

Sorting

Bringing Order to the World

Lecture Outline

- Iterative sorting algorithms (comparison based)
 - Selection Sort
 - Bubble Sort
 - Insertion Sort
- Recursive sorting algorithms (comparison based)
 - Merge Sort
 - Quick Sort

- Radix sort (non-comparison based)
- Properties of Sorting
- In-place sort, stable sort
- Comparison of sorting 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
- Comparison vs non-comparison based
- Iterative
- Recursive
- Divide-and-conquer
- Best/worst/average-case bounds
- Randomized algorithms

Applications of Sorting

- 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

Selection Sort

Selection Sort: Idea

- Given an array of n items
 1. Find the largest item x , in the range of $[0 \dots n-1]$
 2. Swap x with the $(n-1)^{\text{th}}$ item
 3. Reduce n by 1 and go to Step 1

29	10	14	37	13
----	----	----	----	----

37 is the largest, swap it with the last element, i.e. 13.

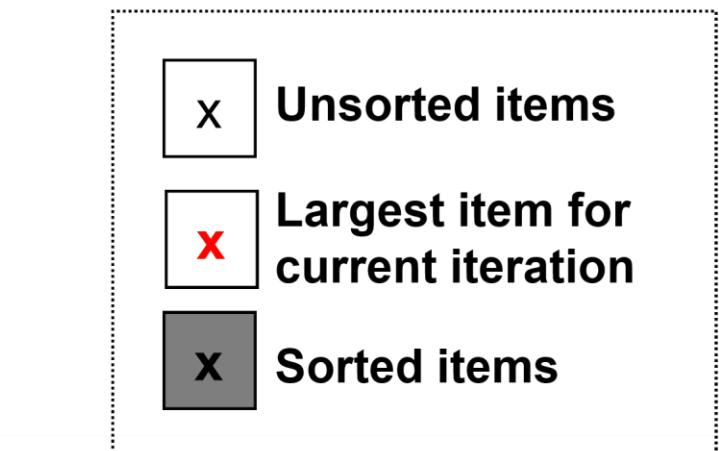
Q: How to find the largest?

29	10	14	13	37
----	----	----	----	----

13	10	14	29	37
----	----	----	----	----

--	--	--	--	--

10	13	14	29	37
----	----	----	----	----



Sorted!

We can also find the smallest and put it the front instead

<http://visualgo.net/sorting?create=29,10,14,37,13&mode=Selection>

Selection Sort: Implementation

```
void selectionSort(int a[], int n) {  
    for (int i = n-1; i >= 1; i--) {  
        int maxIdx = i;  
        for (int j = 0; j < i; j++)  
            if (a[j] >= a[maxIdx])  
                maxIdx = j;  
        // swap routine is in STL <algorithm>  
        swap(a[i], a[maxIdx]);  
    }  
}
```

Step 1:
Search for
maximum
element

Step 2:
Swap
maximum
element
with the last
item i

Selection Sort: Analysis

```
void selectionSort(int a[], int n) {  
    for (int i = n-1; i >= 1; i--) {  
        int maxIdx = i;  
        for (int j = 0; j < i; j++)  
            if (a[j] >= a[maxIdx])  
                maxIdx = j;  
        // swap routine is in STL <algorithm>  
        swap(a[i], a[maxIdx]);  
    }  
}
```

Number of times
executed

$n-1$

" "

$$\begin{aligned} & (n-1)+(n-2)+\dots+1 \\ & = n(n-1)/2 \end{aligned}$$

$n-1$

.....

$$= c_1(n-1) +$$

$$c_2 * n * (n-1)/2$$

$$= O(n^2)$$

- c_1 and c_2 are cost of statements in outer and inner blocks

Bubble Sort

Bubble Sort: Idea

- Given an array of n items
 1. Compare pair of adjacent items
 2. Swap if the items are out of order

3. Repeat until the end of array

- The largest item will be at the last position

4. Reduce n by 1 and go to Step 1

- Analogy

- Large item is like “bubble” that floats to the end of the array



At the end of Pass 1, the last item 37 is at the last position

At the end of Pass 2, the second largest item 29 is at the second last position.



Sorted Item



**Pair of items
under comparison**

[CS1020E AY1617S1 Lecture 10]

12

[CS1020E AY1617S1 Lecture 10]

11

Bubble Sort: Illustration

(a) Pass 1

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

```
void bubbleSort(int a[], int n) {  
    for (int i = n-1; i >= 1; i--) {  
        for (int j = 0; j < i; j++) {  
            if (a[j] > a[j+1])  
                swap(a[j], a[j+1]);  
    }  
}
```

Step 1:
Compare adjacent pairs of numbers

Step 2:
Swap if the

of order

29	10	14	37	13
----	----	----	----	----

<http://visualgo.net/sorting?create=29,10,14,37,13&mode=Bubble>

Bubble Sort: Analysis

- 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 = $0+1+\dots+(n-1) = n(n-1)/2$
- Total time = $c n(n-1)/2 = O(n^2)$

Bubble Sort: Early Termination

- Bubble Sort is inefficient with a $O(n^2)$ time complexity
- However, it has an interesting property
- Given the following array, how many times will the inner loop swap a pair of item?

3	6	11	25	39
---	---	----	----	----

■ Idea

- If we go through the inner loop with no swapping → the array is sorted → can stop early!

Bubble Sort v2.0: Implementation

```
void bubbleSort2(int a[], int n) {  
    for (int i = n-1; i >= 1; i--) {  
        bool is_sorted = true;  
        for (int j = 1; j <= i; j++) {  
            if (a[j-1] > a[j]) {  
                swap(a[j], a[j-1]);  
                is_sorted = false;  
            }  
        } // end of inner loop  
        if (is_sorted) return;  
    }  
}
```

Assume the array
is sorted before

Any swapping will
invalidate the
assumption

If the flag
remains **true**
after the inner
loop → sorted!

