Lecture 4b

Sorting

Bring Order to the World

Lecture Outline

- Iterative sorting algorithms (comparison based)
  - **Selection Sort**
  - **Bubble Sort**
  - **Insertion Sort**
- Properties of Sorting
  - **In-place sort**
  - **Stable sort**
  - Comparison of sorting algorithms
- Note: we only consider sorting data in ascending order in this lecture

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**
  - **Best / worst / average-case complexity**
  - **Recursive**
  - **Divide-and-conquer**
  - **Randomized algorithms**
  - **etc…**

Applications of Sorting

- Uniqueness testing
- Deleting duplicates
- Efficient searching
- Prioritizing events
- Frequency counting
- Reconstructing the original order
- Set intersection/union
- Finding a target pair x, y such that $x+y = z$
- *etc……*

Which one is the largest?

SELECTION SORT

Selection Sort : Idea

- **Observations:**
  - It is simple to find the ***largest item*** in a list of unsorted numbers
  - We know the ***correct position*** of the largest item once it is found
  - Once the largest item is in the correct position, it can be "***ignored***"
- Try applying the idea:

A	0	1	2	3	4	5	6	7	8	9	10
	8	3	-5	4	2	-10	13	2	-7	9	0

Selection Sort: **Attempt One**

- Use a temporary array to store the result?

A	29	10	14	**37**	13
Result					**37**

- Difficulty and drawbacks:
  - Need to "remove" the max number from the original array every round
  - Need to keep track of the current position in the result[] array
  - **Wasteful**: Need an additional size N array for result

Selection Sort: **Attempt Two**

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**

x **Unsorted items**

x **Largest item for current iteration**

x **Sorted items**

Sorted!

Selection Sort: **Implementation**

```
void selectionSort(int a[], int n) {
    int i, j, maxIdx, temp;

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

Step 1: Search for maximum element

Step 2: Swap maximum element with the last item

Swap Elements: Helper Function

```
void swap(int a[], int i, int j) {
      int temp;

      temp = a[i] ;
      a[i] = a[j] ;
      a[j] = temp ;
}
```

- Simple function to swap two elements [i] and [j] in the array a[]
- Useful for other sorting algorithms

Selection Sort: **Analysis**

```
void selectionSort(int a[], int n) {
    int i, j, maxIdx, temp;

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

$$(n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2}$$

$$(n-1)$$

- We can focus on the number of comparison and swapping for sorting algorithm (why?)
- Selection Sort is a **O(N^2)** algorithm

Bubbly numbers

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**

(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**

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

(b) Pass 2

10	14	29	13	**37**
10	14	29	13	**37**
10	14	29	13	**37**
10	14	13	**29**	**37**

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

x	**Sorted Item**
x	**Pair of items under comparison**

Bubble Sort: Implementation

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

Step 1: Compare adjacent pairs of numbers

Step 2: Swap if the items are out of order

Bubble Sort: **Analysis**

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

$$(n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2}$$

- O(N²) comparisons
- O(N²) swaps in the worst case
- Bubble sort is a O(N²) algorithm

Bubble Sort: Early Termination

- Bubble Sort is inefficient, but 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) {
    int i, j;
    bool sorted;
    for (i = n-1; i >= 1; i--) {
        sorted = true;
        for (j = 1; j <= i; j++) {
            if (a[j-1] > a[j]) {
                swap(a, j, j-1);
                sorted = false;
            }
        } //end of inner loop
        if (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: $O(n)$

Let's cut queue!

