DATA STRUCTURE AND ALGORITHMS

LECTURE 2

Sorting

Reference links:

https://cs.nyu.edu/courses/fall17/CSCI-UA.0102-007/notes.php

https://www.comp.nus.edu.sg/~stevenha/cs2040.html

https://visualgo.net/en/sorting

Lecture outline

- Why sorting?
- Sorting applications
- Simple sort algorithms
  - Bubble Sort; Selection Sort; Insertion Sort; Shuffle sort
- Effective sort algorithms
  - Quick Sort; Merge Sort
- Some other algorithms
  - Radix Sort, Heap Sort
- Execirses

Why sorting?

Sorting

- Sorting - puts elements of a list in a certain order.
- Sorting is one of the fundamental problems in computer science.
- The most-used orders are numerical order and lexicographical order.
- Efficient sorting is important for optimizing the use of other algorithms (such as searching and merging algorithms).
- The sorting problem has attracted a great deal of research due to the complexity of solving it efficiently..

Sorting Algorithms

- There are many sorting algorithms, many of them provide a gentle introduction to a variety of core algorithm concepts.
- Although many people consider that is a solved problem, but useful new sorting algorithms are still being invented.
- Common and well-known sorting algorithms: bubble sort, selection sort, insertion sort, quicksort, merge sort, heap sort.

.

Sorting Applications

- Uniqueness testing – Kiểm tra tính duy nhất
- Deleting duplicates – Xóa các bản trùng
- Prioritizing events – Sự kiên ưu tiên
- Frequency counting – Đếm tần xuất
- Reconstructing the original order – Sắp lại trật tự
- Set intersection/union - Bài toán tập hợp
- Efficient searching – Tìm kiếm hiệu quả

Simple Sorting Agorithms

- Bubble Sort – Sắp xếp nổi bọt
- Selection Sort – Sắp xếp chèn
- Insertion Sort – Sắp xếp chọn
- Shuffle Sort – “Xóc” ngẫu nhiên