Bubble Sort v2.0: Analysis

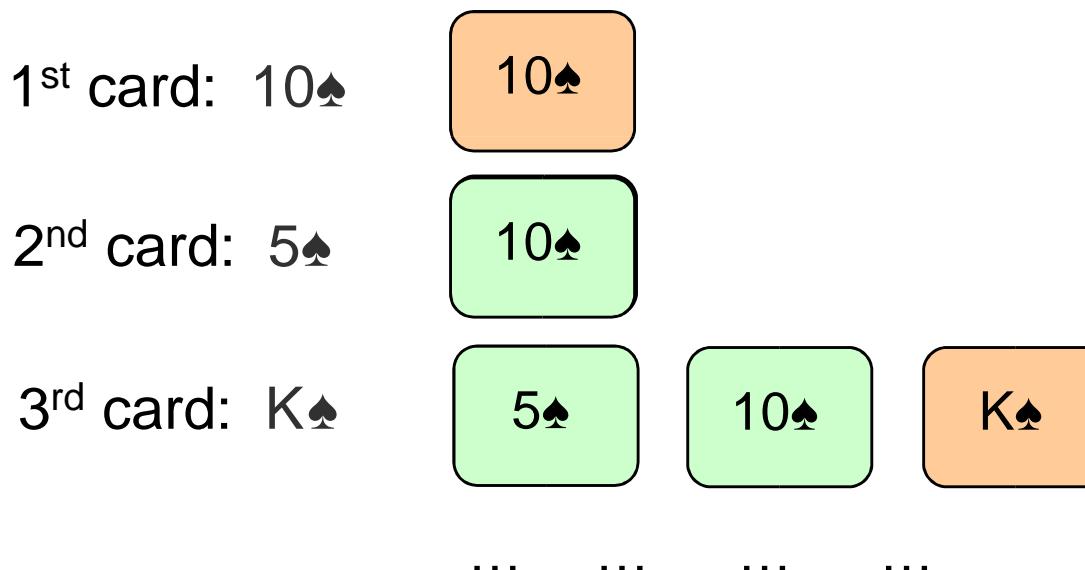
- Worst-case
- Input is in descending order
- Running time remains the same: $O(n^2)$

- Best-case
- Input is already in ascending order
- The algorithm returns after a single outer iteration
- Running time: $O(n)$

Insertion Sort

Insertion Sort: Idea

- Similar to how most people arrange 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 cards

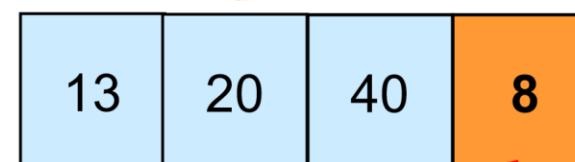
Start



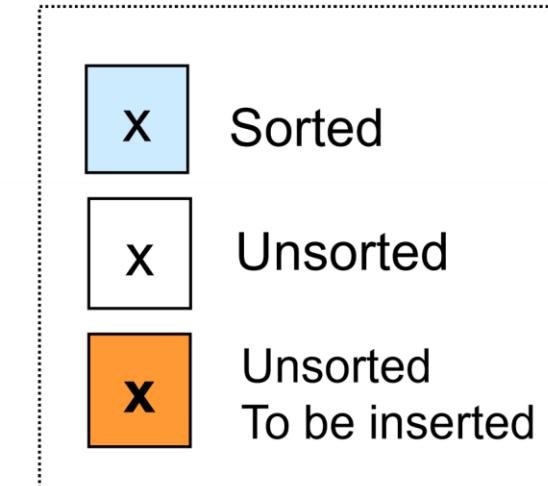
Iteration 1



Iteration 2



Iteration 3



<http://visualgo.net/sorting?create=40,13,20,8&mode=Insertion>

```
void insertionSort(int a[], int n) {  
    for (int i = 1; i < n; i++) {  
        - - -  
        int j;  
  
        for (j = i-1; j >= 0 && a[j] > next; j--)  
            a[j+1] = a[j];  
  
        a[j+1] = next;  
    }  
}
```

next is the item to be inserted

Shift sorted items to make place for **next**

Insert **next** to the correct location

29	10	14	37	13
----	----	----	----	----

<http://visualgo.net/sorting?create=29,10,14,37,13&mode=Insertion>

Insertion Sort: Analysis

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

- And the worst-case time is $O(n^2)$

Merge Sort

Merge Sort: Idea

- Suppose we only know how to merge two sorted sets of elements into one
- Merge {1, 5, 9} with {2, 11} → {1, 2, 5, 9, 11}

- Question
- Where do we get the two sorted sets in the first place?
- Idea (use merge to sort n items)
- 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 ...
- Final step: merge 2 sets of $n/2$ elements to obtain a fully sorted set

Divide-and-Conquer Method

- A powerful problem solving technique

- Divide-and-conquer method solves problem in the following 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

Divide and Conquer: Merge Sort

- Merge Sort 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

Merge Sort: Illustration

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
---	---	---	---	---	---	---

■ Question

- How should we sort the halves in the 2nd step?