INSERTION SORT

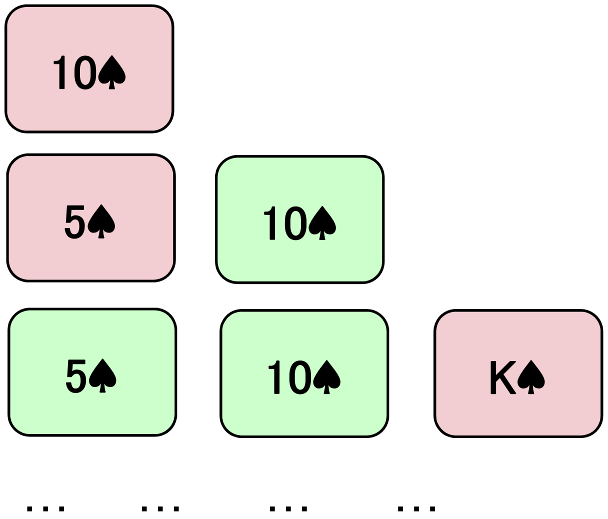
Insertion Sort: **Idea**

- 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 cards

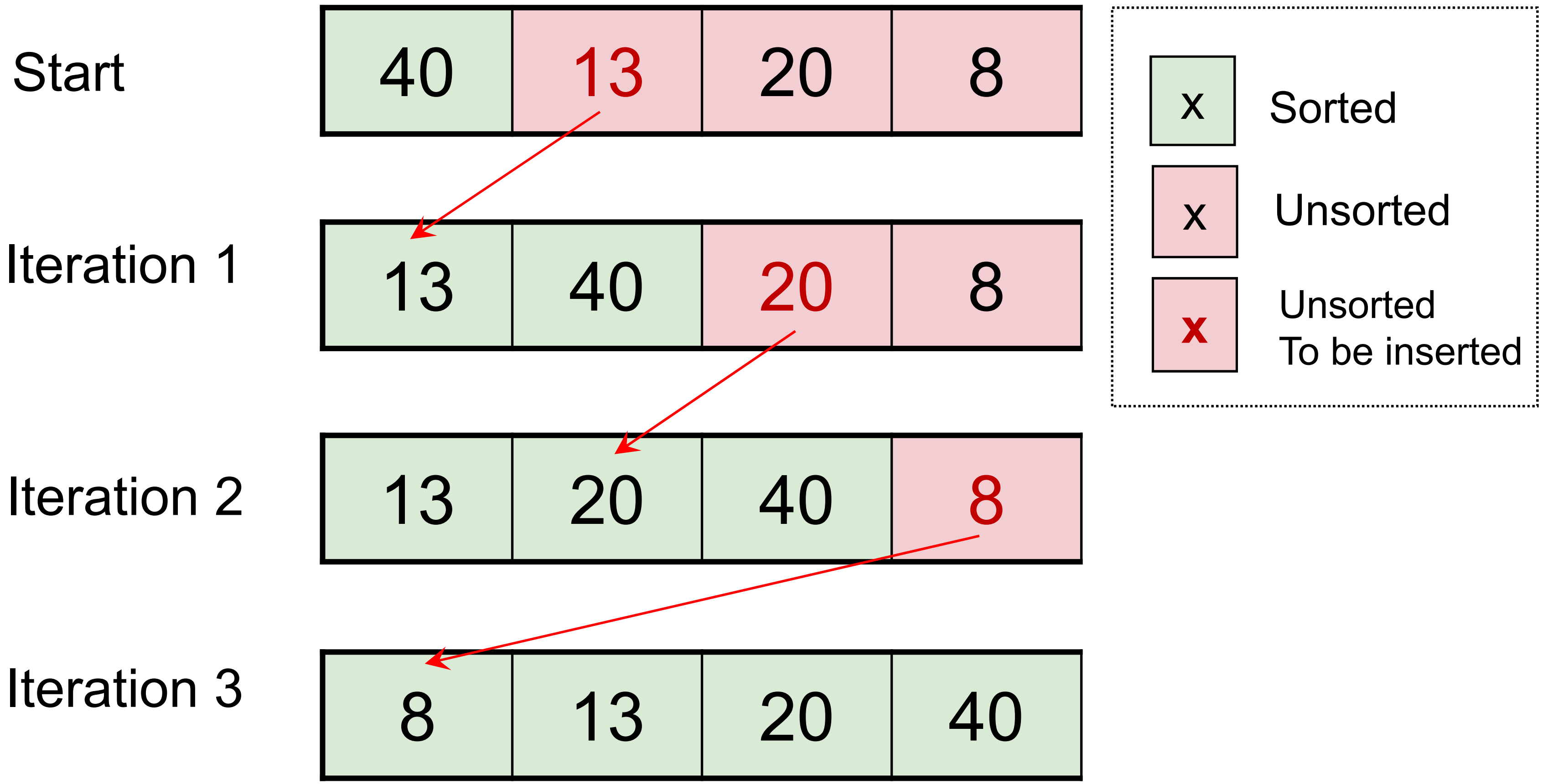
1st card: **10♠**

2nd card: **5♠**

3rd card: **K♠**



Insertion Sort: **Illustration**



Insertion Sort: **Implementation**

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

          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

- Read the condition of the inner loop carefully
  - The order of the two conditions cannot be changed!