Simple Sorting Agorithms

Bubble Sort

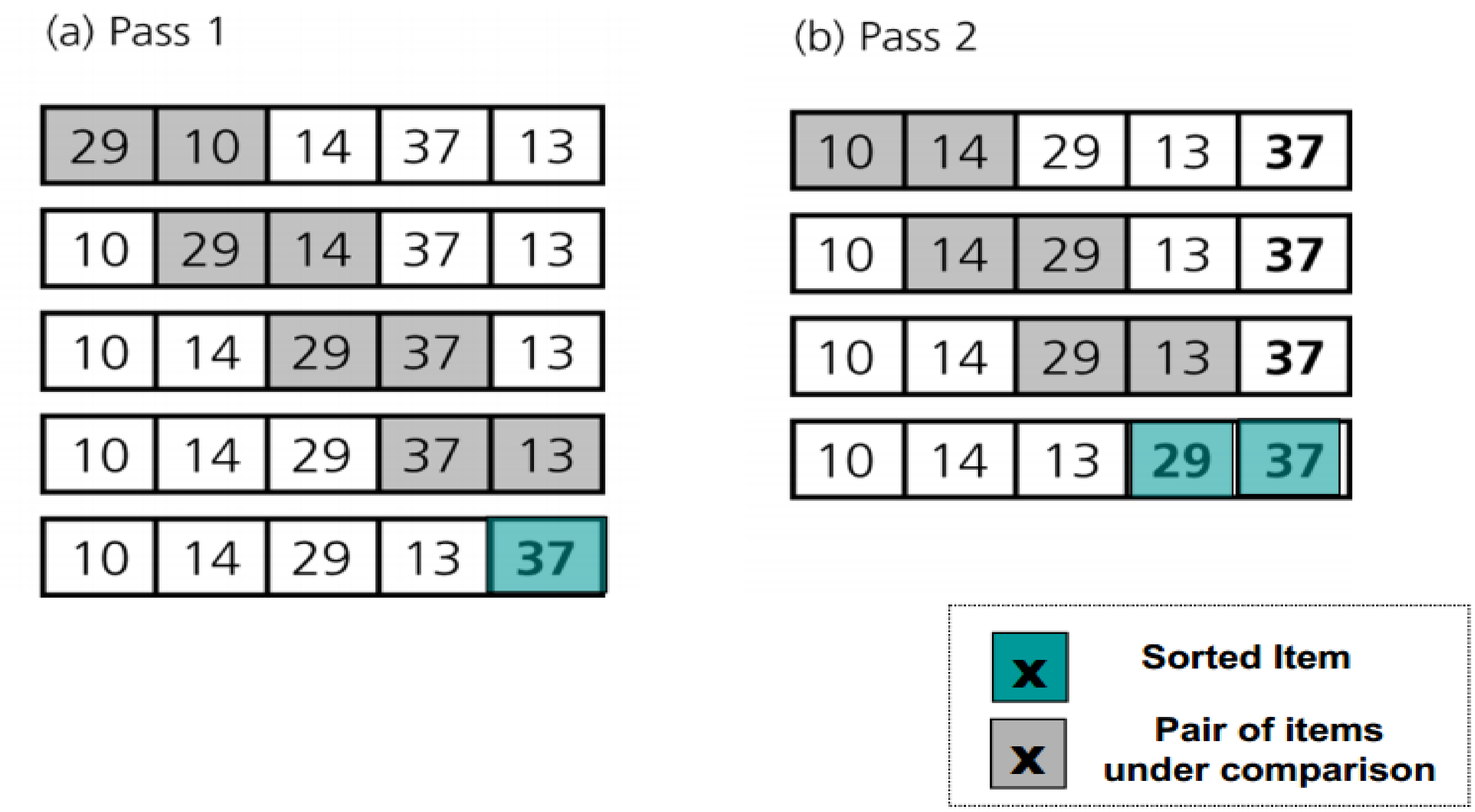
Bubble Sort : Idea

- Given an array of *n* items, sort the items ascending
  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. Go to step 1 with *n* reduced by 1
- Analogy:

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

Bubble Sort : Illustration



Bubble Sort : Pseudo code

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

i

Step 1. Compare adjacent pairs of numbers

Step 2. Swap if the items are out of order

29	10	14	37	13

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

Bubble Sort : Early Termination

- Bubble Sort is inefficient with a O(n^2) time complexity
- However, how does it do when the array like this:

  3 , 6 , 11 , 25 , 39

- Still compares and swaps – waste time.
- Idea:

  If we went through the inner loop with no swapping

  ⇨ the array is sorted ⇨ can stop early!

Bubble Sort : Pseudo Code Ver 2.0

```
void bubbleSort2(int a[], int N) {
   for (int i = 0; i < N; ++i) {
      bool is_sorted = true;
      for (int j = 1; j < N-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 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 Ver 2.0 : 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: $O(n)$
- **Average-case**
  - input is in disorder: $O(n^2)$

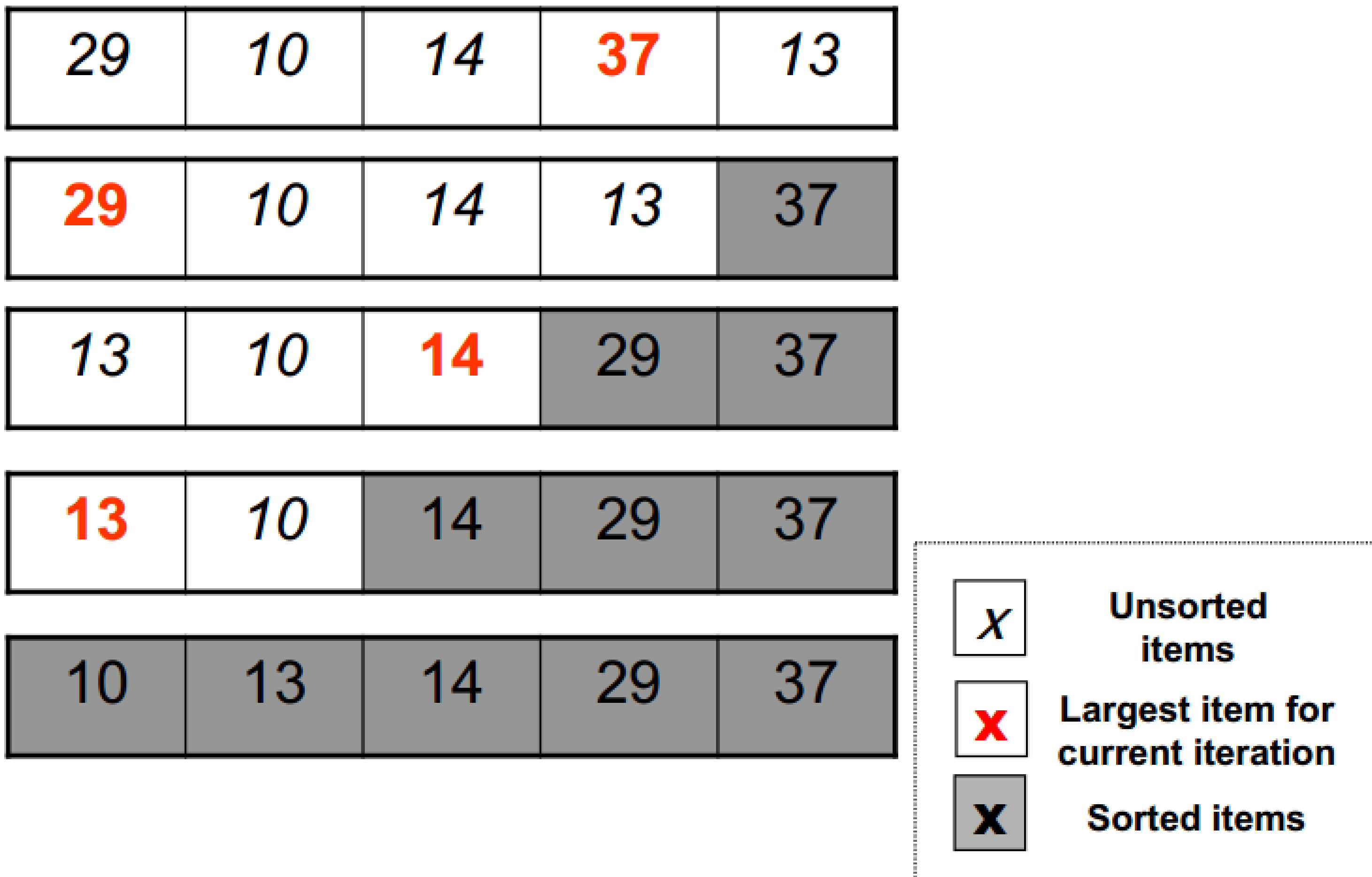
Simple Sorting Agorithms

Selection Sort

Selection Sort : Idea

- Given an array of n items, sort the items ascending
  1. Find the largest item M, in the range of [0…n-1]
  2. Swap M with the $(n-1)^{th}$ item
  3. Go to step 1, reduce n by 1

Selection Sort : Illustration



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;
       }

       int temp = a[maxIdx] ;
       a[maxIdx] = a[i] ;
       a[i] = temp ;
   }
}
```

