Lecture 10

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…$n-1$]
  2. Swap *x* with the $(n-1)^{th}$ item
  3. Reduce *n* by 1 and go to Step 1

Selection Sort: Illustration

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

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

Q: How to find the largest?

- x **Unsorted items**
- x **Largest item for current iteration**
- x **Sorted items**

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

- $n-1$ (int maxIdx = i;)
- $n-1$ (for (int j = 0; j < i; j++))
- $(n-1)+(n-2)+\ldots+1 = n(n-1)/2$ (if (a[j] >= a[maxIdx]) / maxIdx = j;)
- $n-1$ (swap(a[i], a[maxIdx]);)

Total

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

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

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

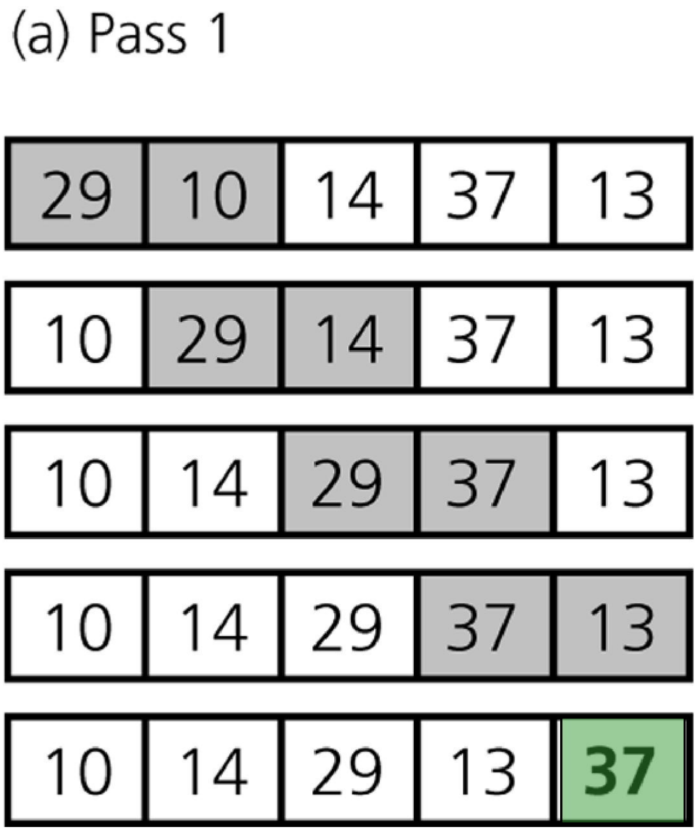
Bubble Sort

Bubble Sort: Idea

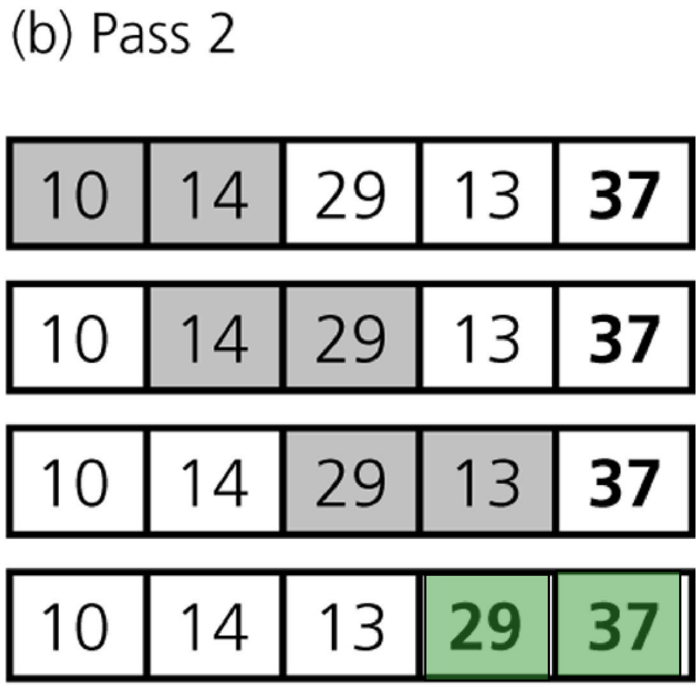
- Given an array of *n* items
  1. Compare pair of adjacent items
  2. Swap if the items are out of order
  3. Repeat until the end of array
     - The largest item will be at the last position
  4. Reduce *n* by 1 and go to Step 1

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

Bubble Sort: Illustration



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



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

Bubble Sort: Implementation

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

Step 1: Compare adjacent pairs of numbers

Step 2: Swap if the items are out 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 = $0+1+\ldots+(n-1) = n(n-1)/2$
- Total time = $c\ n(n-1)/2$ = $\mathbf{O(n^2)}$

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 the inner loop

Any swapping will invalidate the assumption

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

Bubble Sort v2.0: Analysis

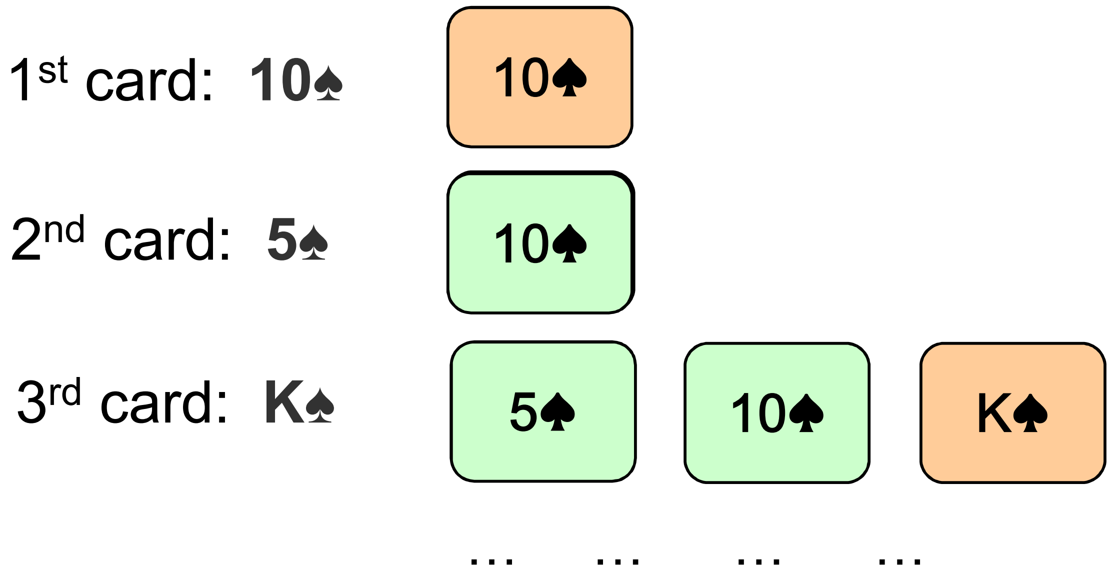
- Worst-case
  - Input is in descending order
  - Running time remains the same: O(n^2)