7	2	6	3	8	4	5
---	---	---	---	---	---	---

```
int mid = (low+high) / 2;  
  
mergeSort(a, low , mid ) ;  
mergeSort(a, mid+1, high) ;  
  
merge(a, low, mid, high) ;  
}  
}
```

Merge sort on
a[low...high]

Divide **a[]** into two
halves and **recursively**
sort them

Conquer: merge the
two sorted halves

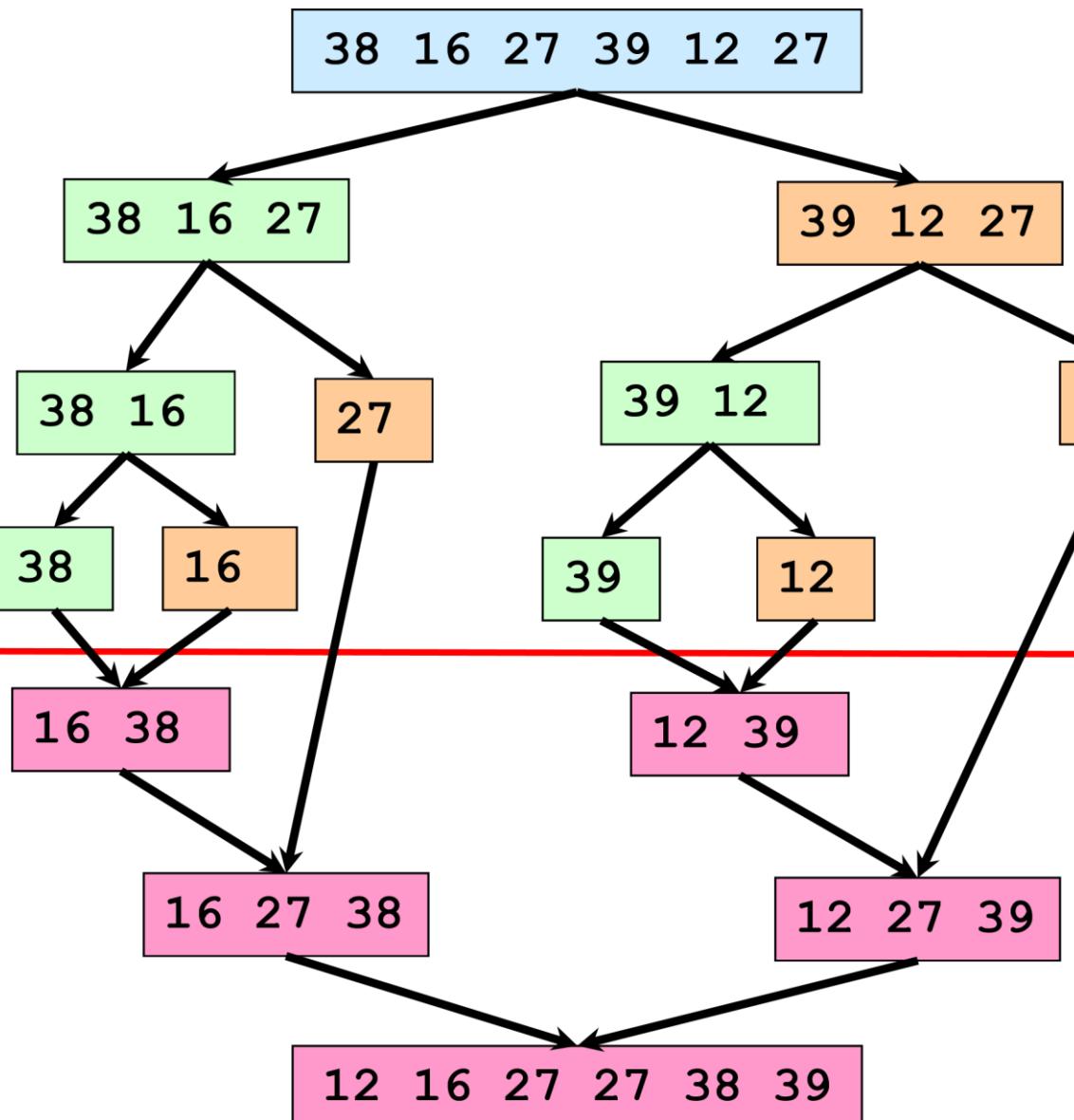
Function to merge

a[mid+1...high] into
a[low...high]

■ Note

- **mergeSort()** is a recursive function
- **low >= high** is the base case, i.e. there is 0 or 1 item

```
void mergeSort(int a[], int low, int high) {  
    if (low < high) {
```



```

mergeSort(a[low...mid])
mergeSort(a[mid+1...high])
merge(a[low..mid],
      a[mid+1..high])
    
```

Divide Phase
Recursive call to
`mergeSort()`

Merge steps

<http://visualgo.net/sorting?create=38,16,27,39,12,27&mode=Merge>

$a[0..2]$

2	4	5
---	---	---

$a[3..5]$

3	7	8
---	---	---

$b[0..5]$

--	--	--	--	--	--

2					
---	--	--	--	--	--

2	3				
---	---	--	--	--	--

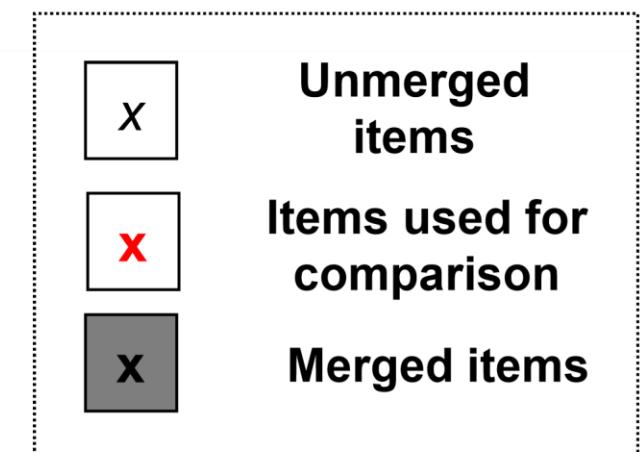
2	3	4			
---	---	---	--	--	--

2	3	4	5		
---	---	---	---	--	--

2	3	4	5	7	8
---	---	---	---	---	---

Two sorted halves to be merged

Merged result in a temporary array



```
int n = high-low+1;  
int* b = new int[n];  
int left=low, right=mid+1, bIdx=0;
```

b is a temporary array to store result

```
while (left <= mid && right <= high) {  
    if (a[left] <= a[right])  
        b[bIdx++] = a[left++];  
    else  
        b[bIdx++] = a[right++];  
}
```