Step 1.
Searching for
Maximum element

Step 2.
Swap maximum
element with the
last item

Trace the
execution here

29	10	14	37	13

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();
   }
}
```

Number of times executed
n-1
n-1
(n-1)+(n-2)+…+1 = n(n-1)/2
n-1
Total = c_1(n-1) + c_2*n*(n-1)/2 = $O(n^2)$

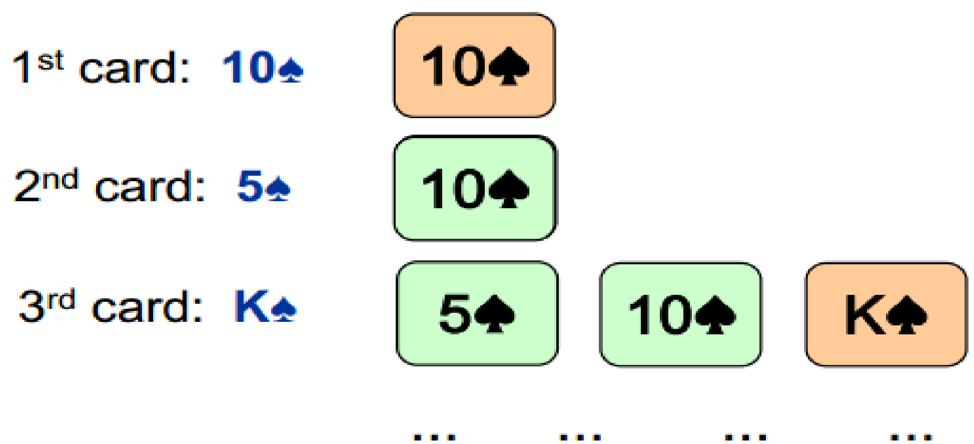
- c_1 and c_2 = cost of stmts in outer and inner block
- Data movement is minimal

Simple Sorting Agorithms

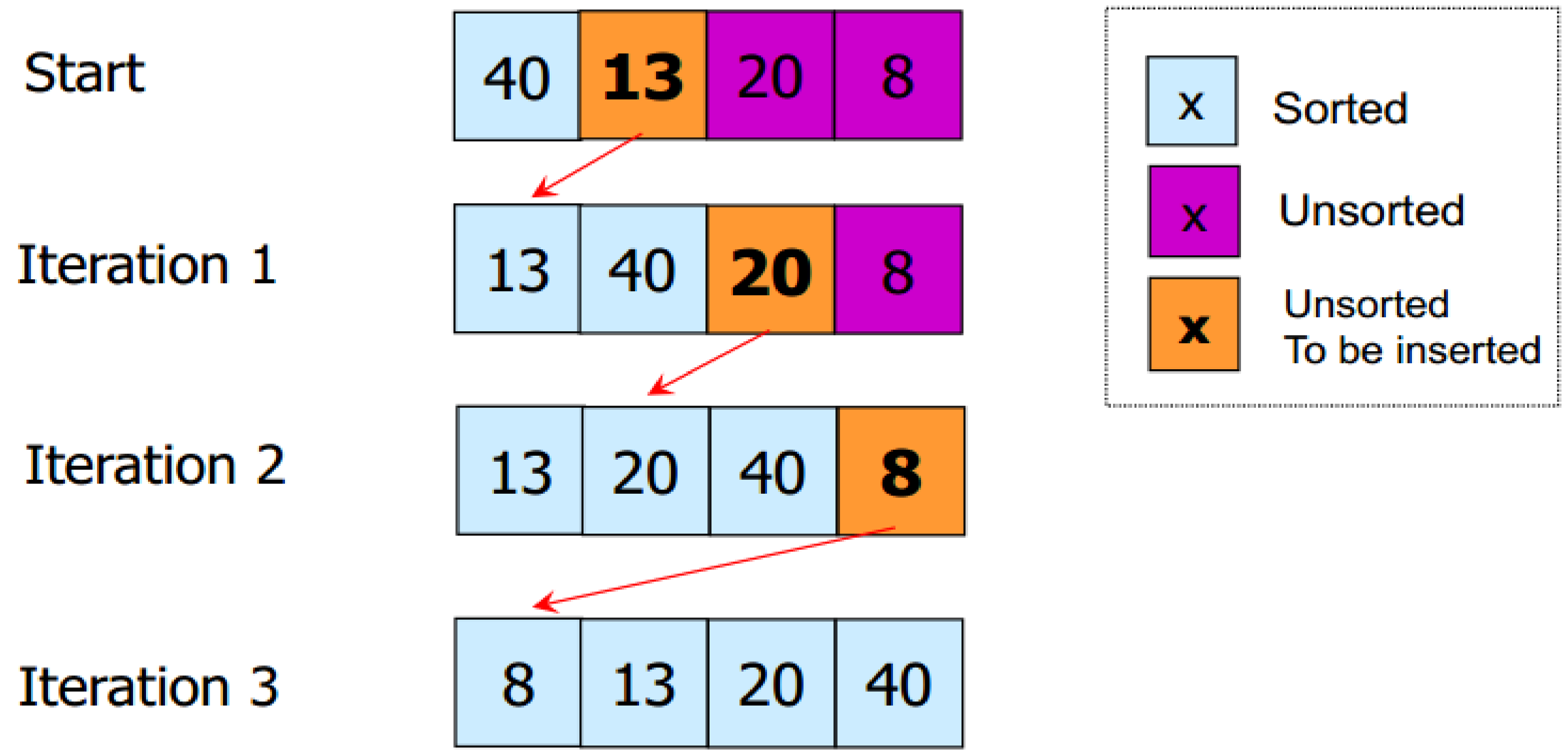
Insertion Sort

Insertion Sort : Idea

- Similar to how most people arrange a hand of poker cards:
  1. Start with one card in your hand
  2. Pick the next card and insert it into its proper sorted order.
  3. Repeat previous step for all N cards



Insertion Sort : Illustration



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 : the item to be inserted

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

Insert **next** to the correct location

29	10	14	37	13

Insertion Sort : Analysis

- 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)

Simple Sorting Agorithms

Shuffle Sort

Shuffle Sort : Idea

- Problem: Rearrange array so that result is a uniformly random permutation

$$2, 3, 4, 5, 6, 7, 8, 9, 10 \Rightarrow 8, 6, 9, 7, 2, 4, 10, 5, 3$$

- Idea:
  - Generate a random real number for each array entry, then sort the array (O(n^2))
  - For each array entry, swap it with an entry at random position (O(n))

Shuffle Sort : Implementation