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

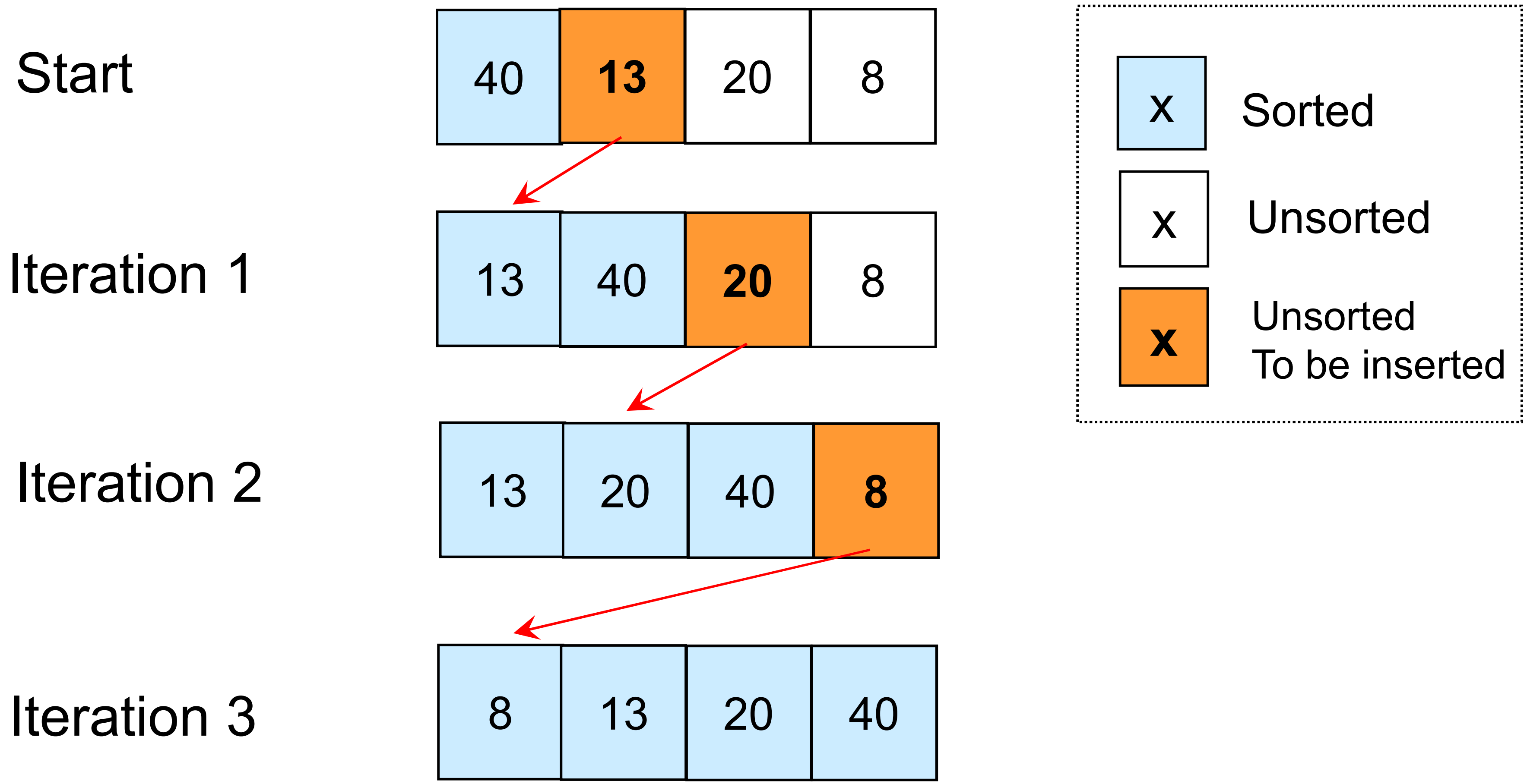
Insertion Sort

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



Insertion Sort: Illustration



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

Insertion Sort: Implementation

```
void insertionSort(int a[], int n) {
  for (int i = 1; i < n; i++) {
    int next = a[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] > 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)**

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

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

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

Merge Sort: Implementation

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

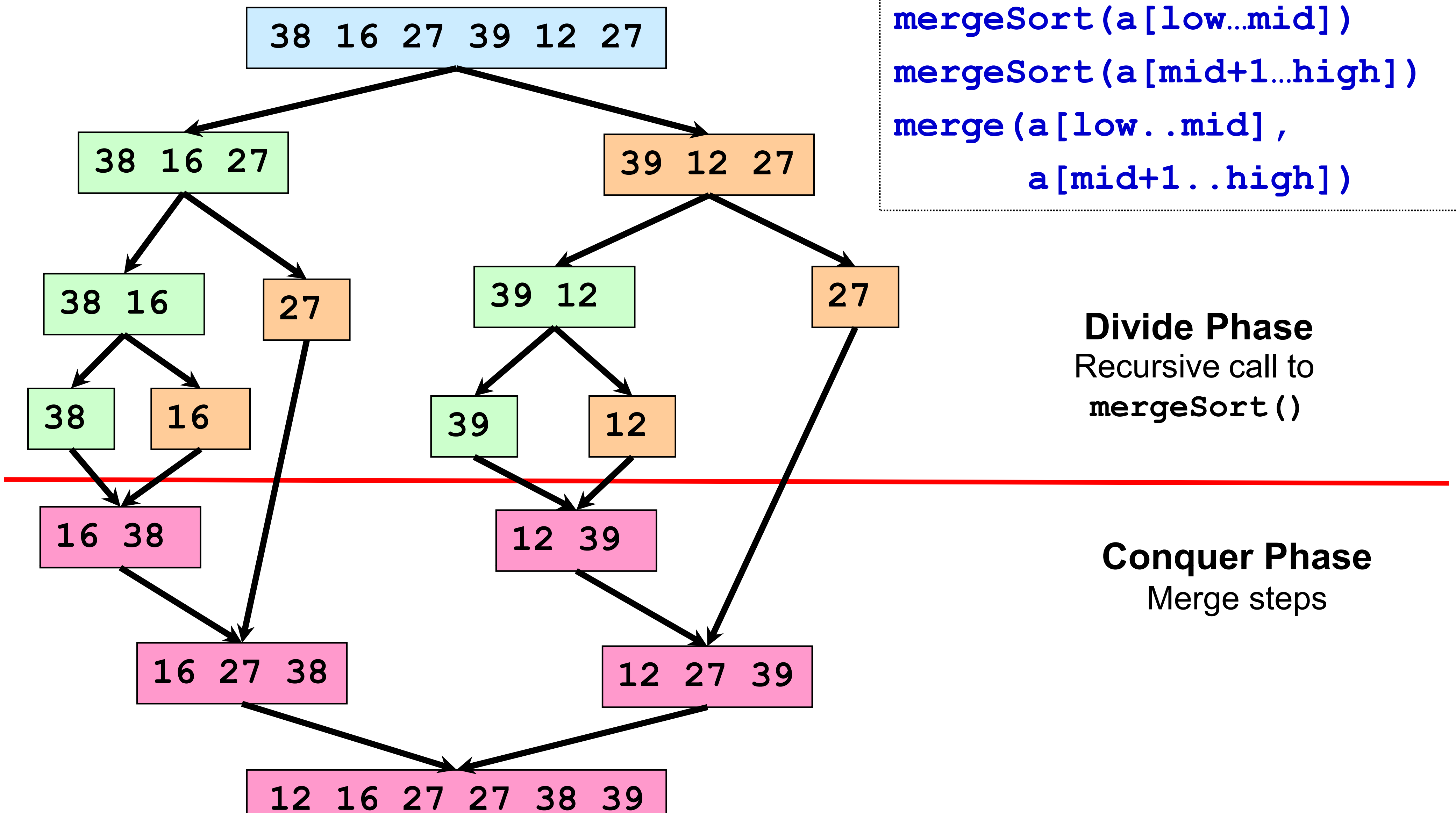
Divide a[ ] into two halves and **recursively** sort them

Conquer: merge the two sorted halves