Normal Merging
halves have unmerged items

// continue on next slide

PS: C++ STL <algorithm> has merge subroutine too

```
void merge(int a[], int low, int mid, int high) {
```

```
// continued from previous slide
```

```
while (left <= mid) b[bIdx++] = a[left++];
```

```
while (right <= high) b[bIdx++] = a[right++];
```

```
for (int k = 0; k < n; k++) {  
    a[low+k] = b[k];
```

Merged result
are copied
back into **a[]**

Remaining
items are
copied into
b[]

```
delete [] b;
```

```
}
```

Remember to free
allocated memory

■ Question

- Why do we need a temporary array **b[]**?

Merge Sort: Analysis

- In `mergeSort()`, the bulk of work is done in the merge step
- For `merge(a, low, mid, high)`
- Let total items = $k = (\text{high} - \text{low} + 1)$
- Number of comparisons $\leq k - 1$
- Number of moves from original array to temporary array = k
- Number of moves from temporary array back to original array = k
- In total, number of operations $\leq 3k - 1 = O(k)$
- The important question is
- How many times is `merge()` called?

Merge Sort: Analysis

Level 0:

mergeSort n items

Level 1:

mergeSort $n/2$ items

Level 2:

mergeSort $n/2^2$ items

Level ($\lg n$):

mergeSort 1 item

Level 0:

1 call to mergeSort

Level 1:

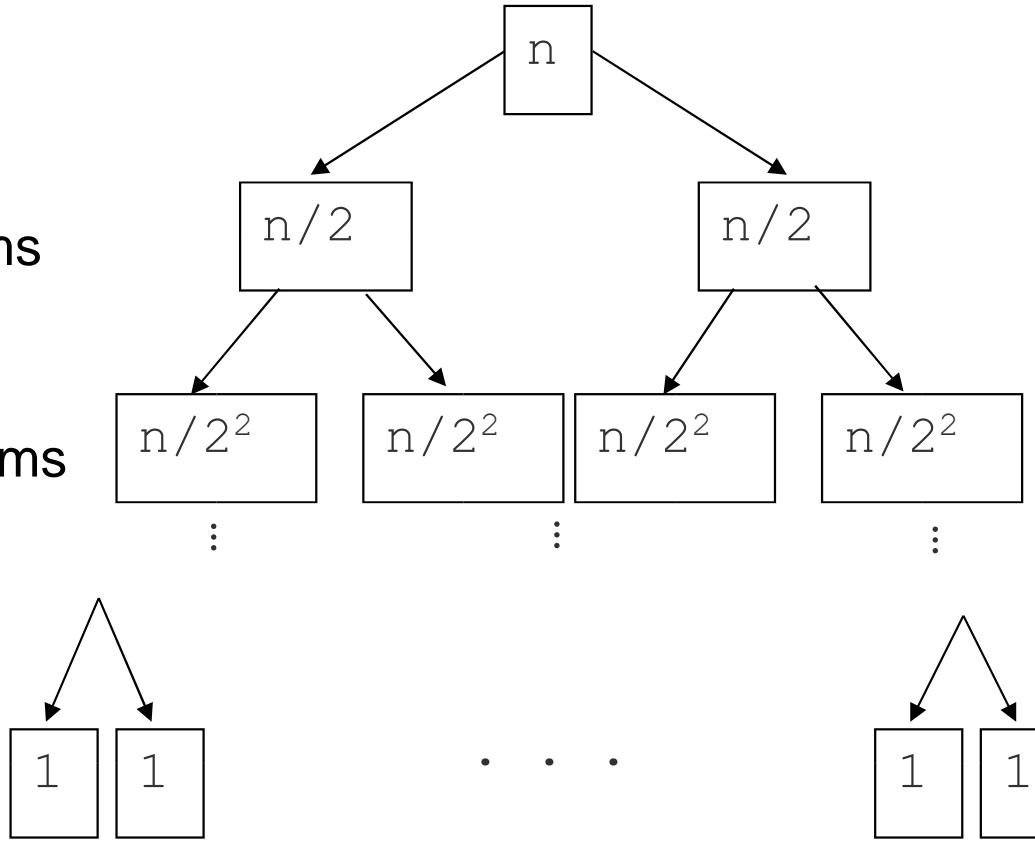
2 calls to mergeSort

Level 2:

2^2 calls to mergeSort

Level ($\lg n$):

$2^{\lg n} (= n)$ calls to mergeSort



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

Merge Sort: Analysis

- Level 0: 0 calls to `merge()`
- Level 1: 1 calls to `merge()` with $n/2$ items in each half,
 $O(1 \times 2 \times n/2) = O(n)$ time
- Level 2: 2 calls to `merge()` with $n/2^2$ items in each half,
 $O(2 \times 2 \times n/2^2) = O(n)$ time
- Level 3: 2^2 calls to `merge()` with $n/2^3$ items in each half,
 $O(2^2 \times 2 \times n/2^3) = O(n)$ time
- ...
- Level $(\lg n)$: $2^{\lg(n)-1} (= n/2)$ calls to `merge()` with $n/2^{\lg(n)}$ ($= 1$) item in each half, $O(n)$ time
- Total time complexity = $O(n \lg(n))$

