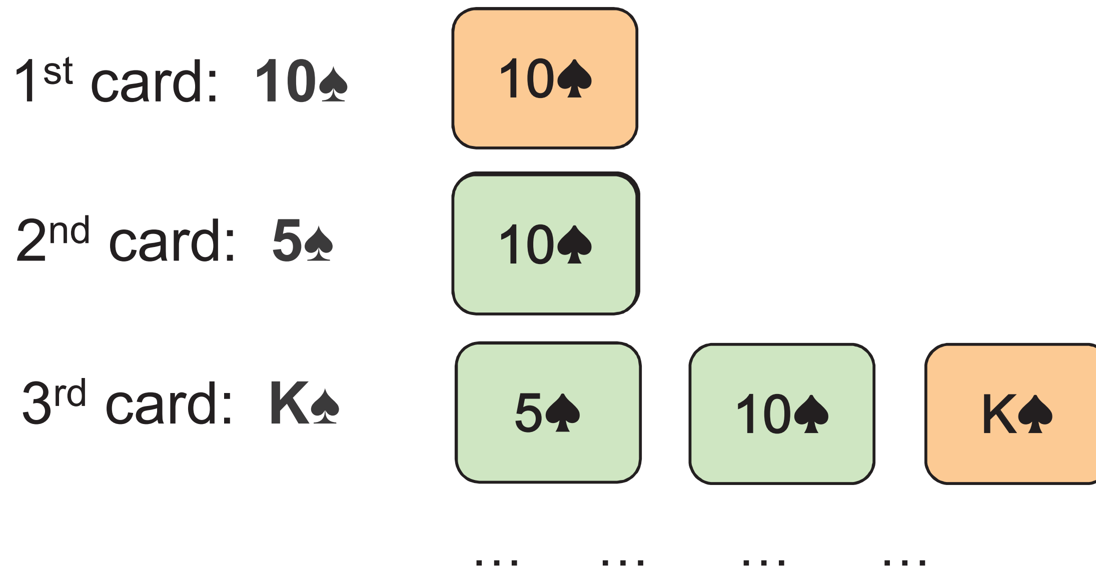
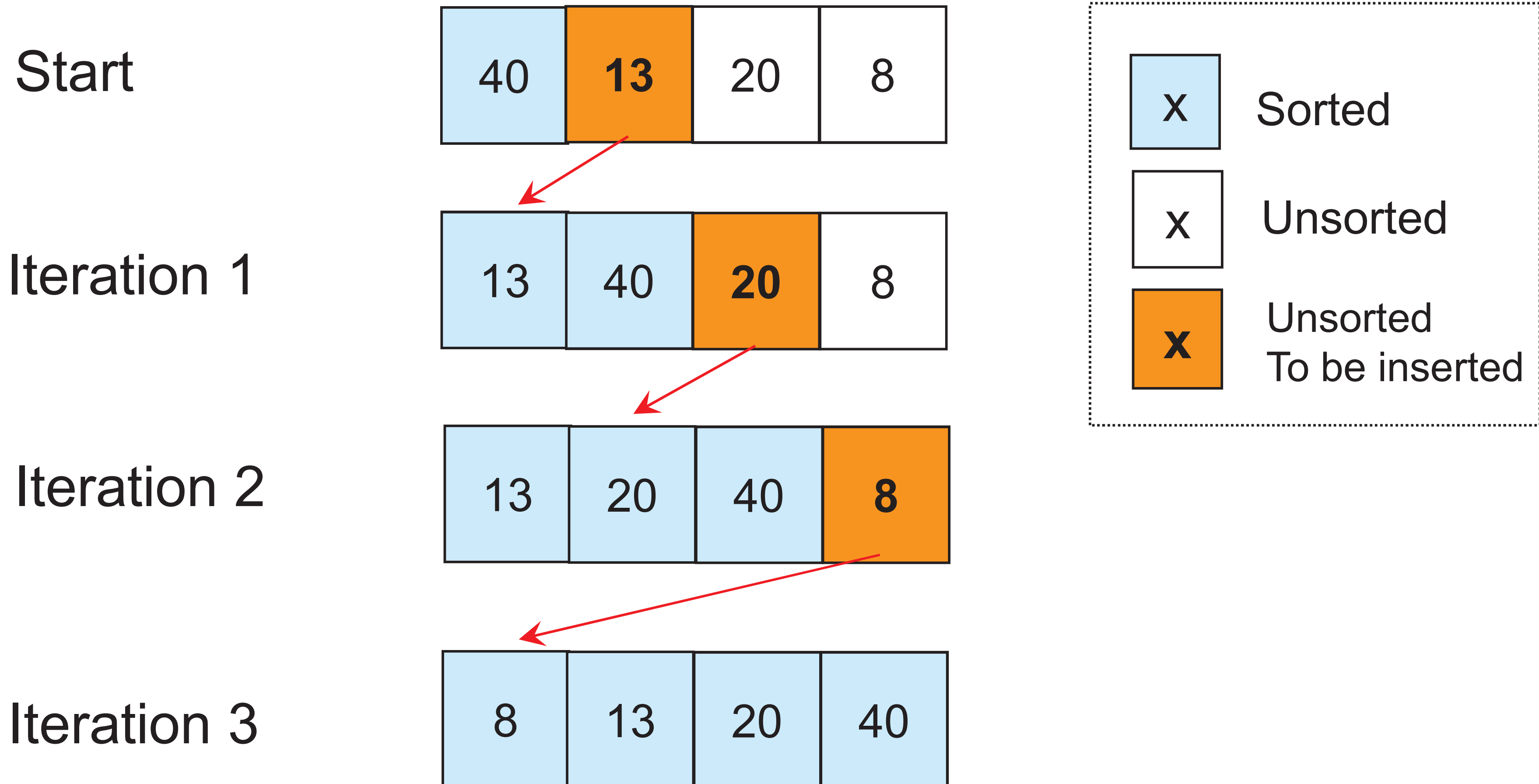
Insertion Sort

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]**

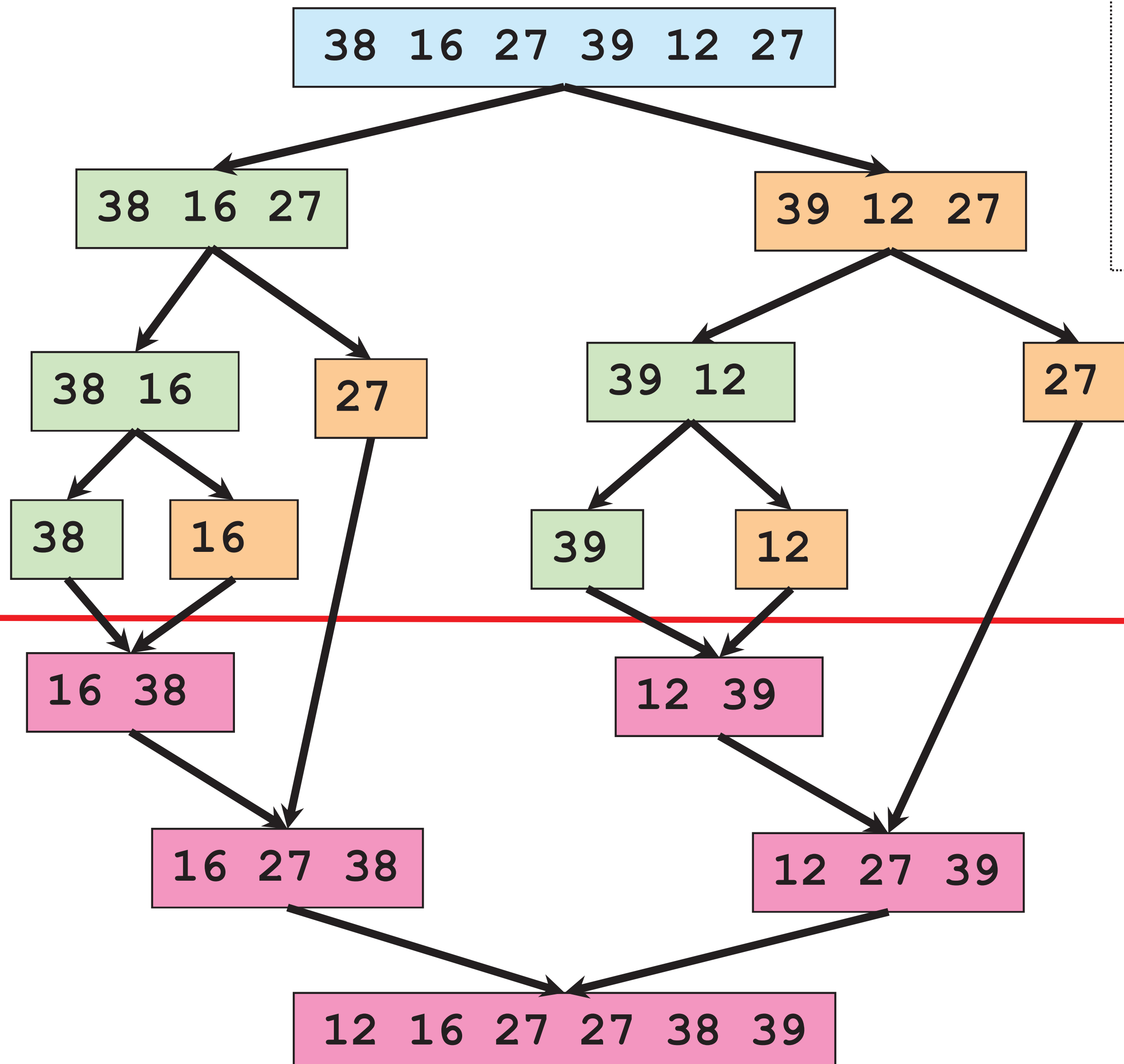
Divide a[ ] into two halves and **recursively** sort them

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])
```

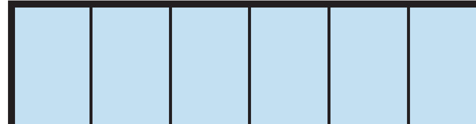
Divide Phase
Recursive call to `mergeSort()`

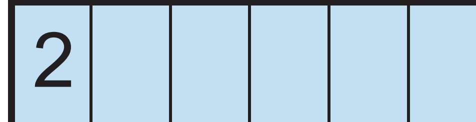