Function to merge **a[low…mid]** and **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

Merge Sort: Example



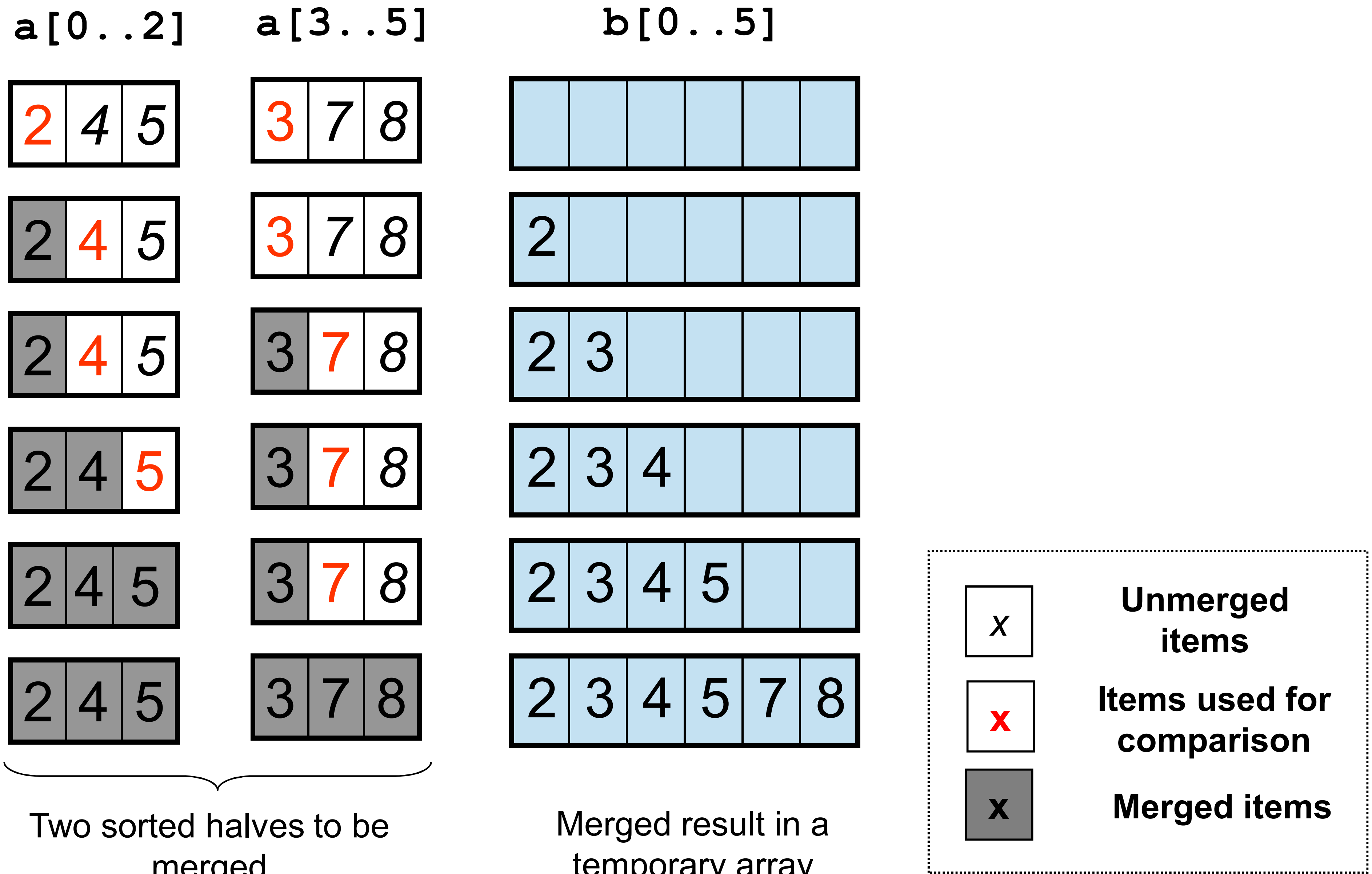

```
mergeSort(a[low…mid])
mergeSort(a[mid+1…high])
merge(a[low..mid],
      a[mid+1..high])
```

Divide Phase
Recursive call to
`mergeSort()`

Conquer Phase
Merge steps

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

Merge Sort: Merge



Two sorted halves to be merged

Merged result in a temporary array

Merge Sort: Merge Implementation

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

```
void merge(int a[], int low, int mid, int high) {
   int n = high-low+1;
   int* b = new int[n];
   int left=low, right=mid+1, bIdx=0;

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

   // continue on next slide
```

b is a temporary array to store result

Normal Merging
Where both halves have unmerged items

Merge Sort: Merge Implementation