```
//initialize an (sorted) array
for(int i = 0; i < array.length; i++)
     Swap( array,i,random.nextInt(10));
//the array is shuffled
```

Effective Sorting Algorithms

Quick Sort

Divide and Conquer

- 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.

Quick Sort : Idea

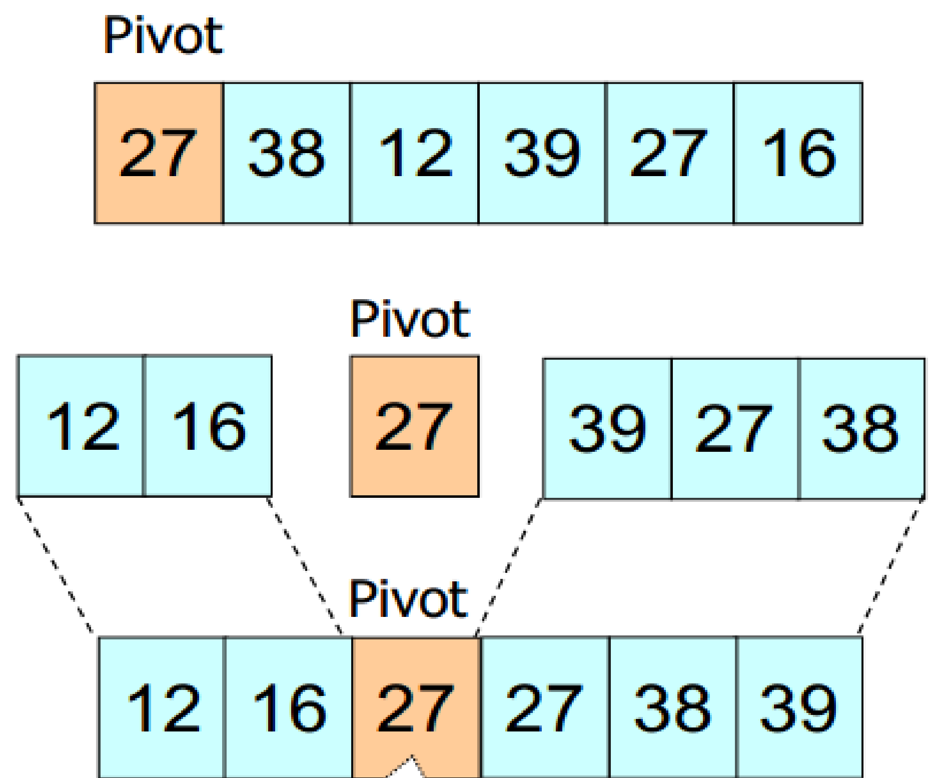
- Quick Sort is a divide-and-conquer algorithm
- Divide Step:
  - Choose an item p (known as pivot) and partition the items of a[i..j] into two parts:
    - Items that are smaller than p
    - Items that are greater than or equal to p
  - Recursively sort the two parts
- Conquer Step: Do nothing!
- Comparison:
  - Merge Sort spends most of the time in conquer step but very little time in divide step

Quick Sort : Divide Step Ilustration

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 : Code