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)**

As long as it can sort, it is a good sort…..?

PROPERTIES OF SORTING

Property: 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
  - Are the 3 sorting algorithms we covered in place?

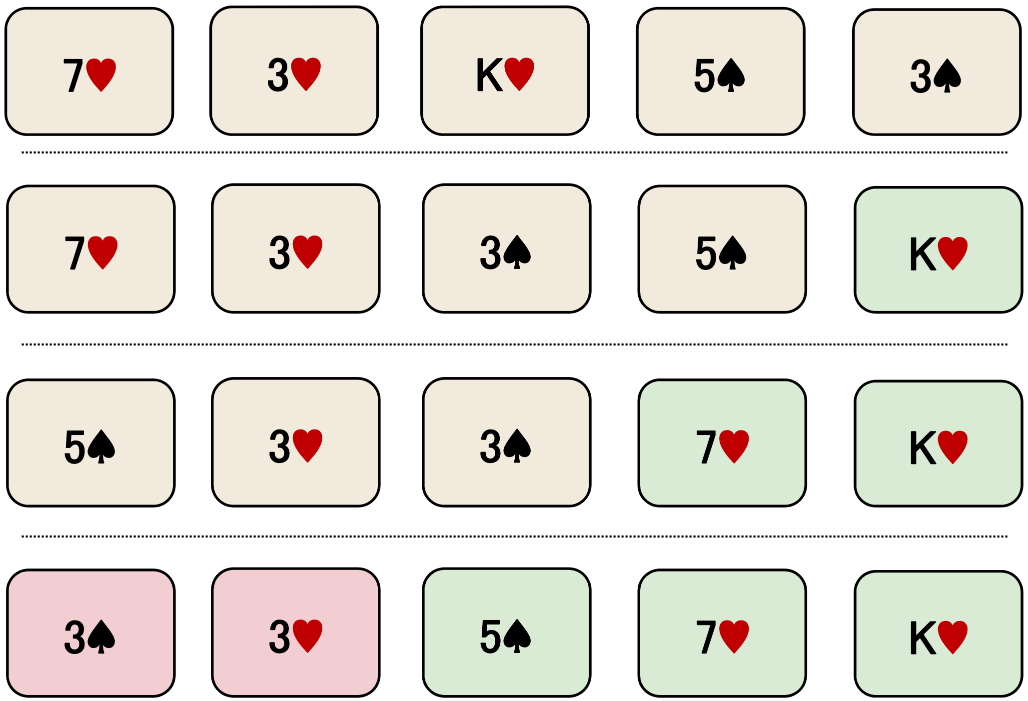
Property: Stable Sorting

- 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**
  1. Assume that **names** have been sorted in alphabetical order
  2. 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

Counter-Example: **Non-Stable Sort**

- **Selection Sort**



Originally sorted by suit
♥ < ♠

Stable sorting should result in
3♥ < 3♠

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

Can sorting be faster?

BEYOND O(N^2)
[FOR YOUR INTEREST ONLY]

Beyond $O(N^2)$

- There are many more interesting sorting algorithms

- Comparison based algorithms:
  - Mergesort, Quicksort, etc
  - Can achieve $O(N \text{ Log}_2 N)$