```
// 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];

delete [] b;
}
```

Remaining items are copied into **b[]**

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

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 = (high – low + 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, number of operations ≤ $3k - 1 = O(k)$
- The important question is
  - How many times is `merge()` called?

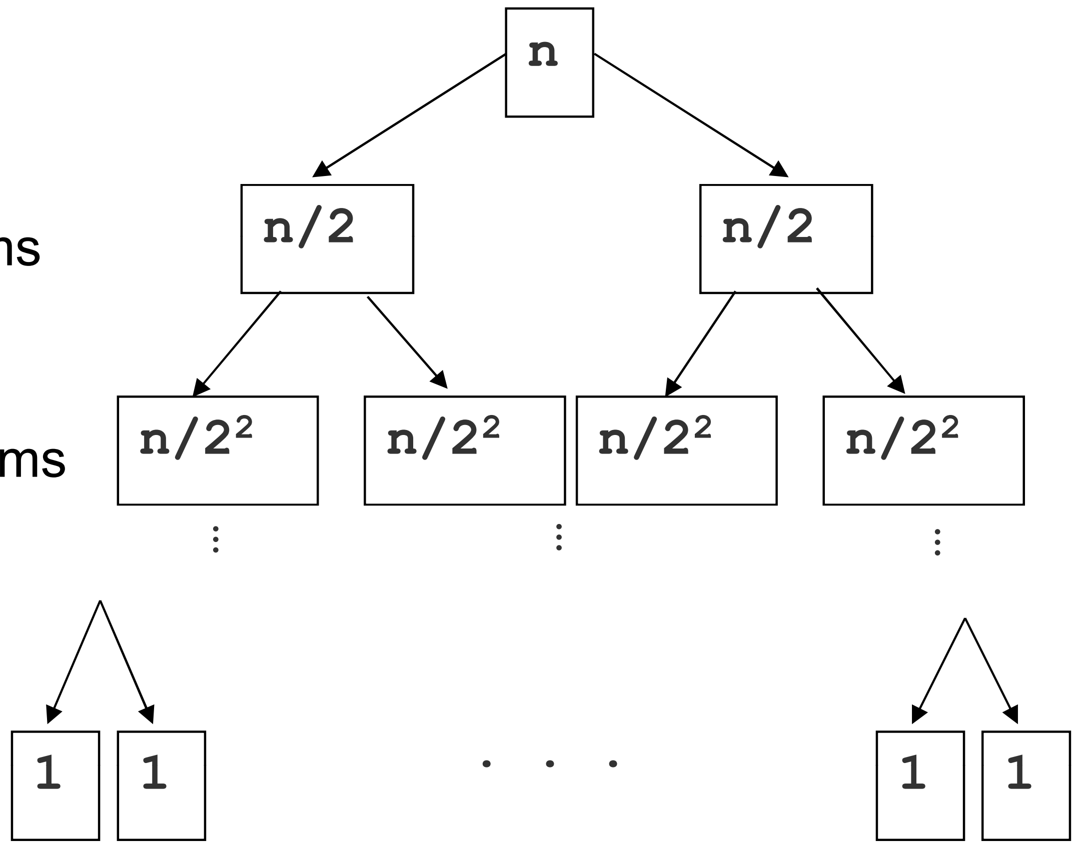
Merge Sort: Analysis

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** call 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*/2²** items in each half,
  $O(2 \times 2 \times n/2^2) = O(n)$ time
- Level 3: $\mathbf{2^2}$ calls to `merge()` with $\boldsymbol{n/2^3}$ items in each half,
  $O(2^2 \times 2 \times n/2^3) = O(n)$ time
- …
- Level (lg *n*): $\mathbf{2^{\lg(n) - 1}(= n/2)}$ calls to `merge()` with $\boldsymbol{n/2^{\lg(n)}}$ **(= 1)** item in each half, $O(n)$ time
- Total time complexity = $\mathbf{O(n \lg(n))}$
- **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

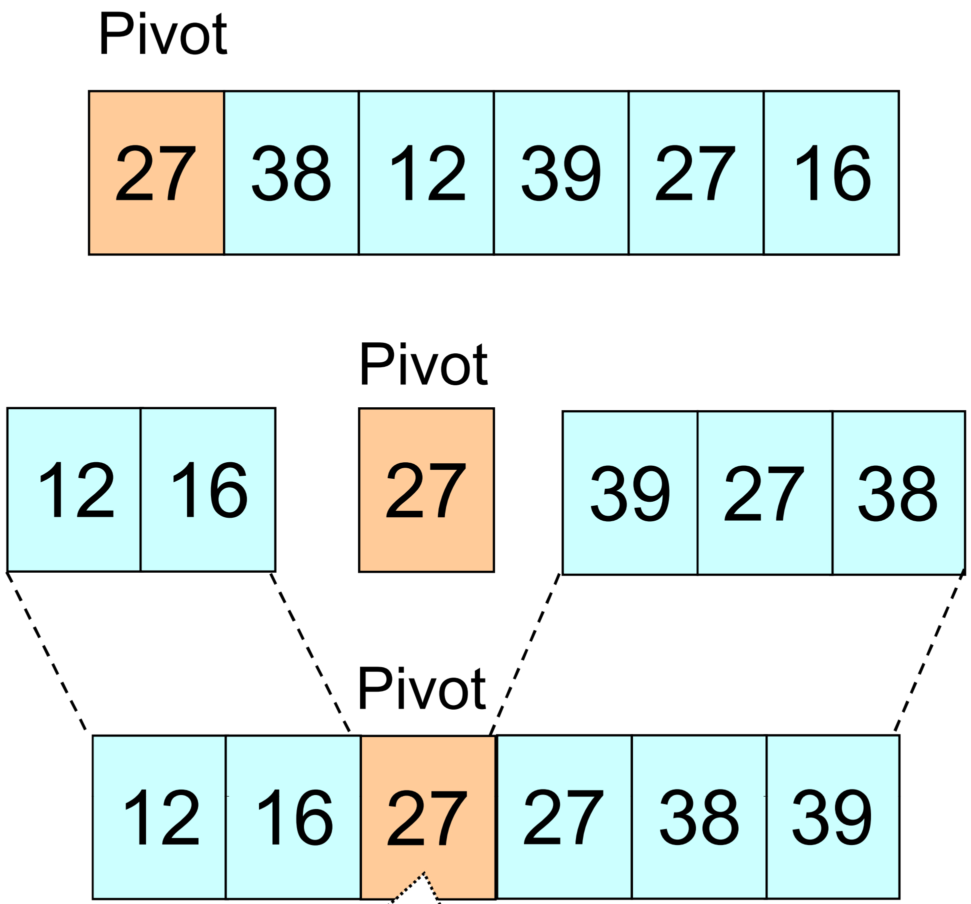
- **Quick Sort** is a divide-and-conquer algorithm
  - **Divide** step
    - Choose an item ***p*** (known as **pivot**) and partition the items of a[i...j] into two parts
      - Items that are smaller than ***p***
      - Items that are greater than or equal to ***p***
    - Recursively sort the two 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

Choose first element as pivot

Partition **`a[]`** about the pivot **`27`**

Recursively sort the two parts



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

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);
      quickSort(a, pivotIdx+1, high);
   }
}
```

