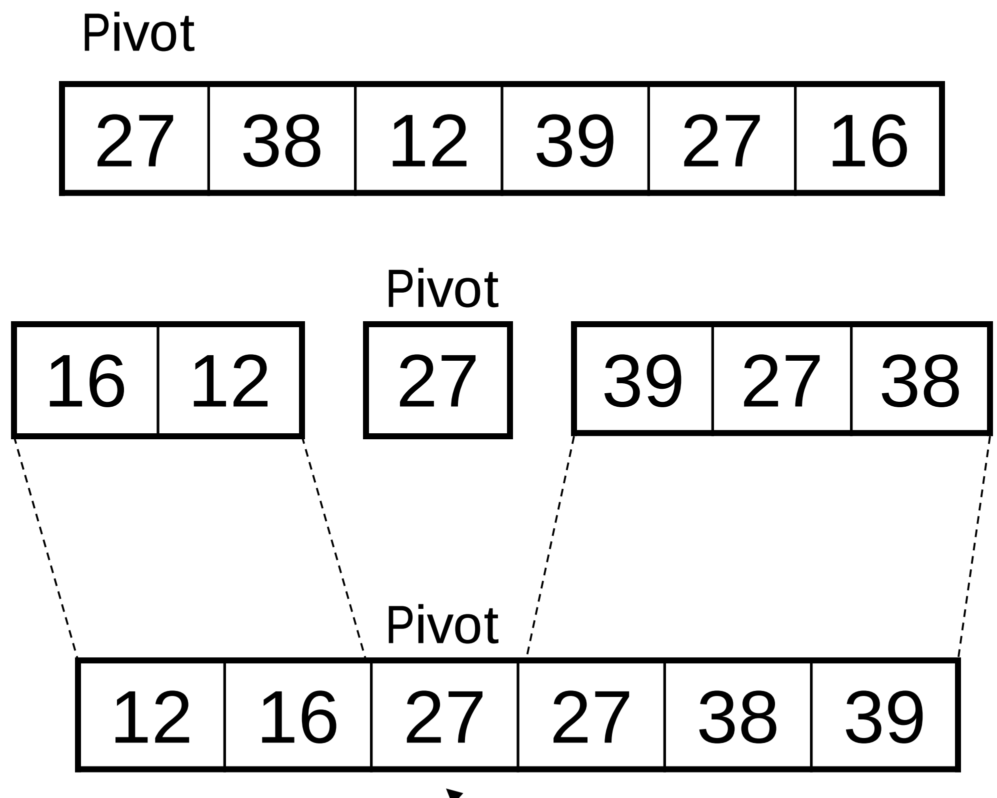
p is called a pivot. It can be the first, the last, the middle, or any item chosen randomly

Mergesort spends most of the time in conquer step but very little time in divide step

Quicksort Example

Partition a[] about the pivot 27

Recursively sort the two parts



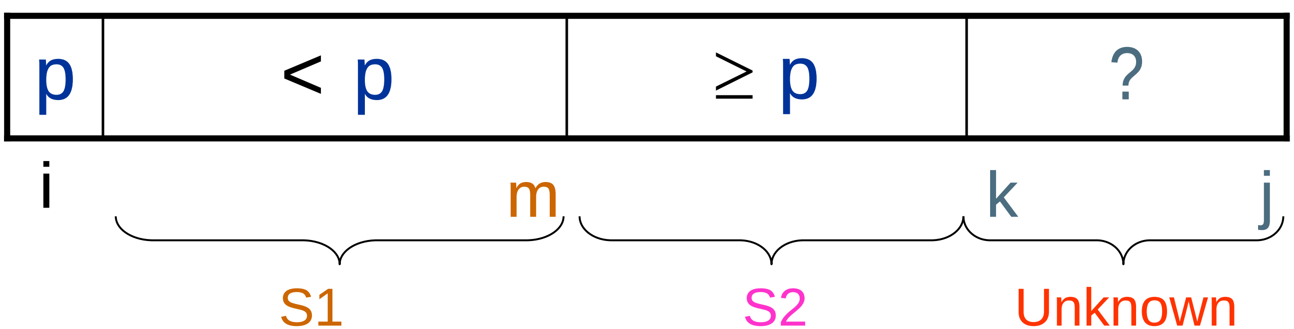
Note that after the partition,
the pivot is moved to its final position!