Merge Sort: Analysis

- Optimal comparison-based sorting method

Merge Sort: Pros and Cons

- Pros
 - The performance is guaranteed, i.e. unaffected by original ordering of the input
 - Suitable for extremely large number of inputs
 - Can operate on the input portion by portion

- Cons
 - Not easy to implement
 - Requires additional storage during merging operation
 - $O(n)$ extra memory storage needed

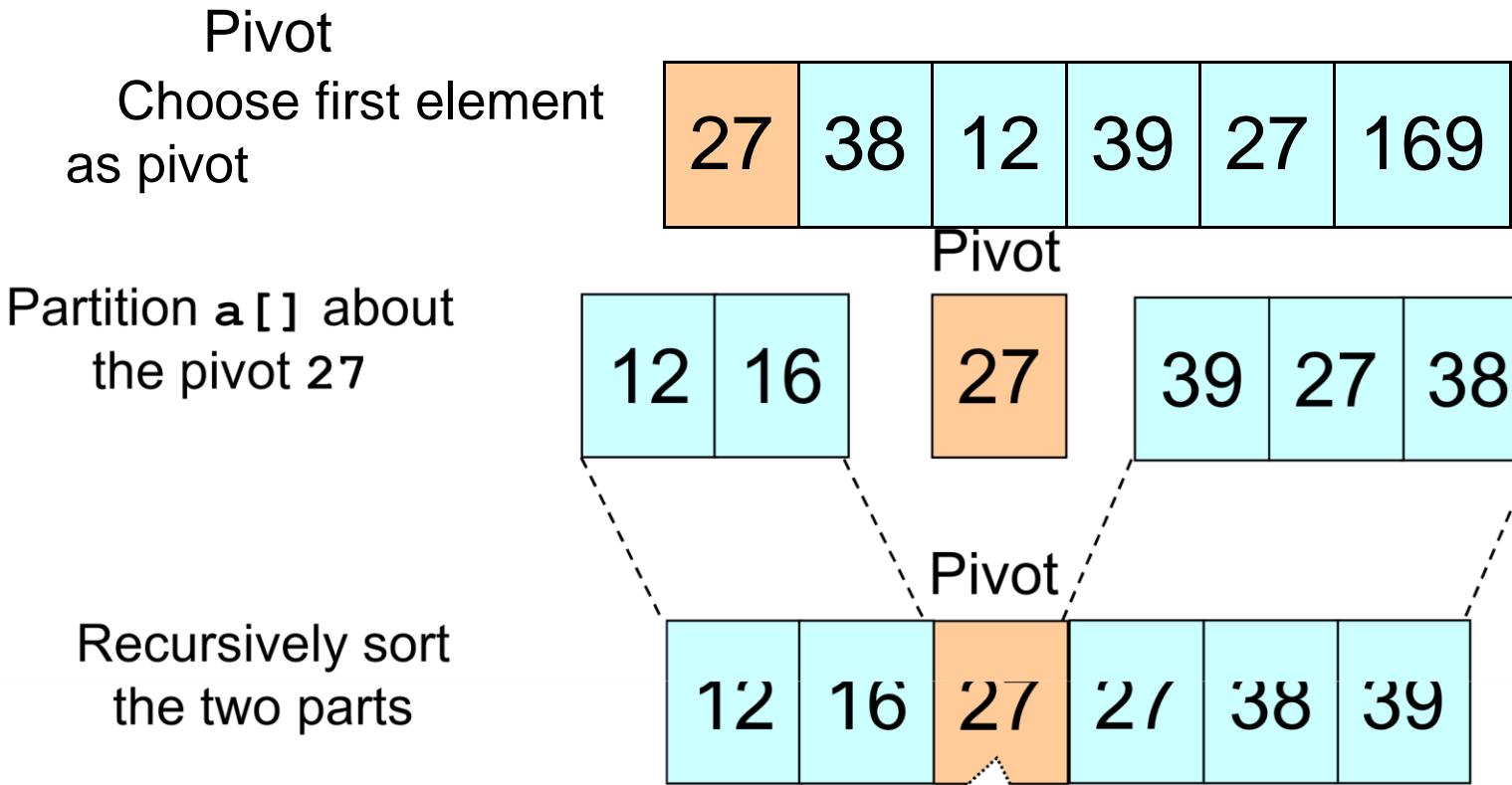
Quick Sort

Quick Sort: Idea

- Quick Sort is a divide-and-conquer algorithm
- Divide step

- Choose an item p (known as **pivot**) and partition the items of $a[i\dots j]$ into two parts
 - Items that are smaller than p
 - Items that are greater than or equal to p
 - Recursively sort the two parts
 - Conquer step
 - Do nothing!
-
- In comparison, Merge Sort spends most of the time in conquer step but very little time in divide step

Quick Sort: Divide Step Example



Notice anything special about the position of pivot in the final sorted items?