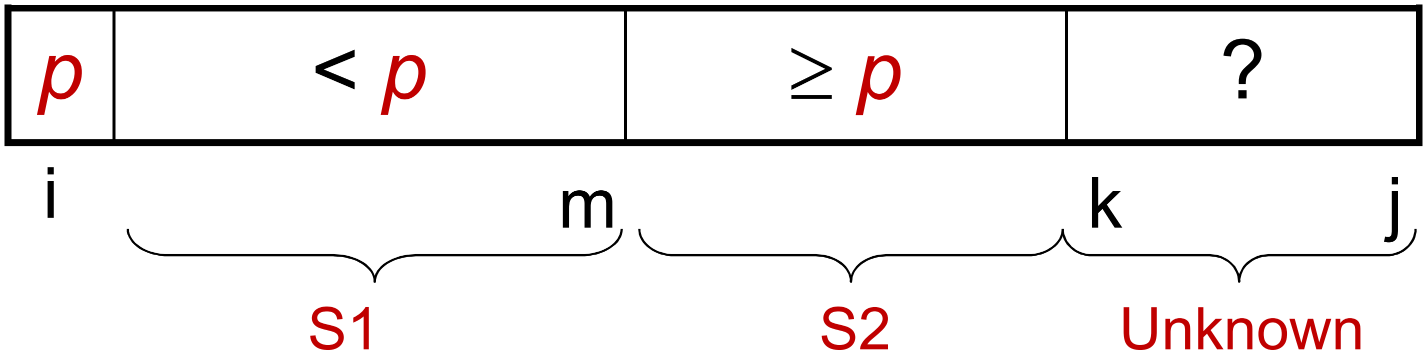
Partition **a[low...high]** and return the index of the pivot item

Recursively sort the two portions

- **partition()** splits **a[low...high]** into two portions
  - **a[low ... pivot–1]** and **a[pivot+1 ... high]**
- Pivot item does not participate in any further sorting

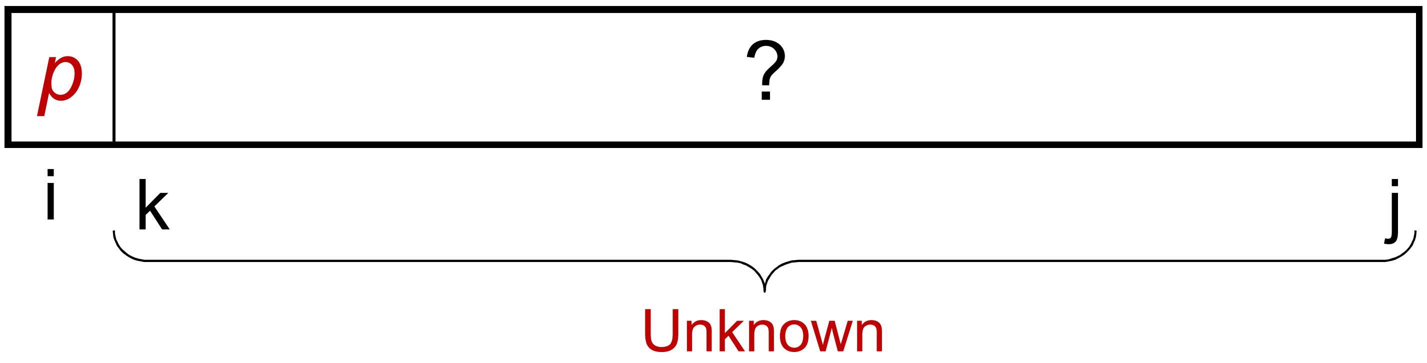
Quick Sort: Partition Algorithm

- To partition a[i...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...j]) are divided into 3 regions
  - **S1** = a[i+1...m] where items < *p*
  - **S2** = a[m+1...k-1] where item ≥ *p*
  - **Unknown** (unprocessed) = a[k...j], where items are yet to be assigned to S1 or S2