```
void quickSort (int a[], int low, int high)
{
   int pivotIdx;

   if (low < high) {
      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() split `a[low…high]` into two portions

  `a[low … pivot – 1]` and `a[pivot + 1 … high]`
- Pivot item do not participate in any further sorting

Quick Sort : Code – Partition()

```
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, m);
      } else {

      }
    }
    swap (a, i, m);
    return m;
}
```

p is the pivot

S1 and **S2** empty initially

Go through each element in unknown region

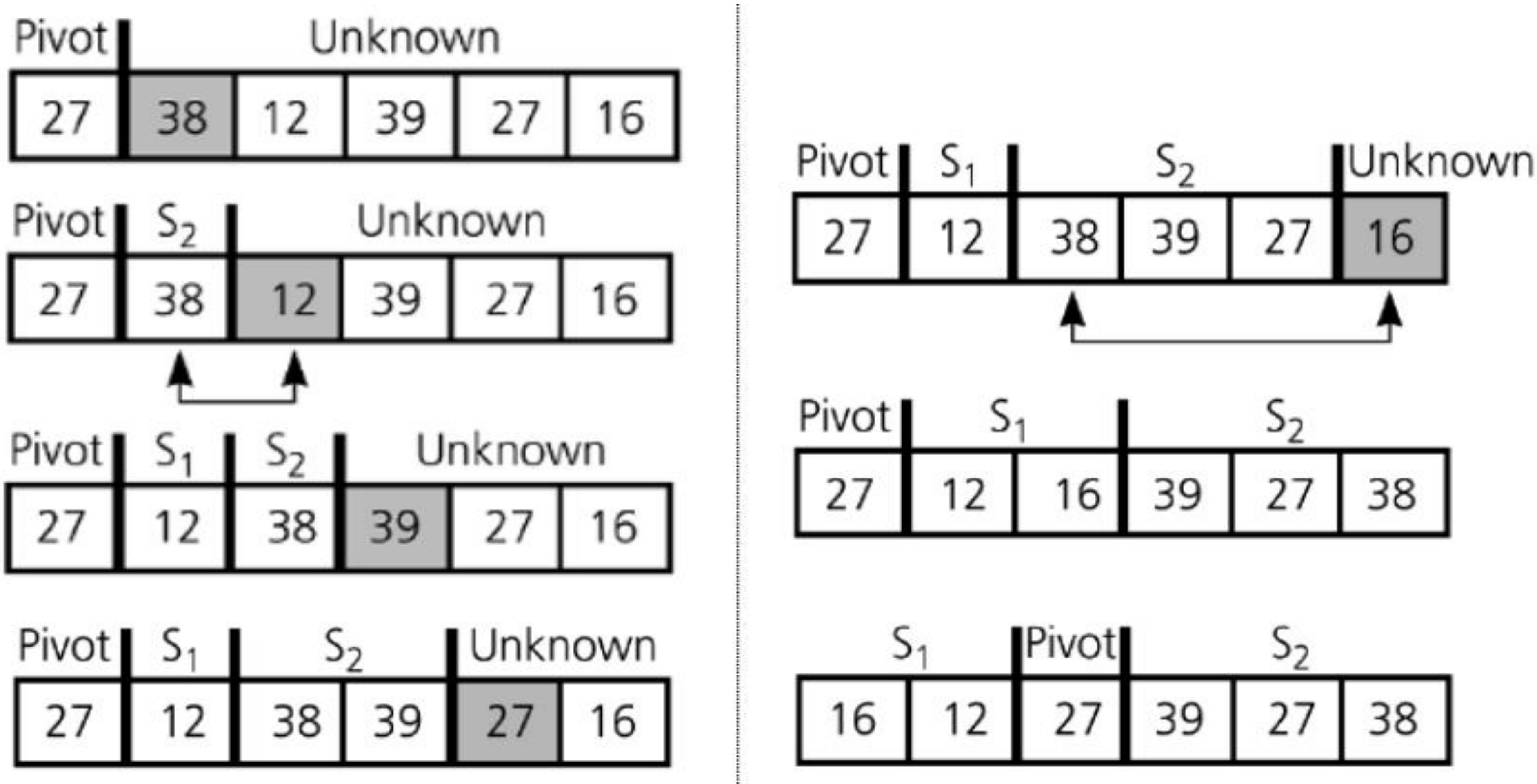
Case 2

Case 1 : do nothing!