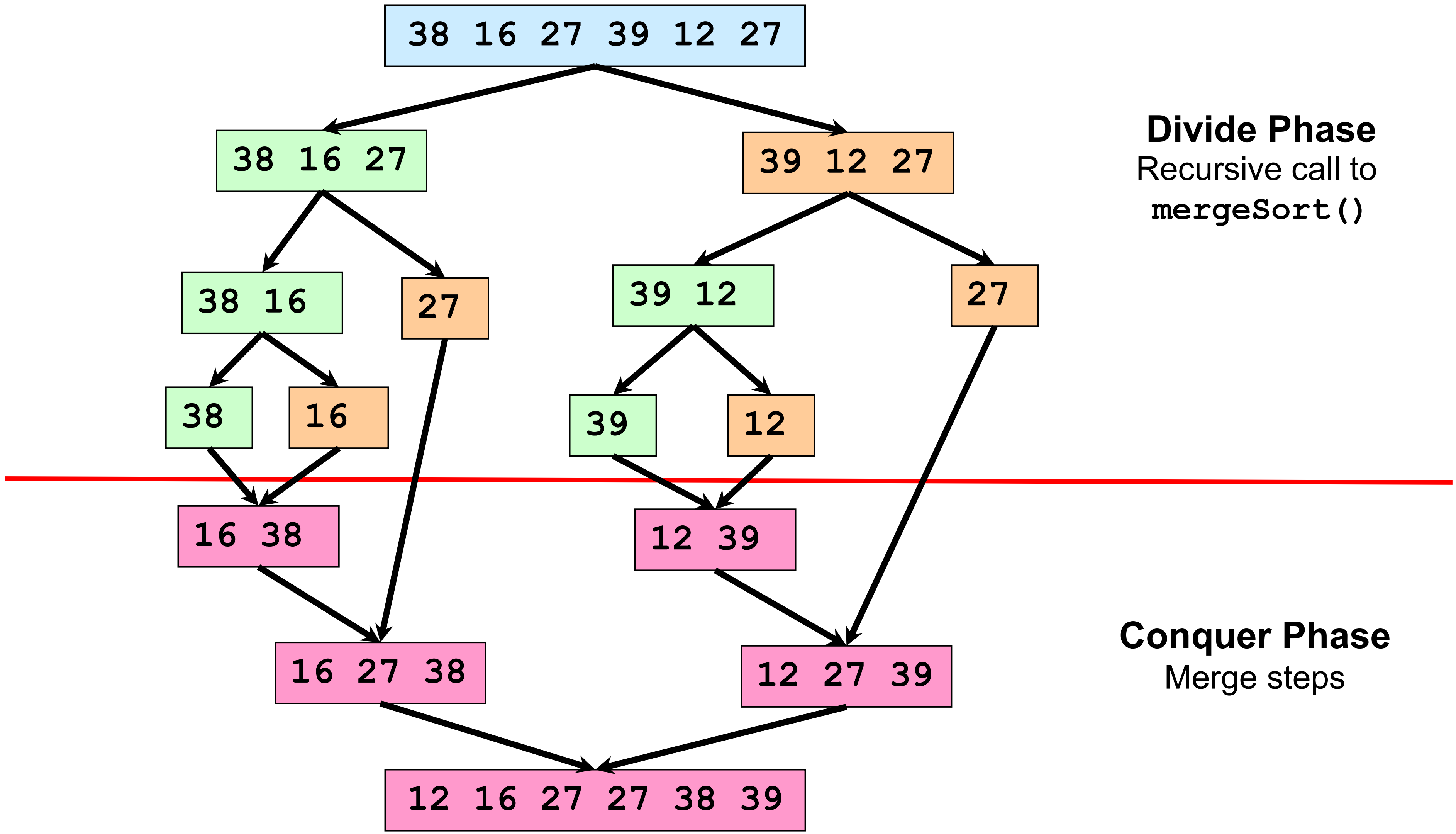
- Non-Comparison based algorithms
  - Radix sort, bucket sort, counting sort etc
  - Complexity can be better than $O(N^2)$

- Not practical but interesting algorithms
  - Sleep sort, stupid sort (gnome sort), bogosort, etc

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

Merge Sort: Example



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
  1. Organize the data into groups according to the next character in each data
  2. 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)

Sleep Sort: Idea

- Ingredients:
  1. We can create new process (running program)
  2. Each process can **sleep(N)** to pause for **N** seconds

- So, given a set of inputs a[0] to a[n-1]
  1. Create a new process and pass a value **a[i]** to it
  2. New process sleeps for **a[i]** seconds, then print out **a[i]**

- Example: a = { 7, 1, 3}
  - P_0, P_1 and P_2 sleeps for 7, 1 and 3 seconds respectively
  - P_1 wake up first and print "1", followed by P_2, then P_0

Summary

- Comparison-Based Sorting Algorithms
  - Iterative Sorting
    - Selection Sort
    - Bubble Sort
    - Insertion Sort

- Properties of Sorting Algorithms
  - In-Place
  - Stable

- Advanced / Additional algorithms for interest