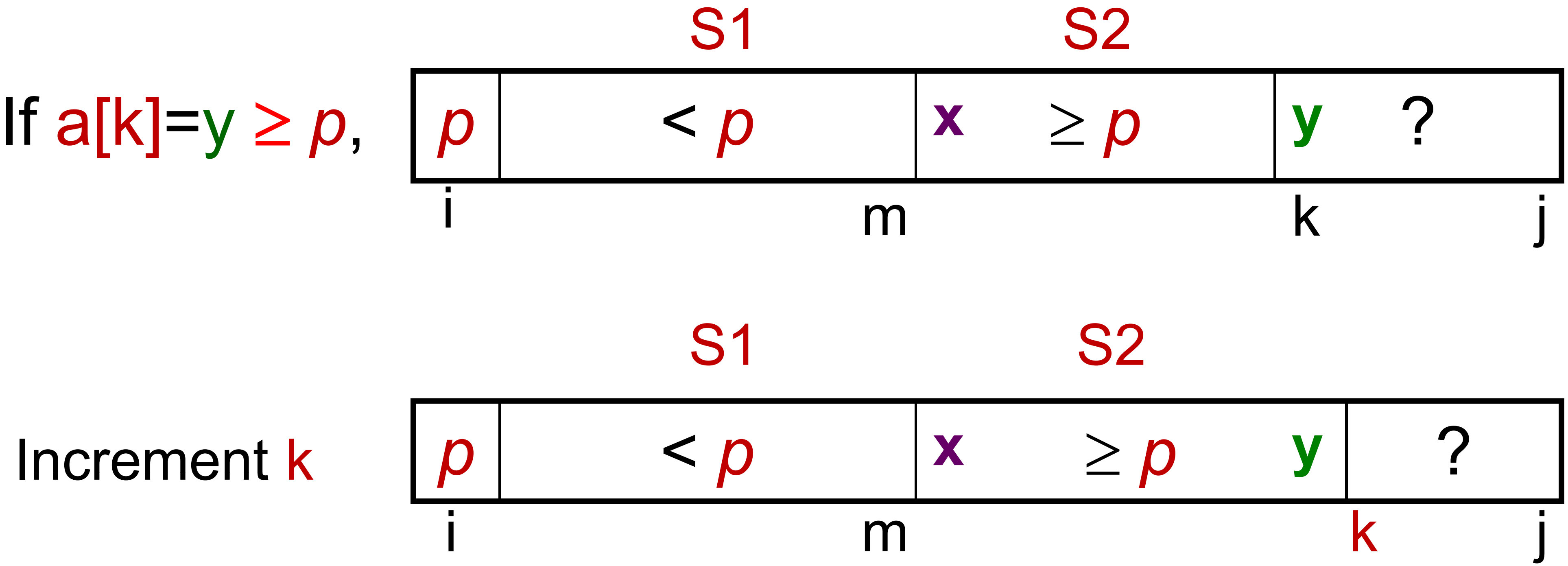
Quick Sort: Partition Algorithm

- 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

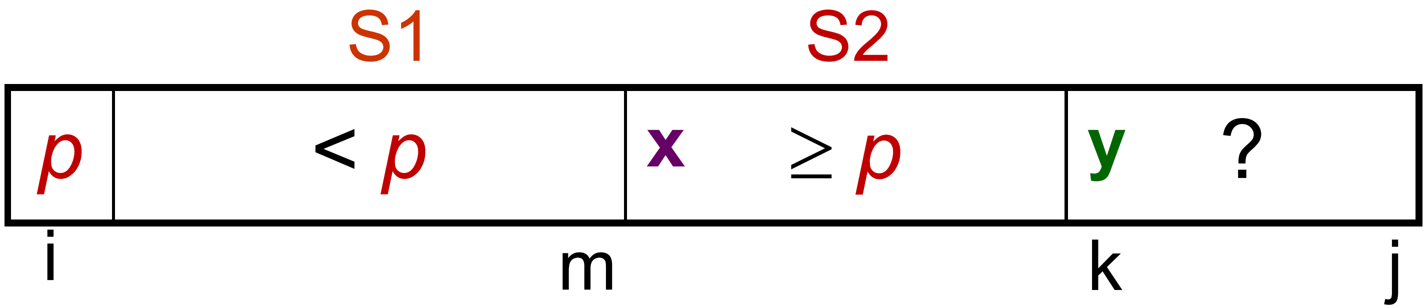
- Case 1: if a[k] >= *p*



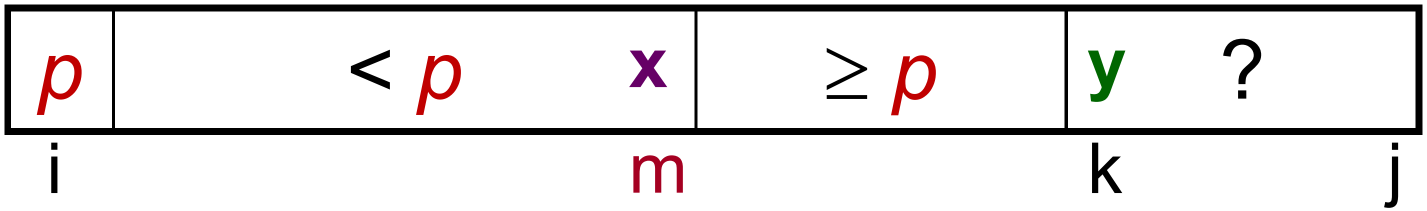
Quick Sort: Partition Algorithm

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

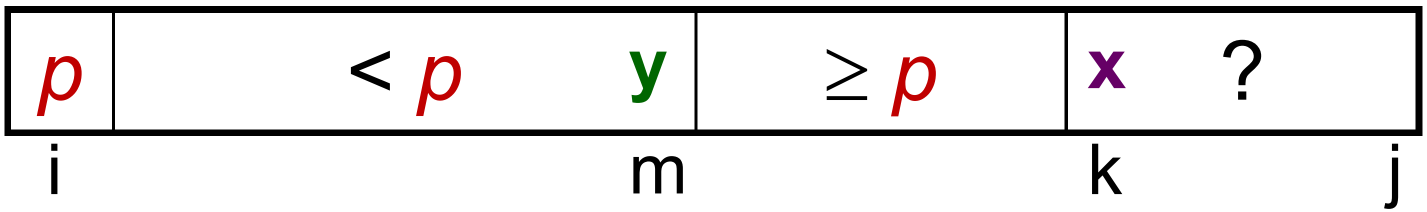
If a[k]=y < *p*



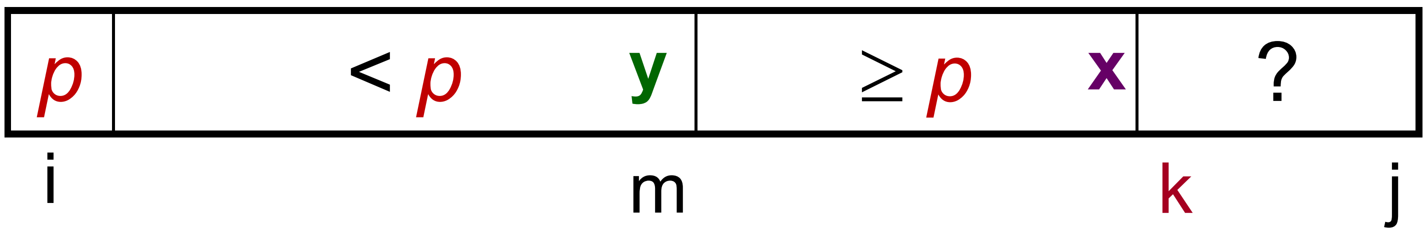
Increment m



Swap x and y



Increment k



Quick Sort: Partition Implementation

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