Code of Quicksort

```
void Quicksort (int a[], int i, int j){
   if (i < j) {
     int pivotIdx = partition(a, i, j);
     //pivot untouched
     Quicksort(a, i, pivotIdx - 1);
     Quicksort(a, pivotIdx + 1, j);
   }
}
```

Partition algorithm idea

- To partition a[i..j], we choose a[i] as the pivot p
- The remaining items (i.e., a[i+1..j]) are divided into three regions:
  - S1 = a[i+1..m] : items < p
  - S2 = a[m+1..k-1] : items ≥ p
  - Unknown = a[k..j] : items to be assigned to S1 or S2

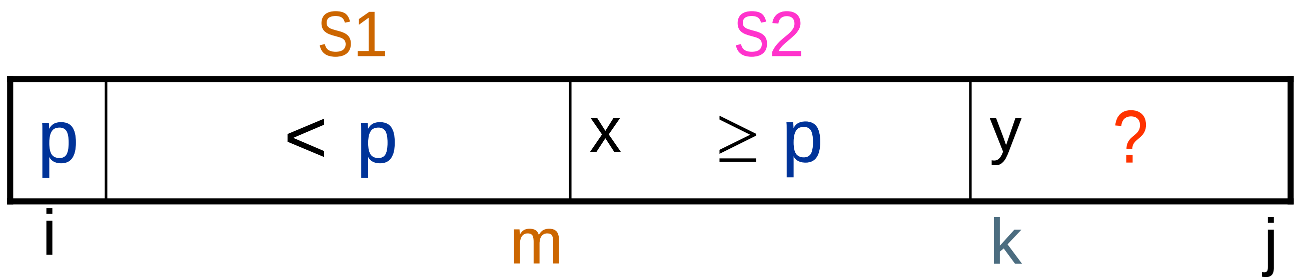


Partition algorithm idea (cont'd)

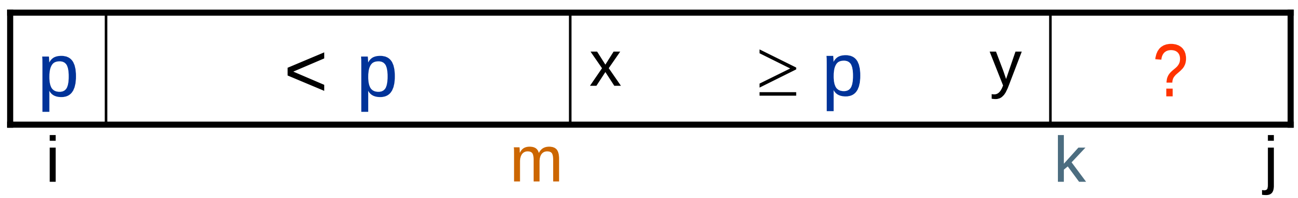
- Initially, regions S1 and S2 are empty. All items excluding p are in the unknown region
- Then, for each item a[k] in the unknown region, Compare a[k] with p:
  - If a[k] >= p, put it into S2
  - Otherwise, put a[k] into S1

Partition algorithm idea (case 1)