Quick Sort: Partition Algorithm

Quick Sort: Implementation

```
void quickSort(int a[], int low, int high) {  
    if (low < high) {  
        int pivotIdx = partition(a, low, high);  
        quickSort(a, low, pivotIdx-1); }pivot item  
        quickSort(a, pivotIdx+1, high); }  
    }  
}
```

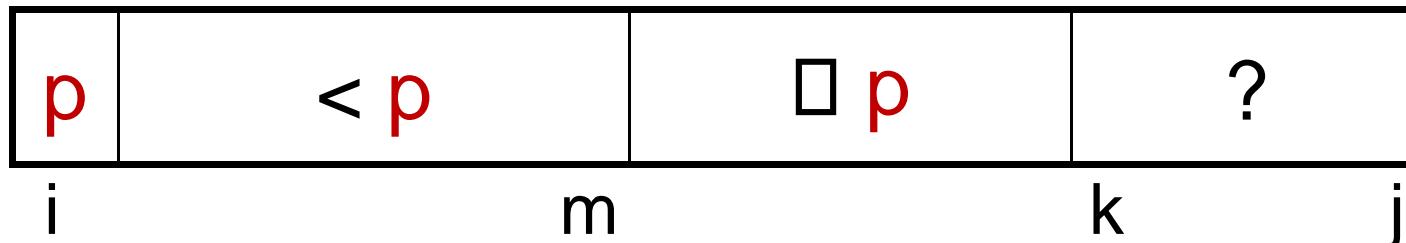
Partition

; $a[low...high]$
and return the
index of the

Recursively sort
the two portions

- partition() splits $a[low...high]$ into two portions

- $a[\text{low} \dots \text{pivot}-1]$ and $a[\text{pivot}+1 \dots \text{high}]$
- Pivot item does not participate in any further sorting
- To partition $a[i \dots j]$, we choose $a[i]$ as the pivot p
- Why choose $a[i]$? Are there other choices?
- The remaining items (i.e. $a[i+1 \dots j]$) are divided into 3 regions
- $S_1 = a[i+1 \dots m]$ where items $< p$
- $S_2 = a[m+1 \dots k-1]$ where item $\geq p$
- Unknown (unprocessed) = $a[k \dots j]$, where items are yet to be assigned to S_1 or S_2