Swap pivot with `a[m]`

m is the index of pivot

Quick Sort : Partition() Example



Quick Sort : Analysis

- Recusive algorithm complexity assessment – học sau.
- Total time complexity
  - Best case = O(*n* lg(*n*)).
  - Worst case = O(n^2);
  - Average case O(*n* lg(*n*))
- Optimal comparison based sort method.

Effective Sorting Algorithms

Merge Sort

Merge Sort : Idea

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

  {1,5,9} merge 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):
  1. Merge each pair of elements into sets of 2
  2. Merge each pair of sets of 2 into sets of 4
  3. Repeat previous step for sets of 4 …
  4. Final step Merges 2 sets of *n*/2 elements to obtain a sorted set.

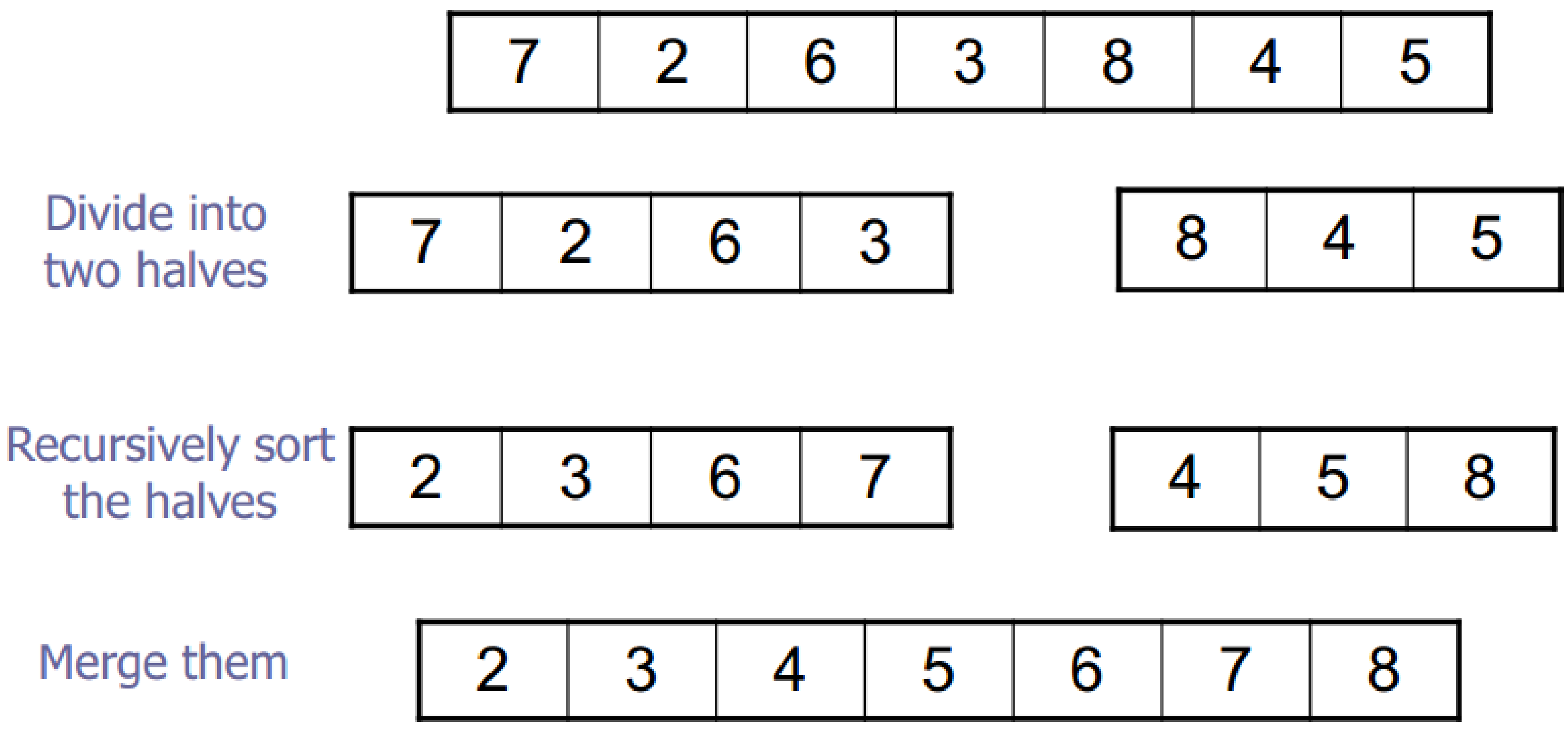
Merge Sort : Idea (cont.)