If a[k] = y $\geq$ p,



Increment k



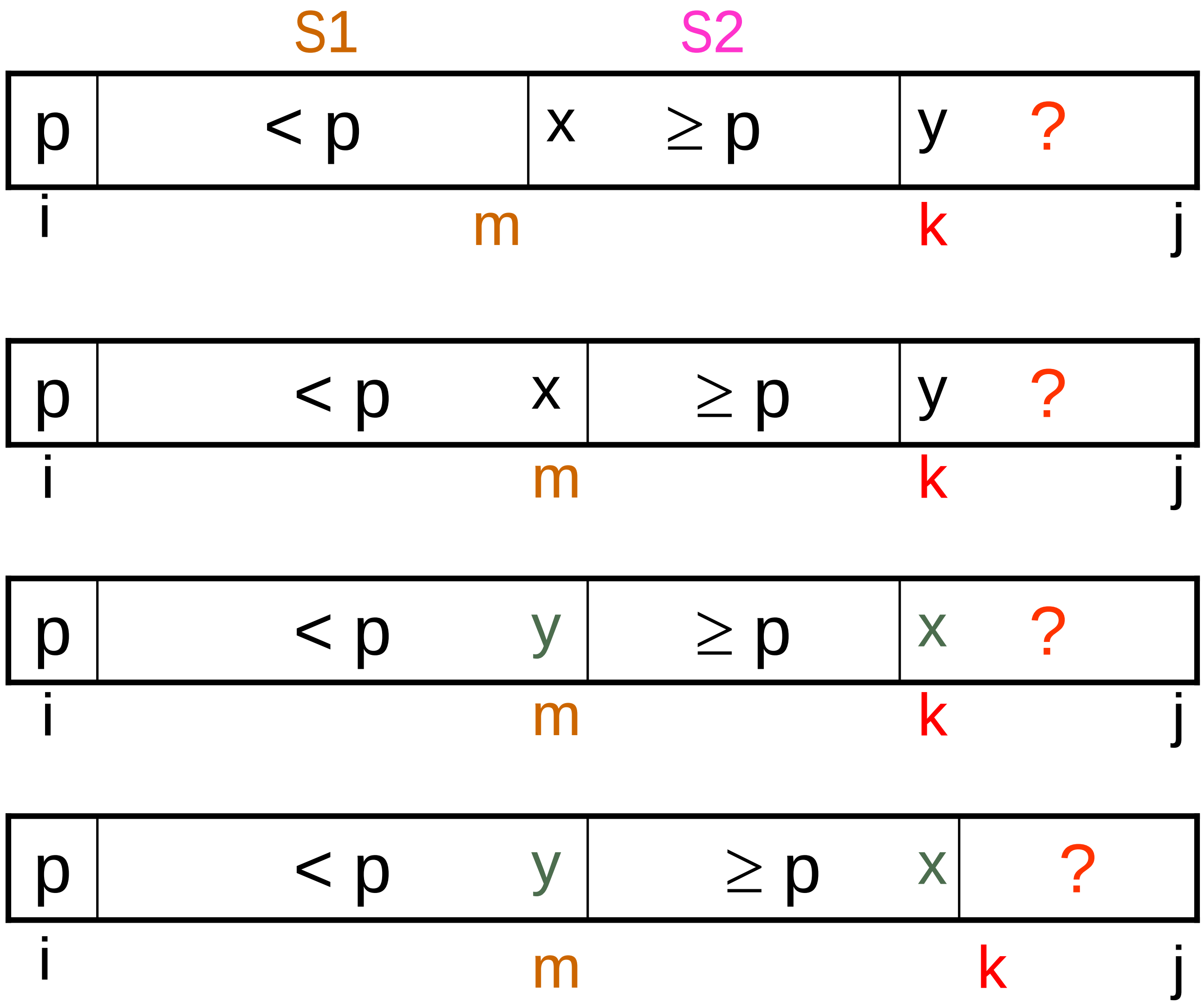
Partition algorithm idea (case 2)

If a[k] = y < p,

Increment m

Swap x and y

Increment k



Partition algorithm

```
int partition(int a[], int i, int j) {
    int p = a[i];              // p is the pivot
    int m = i;                           // Initially S1 and S2 are empty
    for (int k = i+1; k<=j; ++k) { // process unknown region
        if (a[k] < p) {                  // put a[k] to S1
            ++m;
            swap(a,k,m);
        } else {                         // put a[k] to S2! Do nothing!
        }
    }
    swap (a,i,m);                        // put the pivot at the right place
    return m;
}
```