Quick Sort: Partition Algorithm

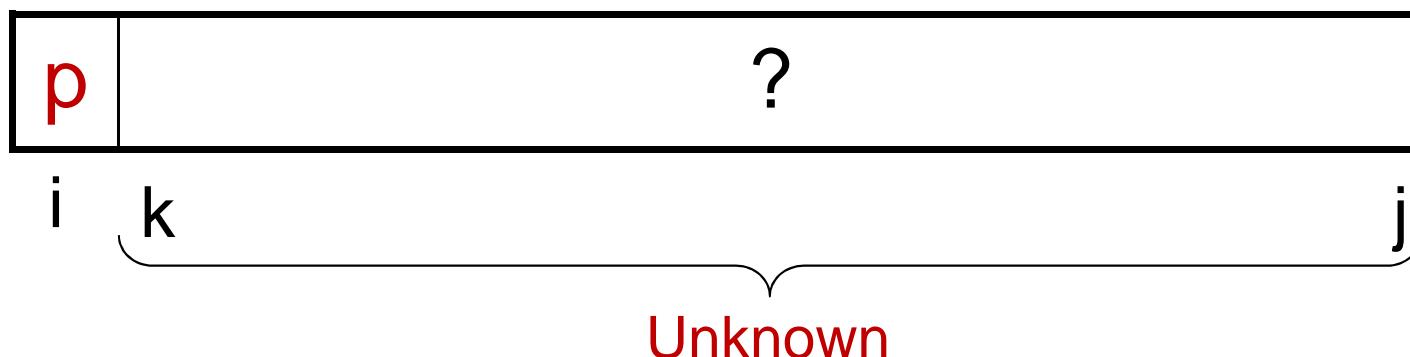
S1

S2

Unknown

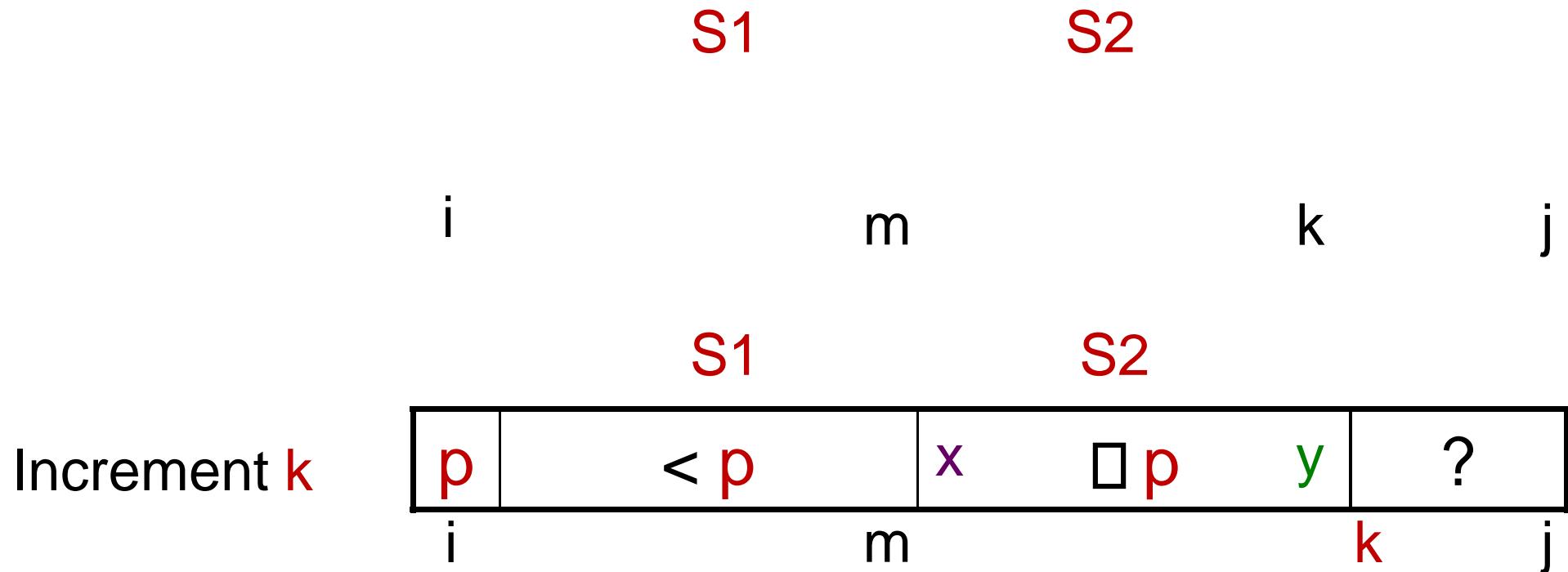
Quick Sort: Partition Algorithm

- Initially, regions **S1** and **S2** are empty
- All items excluding **p** are in the **unknown** region
- 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**



Quick Sort: Partition Algorithm

- Case 1: if $a[k] \geq p$



Quick Sort: Partition Algorithm

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

p	< p	x	$\neq p$	y	?
---	-----	---	----------	---	---

Quick Sort: Partition Algorithm

- Case 2: if $a[k] < p$

	S_1	S_2	
If $a[k]=y < p$	p	$< p$	$x \quad \square p \quad y \quad ?$
	i	m	k
Increment m	p	$< p \quad x$	$\square p \quad y \quad ?$
	i	m	k
Swap x and y	p	$< p \quad y$	$\square p \quad x \quad ?$
	i	m	k
Increment k	p	$< p \quad y$	$\square p \quad x \quad ?$
	i	m	k

Quick Sort: Partition Implementation

PS: C++ STL <algorithm> has partition subroutine too

```
int m = i,
for (int k = i+1; k <= j; k++) {
    if (a[k] < p) {
        m++;
        swap(a[k], a[m]);
    }
    else {
    }
}
swap(a[i], a[m]);
return m;
```

p is the pivot

S1 and S2 empty initially

Go through each element in unknown region

Case 2

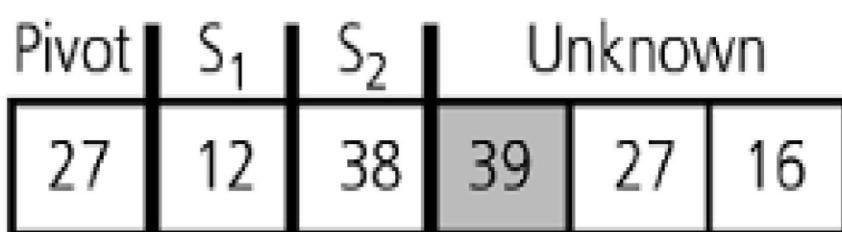
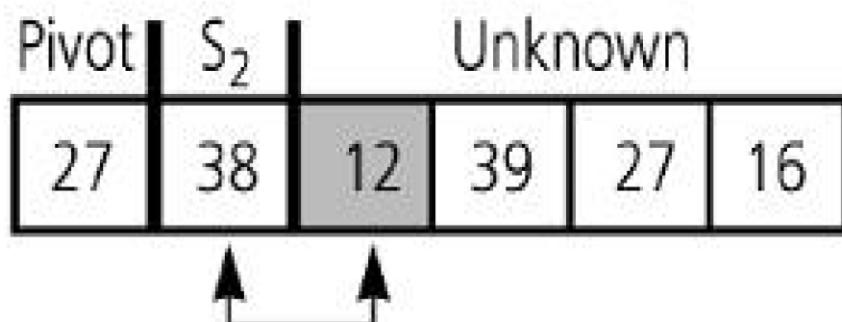
Case 1. Do nothing!

Swap pivot with a[m]

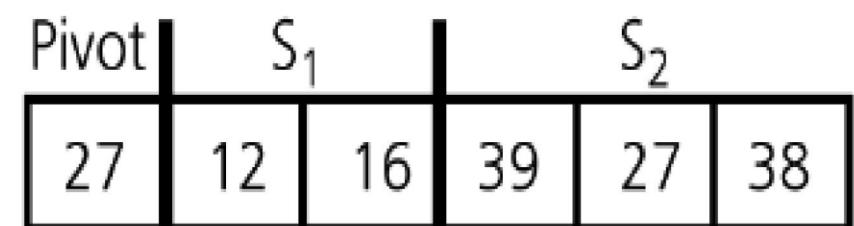
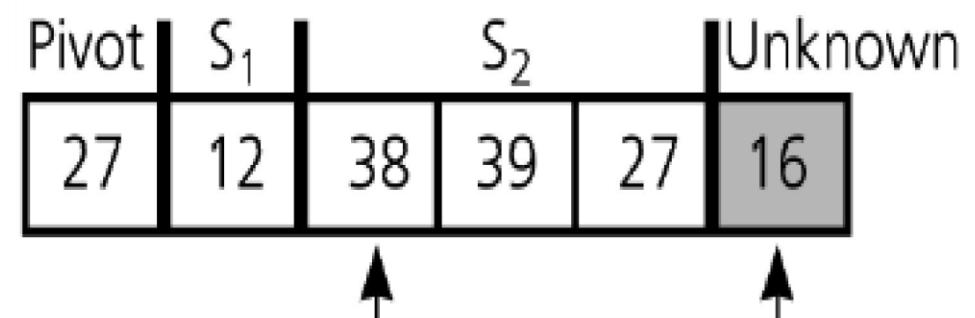
m is the index of pivot

```
int partition(int a[], int i, int j) {  
    int p = a[i];
```