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

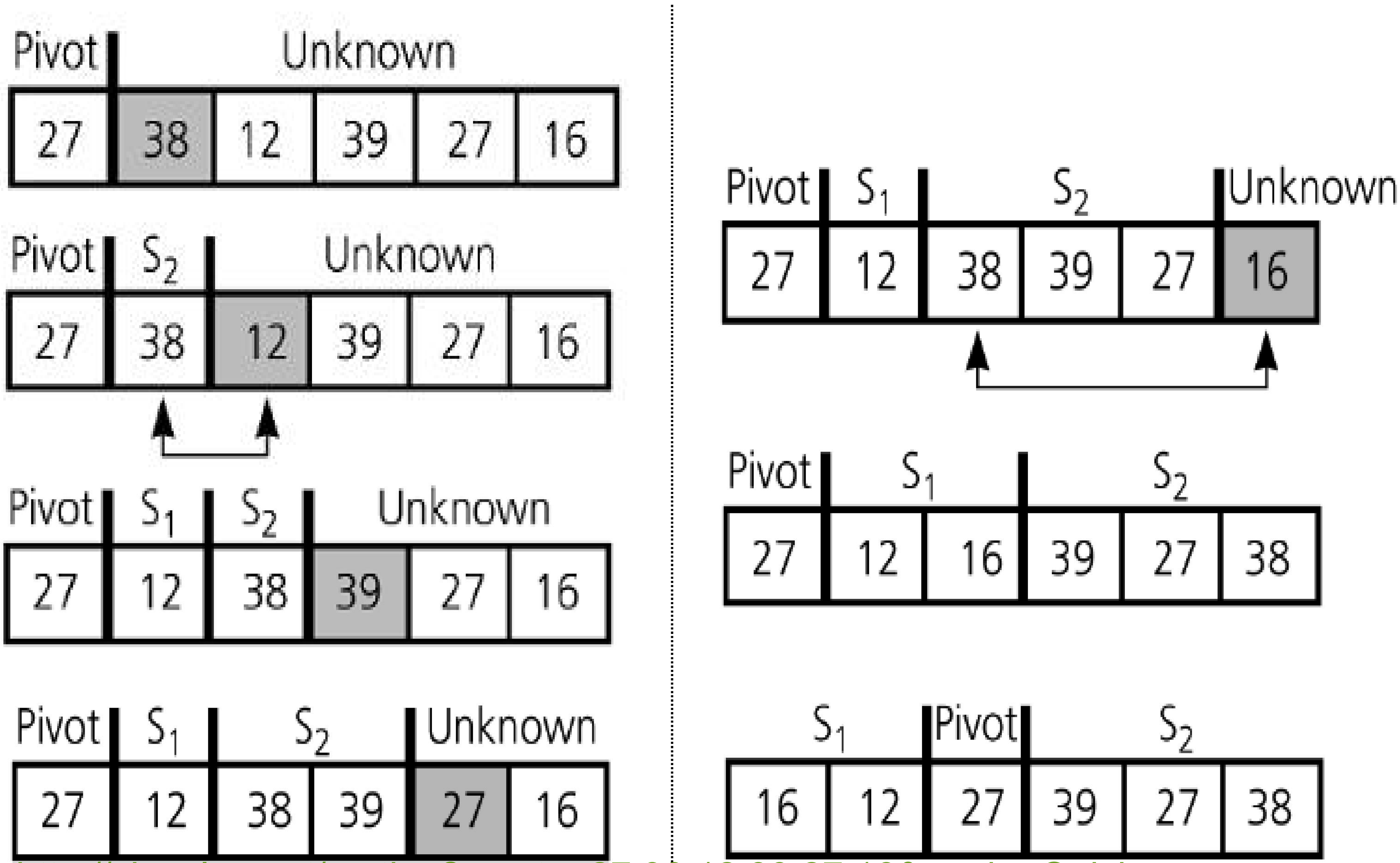
Case 2

Case 1: Do nothing!

Swap pivot with **a[m]**

m is the index of pivot

Quick Sort: Partition Example



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

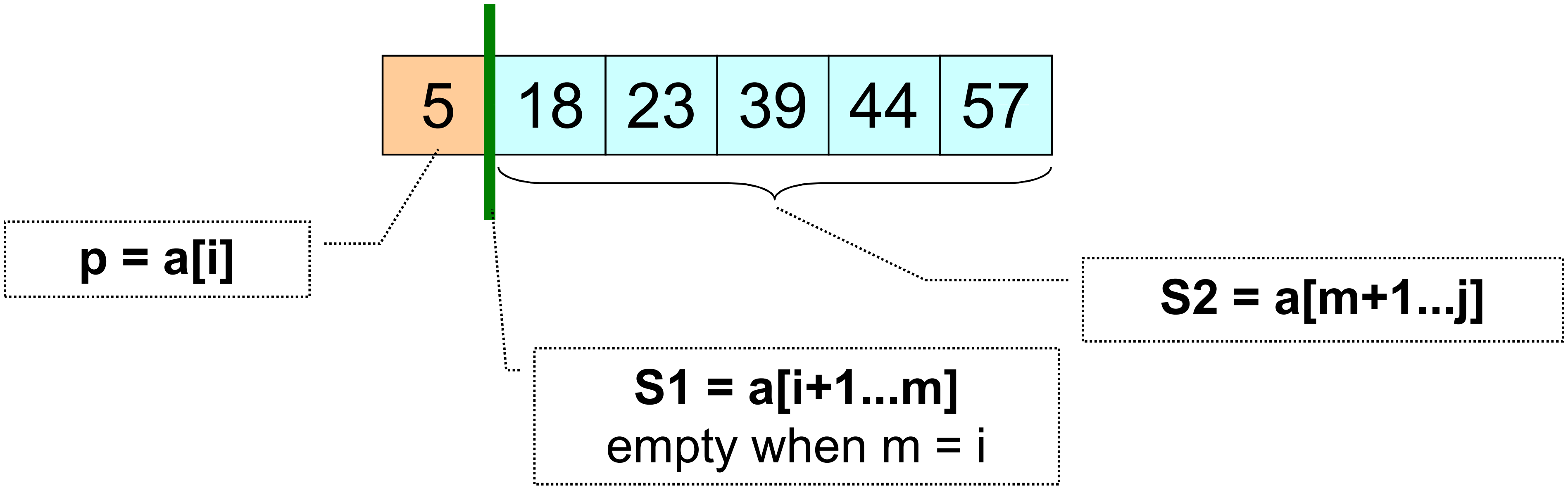
Quick Sort: Partition Analysis

- There is only a single for-loop
  - Number of iterations = number of items, *n*, in the unknown region
    - *n* = high – low
  - Complexity is O(*n*)

- Similar to **Merge Sort**, the complexity is then dependent on the number of times **partition()** is called