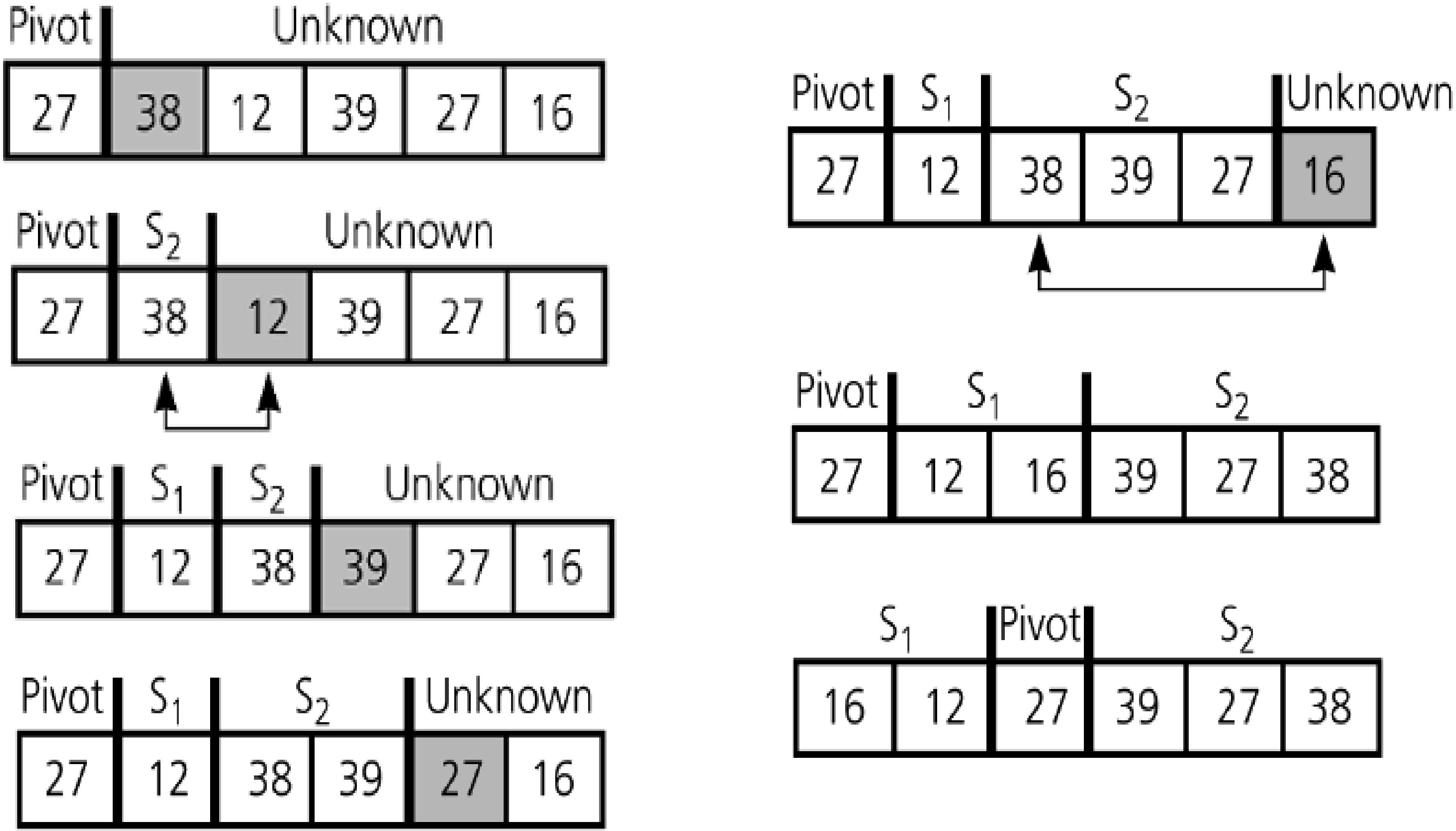
Can the else part be removed?

Complexity of partition algorithm

As there is only one *for* loop and the size of the array is n=j-i+1, so

the complexity is O(n)

Partition algorithm by example



Worst Case for Quicksort

When a[0..n-1] is in increasing order:

10	13	14	29	37

p=a[i]

S_2 = a[m+1,j]

S_1 =a[i+1,m] is empty
when m = i

What is the index returned by partition()?