Pivot	Unknown					
	27	38	12	39	27	16



Pivot	S_1	S_2	Unknown			
	27	12	38	39	27	16



S_1	Pivot	S_2				
	16	12	27	39	27	38

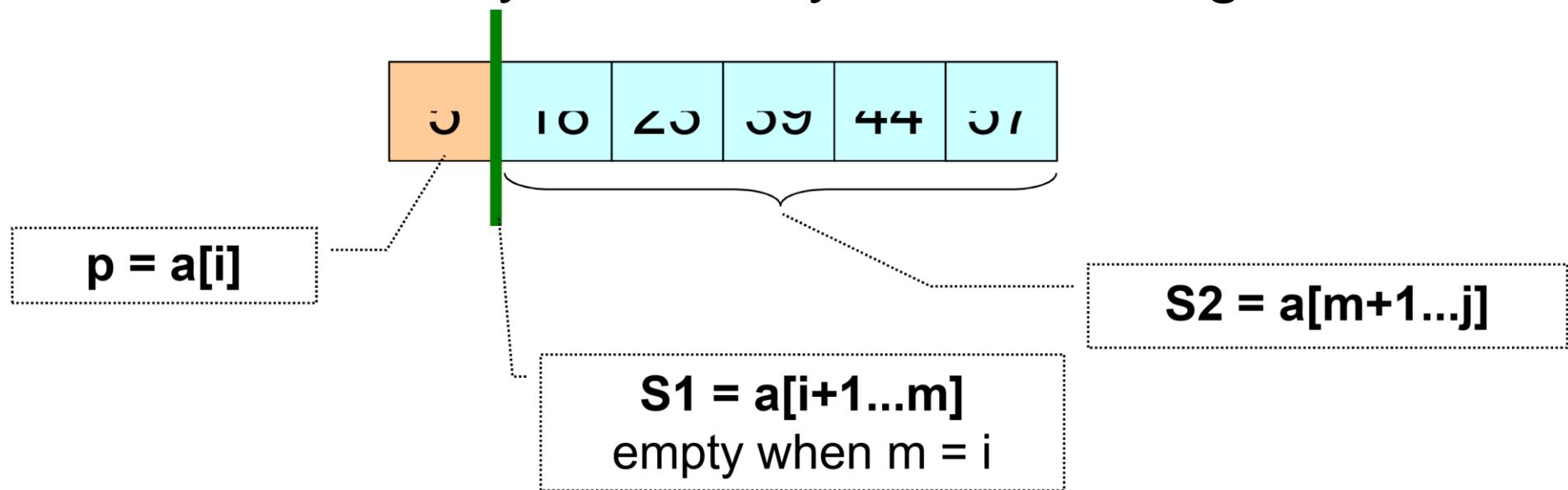
<http://visualgo.net/sorting?create=27,38,12,39,27,16&mode=Quick>

Quick Sort: Partition Analysis

- There is only a single for-loop
- Number of iterations = number of items, n , in the unknown region
- $n = \text{high} - \text{low}$
- Complexity is $O(n)$
- Similar to Merge Sort, the complexity is then dependent on the number of times `partition()` is called

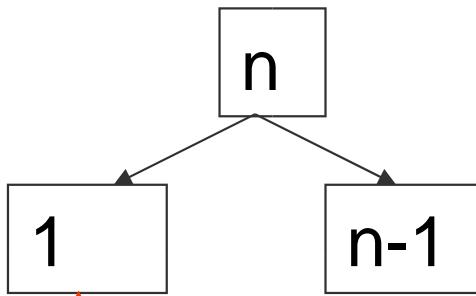
Quick Sort: Worst Case Analysis

- When the array is already in ascending order



- What is the pivot index returned by `partition()`?
- What is the effect of `swap(a, i, m)`?

- S_1 is empty, while S_2 contains every item except the pivot



1

n-1

1

n-2

1

1

contains the pivot only!

Total no.
of levels
= n

As each partition takes linear time, the algorithm in its worst case has n levels and hence it takes time $n+(n-1)+\dots+1 = O(n^2)$

Quick Sort: Best/Average Case Analysis

- Best case occurs when partition always splits the array into two equal halves
- Depth of recursion is $\log n$
- Each level takes n or fewer comparisons, so the time complexity is $O(n \log n)$
- In practice, worst case is rare, and on the average we get some good splits and some bad

ones (details in CS3230 :O) ■ Average time is also $O(n \log n)$

Lower Bound: Comparison-Based Sort

- It is known that
- All comparison-based sorting algorithms have a complexity lower bound of $n \log n$

- Therefore, any comparison-based sorting algorithm with worst-case complexity $O(n \log n)$ is optimal

- Treats each data to be sorted as a character stringIt is not using comparison, i.e. no arison between the data is neededIn each iteration
- Organize the data into groups according to the next character in each data
- The groups are then “concatenated” for next iteration

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

Combined

■ Quick Sort

- 1285 5_a 150 4746 602 5_b 8356 (pivot=1285)
- 1285 (5_a 150 602 5_b) (4746 8356)
- 5_b 5_a 150 602 1285 4746 8356

	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
Merge Sort	$O(n \lg n)$	$O(n \lg n)$	No	Yes

Quick Sort	$O(n^2)$	$O(n \lg n)$	Yes	No
Radix sort	$O(dn)$	$O(dn)$	No	yes

[CS1020E AY1617S1 Lecture 10]

Sorting Algorithms: Summary

Summary

- Comparison-Based Sorting Algorithms
- Iterative Sorting
- Selection Sort
- Bubble Sort
- Insertion Sort
- Recursive Sorting
- Merge Sort
- Quick Sort