- 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.

Merge Sort : Idea (cont.)

- Merge Sort is 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



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

Merge Sort : MergeSort code

```
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]**

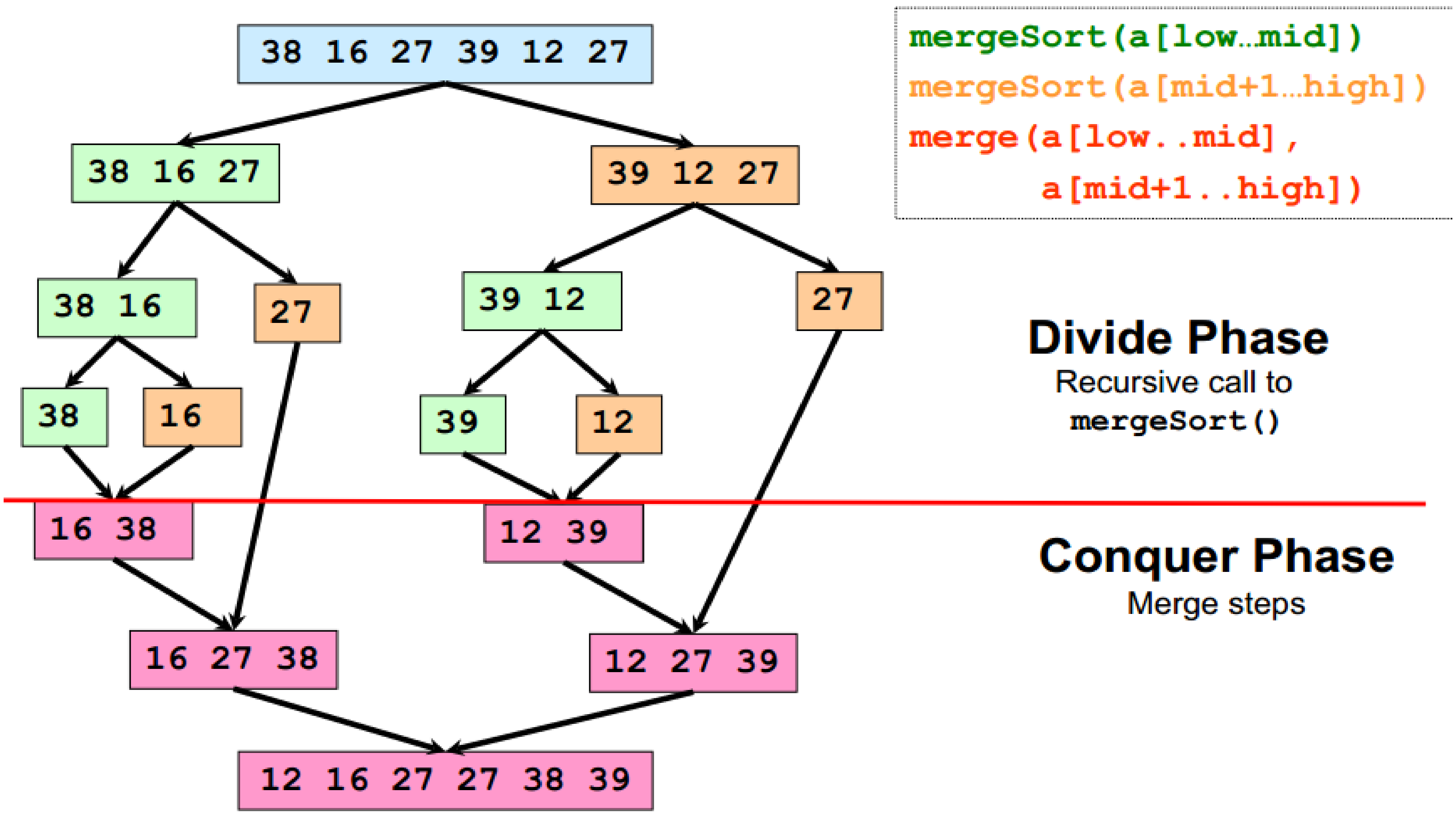
Divide a into two halves and **recursively** sort them

Conquer: merge the two sorted halves

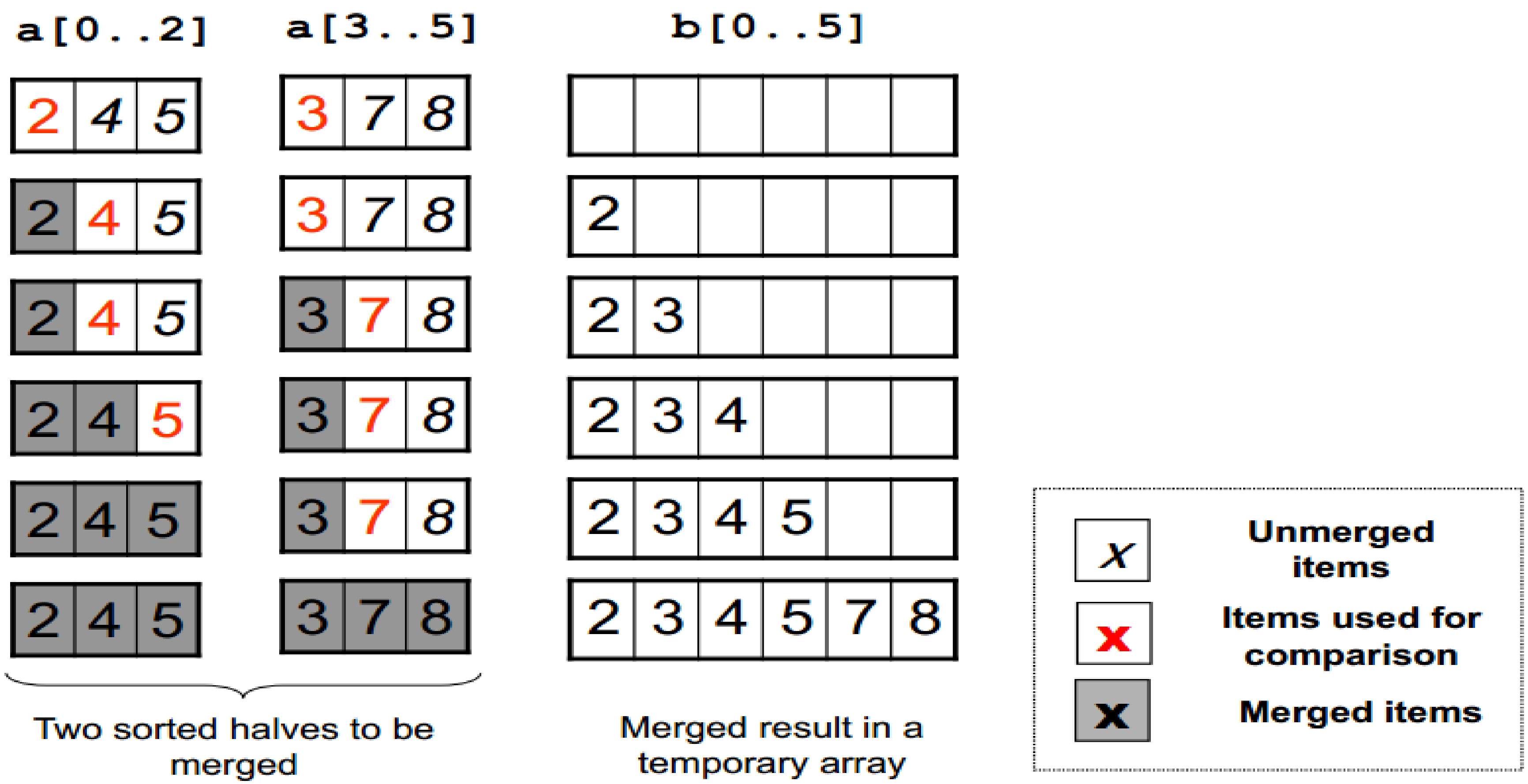
Function to merge **a[low…mid]** and **a[mid+1…high]** into **a[low…high]**

- Note:
  - mergeSort() is a recursive function
  - low >= high is the base case, i.e the array has 0 or 1 item

Merge Sort : An example



Merge Sort : An example



Merging Two Sorted Halves

Merge Sort : Merge code

```
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 code (cont.)

```
    // continue 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

- Recusive algorithm complexity assessment – later study.
- Total time complexity = O(n lg(n)).
- Optimal comparison based sort 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

Properties of Sorting

In-Place Sorting

Stable Sorting

In-Place Sorting

- 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
- Merge Sort is not in-place.
  - Because it need a temporary array for merging two sorted arrays.

Stable Sorting

- A sorting algorithm is stable if it does not reorder elements that are equal
- It is a useful property when:
  - The item contains a number of sort-able fields
  - We can then perform several sortings base on different field each time
- Example:
  - Student names have been sorted into alphabetical order
  - If it is sorted again according to tutorial group number:
    - A stable sorting algorithm will make all within the same group to appear in alphabetical order

Sorting Algorithms: Summary

Algorithm	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
Quick Sort	O(n^2)	O($n$log$n$)	Yes	No
Merge Sort	O($n$log$n$)	O($n$log$n$)	No	Yes

Exercises

- Implement simple sorting algorithms:
  - Bubble sort
  - Selection sort (two implementation)
  - Insertion sort
- Use generic data type for sorting arbitrary type
- Test the sorting algorithms on various data set (ints, doubles, strings, students, cards...)
- Homework: Hw2_SimpleSort.doc