Conquer Phase
Merge steps

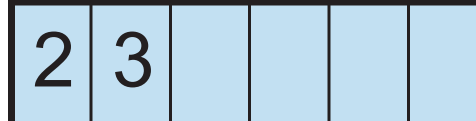
http://visualgo.net/sorting?create=38,16,27,39,12,27&mode=Merge

Merge Sort: Merge

a[0..2] a[3..5] b[0..5]

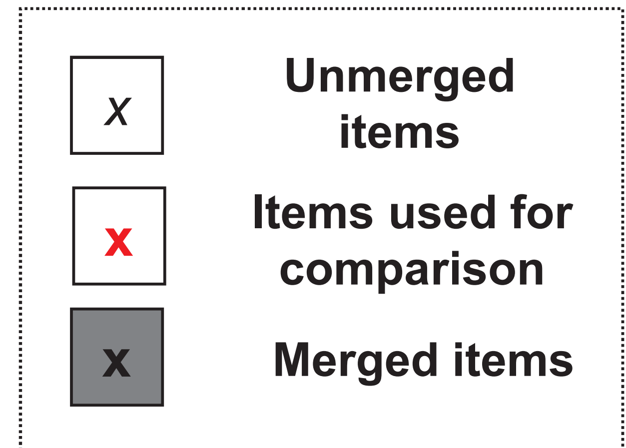






Two sorted halves to be merged

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 $\leq k - 1$
  - 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?

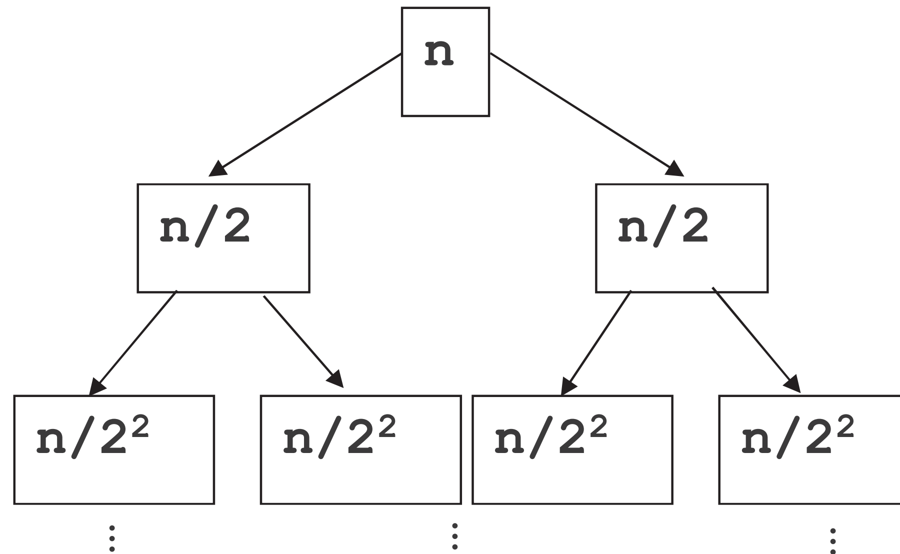
Merge Sort: Analysis

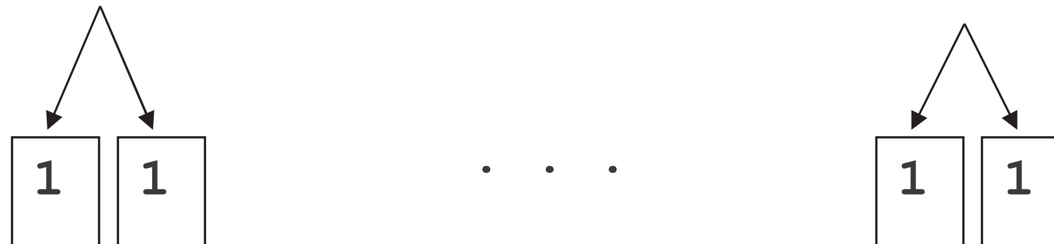
Level 0:
mergeSort `n` items

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:
$\mathbf{2^2}$ calls to mergeSort

Level (**lg *n***):
$\mathbf{2^{\lg n}}$**(= *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: **2²** calls to `merge()` with ***n*/2³** 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 $\mathbf{n/2^{\lg(n)} (= 1)}$ 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