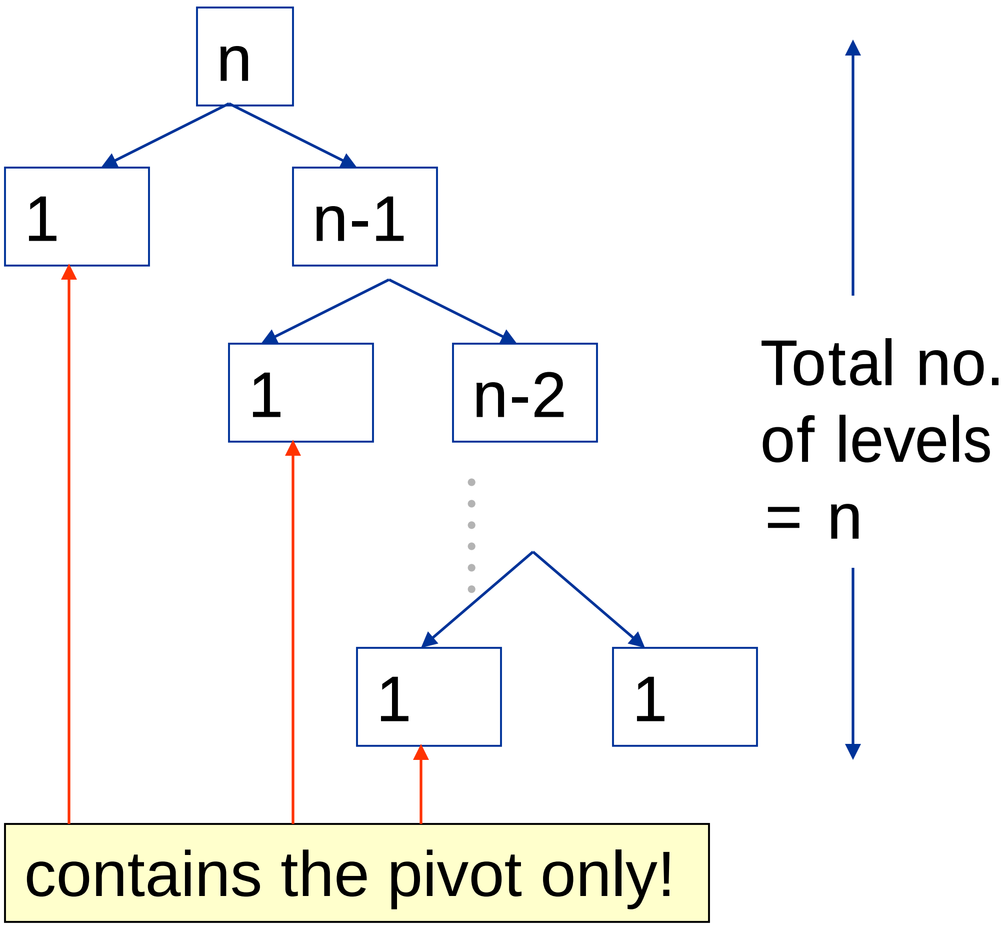
swap(a,i,m) will swap the pivot with itself!

The left partition is empty and

the right partition is the rest excluding the pivot

When a[0..n-1] is in decreasing order?

Worst Case for Quicksort (cont'd)



As partition takes O(n) time, the algorithm in the worst case takes time n+(n-1)+...+1 = O(n^2)

Best/average case for Quicksort

- Best case occurs when partition always splits the array into two equal halves
  - Depth of recursion is lg n
  - Time complexity is O(n lg n)
- In practice, worst case is rare, and on the average we get some good splits and some bad ones.
  - Average time is O(n lg n)

Lower bound

All comparison based sorting algorithms have lower bound n log n, i.e.,

Ω**(n log n)**

Therefore, any comparison based sorting algorithm with worst case complexity O(n log n) is considered optimal.