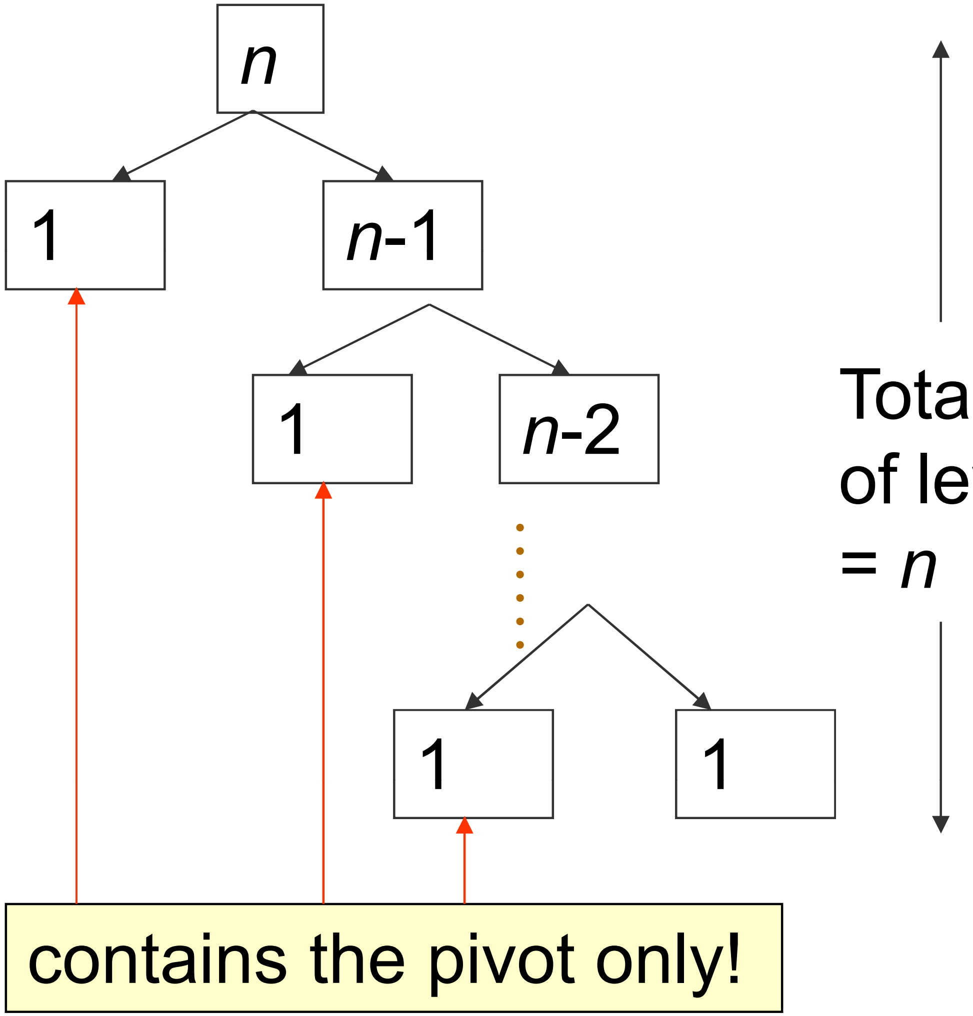
Quick Sort: Worst Case Analysis

- 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)?
- S1 is empty, while S2 contains every item except the pivot

Quick Sort: Worst Case Analysis



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

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**

Radix Sort

Radix Sort: Idea

- Treats each data to be sorted as a **character string**
- It is not using comparison, i.e. **no comparison** between the data is needed
- In each iteration
  - Organize the data into groups according to the next character in each data
  - The groups are then "concatenated" for next iteration

Radix Sort: Example

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150	Original integers
(156**0**, 215**0**) (106**1**) (022**2**) (012**3**, 028**3**) (215**4**, 000**4**)	Grouped by fourth digit
1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004	Combined
(00**0**4) (02**2**2, 01**2**3) (21**5**0, 21**5**4) (15**6**0, 10**6**1) (02**8**3)	Grouped by third digit
0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283	Combined
(0**0**04, 1**0**61) (0**1**23, 2**1**50, 2**1**54) (0**2**22, 0**2**83) (1**5**60)	Grouped by second digit
0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560	Combined
(**0**004, **0**123, **0**222, **0**283) (**1**061, **1**560) (**2**150, **2**154)	Grouped by first digit
0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154	Combined (sorted)

Radix Sort: Implementation

```
void radixSort(vector<int>& v, int d) {
   int i;
   int power = 1;
   queue<int> digitQueue[10];

   for (i = 0; i < d; i++) {
      distribute(v, digitQueue, power);
      collect(digitQueue, v);
      power *= 10;
   }
}
```

10 groups. Each is a queue to retain the order of item

- **distribute()**: Organize all items in **v** into groups using digit indicated by the power
- **collect()**: Place items from the groups back into **v**, i.e. "concatenate" the groups

Radix Sort: Implementation

```
void distribute(vector<int>& v,
                queue<int> digitQ[], int power) {
  int digit;
  for (int i = 0; i < v.size(); i++){
    digit = (v[i]/power) % 10;
    digitQ[digit].push(v[i]);
  }
}
```

- Question
  - How do we extract the digit used for the current grouping?

Radix Sort: Implementation

```
void collect(queue<int> digitQ[], vector<int>& v) {
  int i = 0, digit;

  for (digit = 0; digit < 10; digit++)
    while (!digitQ[digit].empty()) {
      v[i] = digitQ[digit].front();
      digitQ[digit].pop();
      i++;
    }
}
```

- Basic Idea
  - Start with `digitQ[0]`
    - Place all items into vector **v**
  - Repeat with `digitQ[1]`, `digitQ[2]`, ...

Radix Sort: Analysis

- For each iteration
  - We go through each item once to place them into group
  - Then go through them again to concatenate the groups
  - Complexity is O(*n*)
- Number of iterations is *d*, the maximum number of digits (or maximum number of characters)
- Complexity is thus **O(*dn*)**

Properties of Sorting

In-Place Sorting

- A sort algorithm is said to be an **in-place** sort
  - If it requires only a **constant amount** (i.e. O(1)) of **extra space** during the sorting process

- Questions
  - **Merge Sort** is not in-place, why?
  - Is **Quick Sort** in-place?
  - Is **Radix Sort** in-place?

Stable Sorting

- A sorting algorithm is **stable** if the **relative order** of **elements with the same key value** is preserved by the algorithm

- Example application of stable sort
  - Assume that **names** have been sorted in alphabetical order
  - Now, if this list is sorted again by **tutorial group number**, a stable sort algorithm would ensure that all students in the same tutorial groups still appear in alphabetical order of their names

Non-Stable Sort

- Selection Sort

  1285 5_a 4746 602 5_b (8356)

  1285 5_a 5_b 602 (4746 8356)

  602 5_a 5_b (1285 4746 8356)

  5_b 5_a (602 1285 4746 8356)

- 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

Sorting Algorithms: Summary

	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

Summary

- Comparison-Based Sorting Algorithms
  - Iterative Sorting
    - Selection Sort
    - Bubble Sort
    - Insertion Sort
  - Recursive Sorting
    - Merge Sort
    - Quick Sort
- Non-Comparison-Based Sorting Algorithms
  - Radix Sort
- Properties of Sorting Algorithms
  - In-Place
  - Stable