Radix Sort

- Treats each data to be sorted as a character string of w digits
- Integers with less than w digits are padded with leading zeros
- It does not use comparison, i.e., no comparison between the data is needed
- For each iteration, starting with the least significant (rightmost) digit to the most significant digit, we pass through the data and put them into 10 groups (queues) (one for each digit [0..9]), according to the corresponding digit in each data
- Then we re-concatenate the groups again for subsequent iteration

Radix Sort of Eight Integers

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

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

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

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

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

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

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

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

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

Pseudocode of Radix sort

```
create 10 buckets (queues) for each digit (0 to 9)
for each digit placing
    for each element in list
        move element into respective bucket
    for each bucket, starting from smallest digit
        while bucket is non-empty
            restore element to list
```

Complexity of Radix Sort

Complexity is $O(n)$, or $O(dn)$ where d is the maximum number of characters in the data.

In-place Sort

- A sort algorithm is said to be an "in-place" sort if it requires only a constant amount (ie., O(1)) of extra space during the sorting process.
  - Mergesort is not in-place, why?

Stable Sort

A sorting algorithm is "stable" if it does not reorder elements that are equal.

Example:

Student names have been sorted into alphabetical order. If it is sorted again according to tutorial group number, a stable sort algorithm will make all within the same group to appear in alphabetical order.

Is the sorting algorithm used in excel stable?

Selection sort and Quicksort (without modifications) are not stable - why?

Stable Sort counter examples

Example:
Selection sort:
1285 5a 4746 602 5b (8356)
1285 5a 5b 602 (4746 8356)
602 5a 5b (1285 4746 8356)
5b 5a (602 1285 4746 8356)

Quicksort:
1285 5a 150 4746 602 5b 8356 (pivot in bold)
1285 (5a 150 602 5b) (4746 8356)
5b 5a 150 602 **1285** 4746 8356

Summary of Sorting Algorithms

	Worst Case	Best Case	In-place?	Stable?
Selection Sort	$O(n^2)$	$O(n^2)$	Yes	No
Insertion Sort	$O(n^2)$	$O(n)$	Yes	Yes
Bubble Sort	$O(n^2)$	$O(n^2)$	Yes	Yes
Bubble Sort 2	$O(n^2)$	$O(n)$	Yes	Yes
Mergesort	$O(n \lg n)$	$O(n \lg n)$	No	Yes
Radix sort	$O(dn)$	$O(dn)$	No	yes
Quicksort	$O(n^2)$	$O(n \lg n)$	Yes	No

STL sort

The Standard Template Library (STL) provides several sort functions in library header <algorithm>

```
#include <algorithm>
void sort( iterator start, iterator end );
```

guaranteed performance O(n log n)

STL sort example – (1a)

```
vector<int> v;
v.push_back( 23 );
v.push_back( -1 );
v.push_back( 9999 );
v.push_back( 0 );
v.push_back( 4 );

sort( v.begin(), v.end() );          // using operator< for int

cout << "After sorting: ";
for( unsigned int i = 0; i < v.size(); i++ ) {
    cout << v[i] << endl ;
}
```

STL sort example – (1b)

To sort it into descending order:

sort(v.begin(), v.end(), greater<int>());

greater<int>() is called a comparison functor.

Other STL comparison functors are:

- equal_to,
- not_equal_to,
- less,
- greater_equal,
- less_equal

STL sort example – (2a)

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

class Circle {
public:
    Circle(int i): r(i) {}
    int getr() { return r;}
private:
    int r;
};
bool smaller ( Circle x,  Circle y) { return x.getr() < y.getr(); }
bool bigger ( Circle x,  Circle y) { return x.getr() > y.getr(); }
```

STL sort example – (2b)

```
int main(){
    Circle c1(1), c2(2), c3(3);
    vector<Circle> v;
    v.push_back(c2);
    v.push_back(c1);
    v.push_back(c3);
    vector<Circle>::iterator i = v.begin();
    vector<Circle>::iterator e = v.end();
    sort (i, e, smaller);  // use bigger for descending
    cout << " In ascending order:" << endl;
    for (; i != e; ++i)  cout << i ->getr() << endl;
    return 0;
}
```

Other sort related STL functions

Sort

1. sort
2. stable_sort
3. partial_sort
4. partial_sort_copy
5. is_sorted
6. nth_element

- merge
- inplace_merge