# ECE2810J
## Data Structures and Algorithms

## Comparison Sort

**Learning Objectives:**

- Know the difference between comparison sort and non-comparison sort

- Know the procedures of merge sort and quick sort

- Know the master theorem

- Know different characteristics of sorting algorithms, such as time complexity, stability, etc.

# Outline

- Sorting Basics

- Merge Sort

- Quick Sort

- Comparison Sort Summary

# Sorting

- Given array A of size N, reorder A so that its elements are in order.
  - "In order" with respect to a consistent comparison function, such as "$\leq$" or "$\geq$".

- Sorting order
  - Ascending order
  - Descending order
- Unless otherwise specified, we consider sorting in ascending order.

# Characteristics of Sorting Algorithms

- Average-case time complexity
- Worst-case time complexity
- Space usage: **in place** or not?
  - **in place**: requires $O(1)$ additional memory
  - Don't forget the stack space used in recursive calls
  - **In place is better**
    - Why? The data can fit into cache, not main memory
  - Real example: quick sort versus merge sort. Both have average-case time complexity of $O(n \log n)$. Quick sort is faster, due to in place

# Characteristics of Sorting Algorithms

- **Stability**: whether the algorithm maintains the relative order of records with equal keys

(4, b), (3, e), (3, b), (5, b) ➡ (3, e), (3, b), (4, b), (5, b)

Sort on the first number          **Stable!**

- Usually there is a secondary key whose ordering you want to keep. Stable sort is thus useful for sorting over multiple keys

- Example: sort complex numbers a+bi

  - Ordering rule: first compare a; when there is a tie, compare b

  - One sorting method: first sort b, then sort a

3+5i, 2+6i, 3+4i, 5+2i    Sort on b ➡ 5+2i, 3+4i, 3+5i, 2+6i

… sort on a ➡ 2+6i, 3+4i, 3+5i, 5+2i    Stability is important!

# Types of Sorting Algorithms

- Sorting algorithms can be classified as **comparison sort** and **non-comparison sort**.

- **Comparison sort**: each item is compared against others to determine its order.

- **Non-comparison sort**: each item is put into predefined "bins" independent of the other items presented.
    - No comparison with other items needed.
    - It is also known as **distribution-based sort**.

# Types of Sorting Algorithms

- General types of comparison sort
  - Insertion-based: insertion sort
  - Selection-based: selection sort, heap sort
  - Exchange-based: bubble sort, quick sort
  - Merging-based: merge sort

- Non-comparison sort:
  counting sort, bucket sort, radix sort

# Insertion Sort

- `A[0]` alone is a sorted array.
- For `i=1` to `N-1`
  - **Insert** `A[i]` into the appropriate location in the sorted array `A[0], …, A[i-1]`, so that `A[0], …, A[i]` is sorted.
  - To do so, save `A[i]` in a temporary variable `t`, shift sorted elements greater than `t` right, and then insert `t` in the gap.
- Worst-case time complexity? $O(N^2)$
- In place? Yes. O(1) additional memory.
- Stable?
  - Yes, because elements are visited in order and equal elements are inserted after its equals.

# Insertion Sort
Best Case Time Complexity

- For `i=1` to `N-1`
  - **Insert** `A[i]` into the appropriate location in the sorted array `A[0], …, A[i-1]`, so that `A[0], …, A[i]` is sorted.

- The **best case** time complexity is $O(N)$.
  - It happens when the array is already sorted.
  - For other sorting algorithms we will talk, their best case time complexity is $\Omega(N \log N)$.

9

# Selection Sort

- For **i=0** to **N-2**
    - Find the smallest item in the array **A[i], ..., A[N-1]**. Then, swap that item with **A[i]**.

- Finding the smallest item requires **linear scan**.

# ? Which Statements Are Correct for Selection Sort?

For **`i=0`** to **`N-2`**

Find the smallest item in the array **`A[i]`, ..., `A[N-1]`**. Then, swap that item with **`A[i]`**.

- **A.** Its worse-case time complexity is $O(N^2)$
- **B.** Its best-case time complexity is $\Omega(N^2)$
- **C.** It is not in-place
- **D.** It is stable

# Bubble Sort

```
For i=N-2 downto 0
  For j=0 to i
    If A[j]>A[j+1] swap A[j] and A[j+1]
```

- Compares two adjacent items and swap them to keep them in ascending order.
  - From the beginning to the end. The last item will be the largest.

- Worst-case time complexity? $O(N^2)$

- In place?  Yes.

- Stable?
  - Yes, because equal elements will not be swapped.

# Two Problems with Simple Sorts

- They learn only one piece of information per comparison and hence might compare every pair of elements.
  - Contrast with binary search: learns N/2 pieces of information with first comparison.
- They often move elements one place at a time (bubble sort and insertion sort), even if the element is "far" from its **final place**.
  - Contrast with selection sort, which moves each element exactly to its final place.

- Fast sorts attack these two problems.
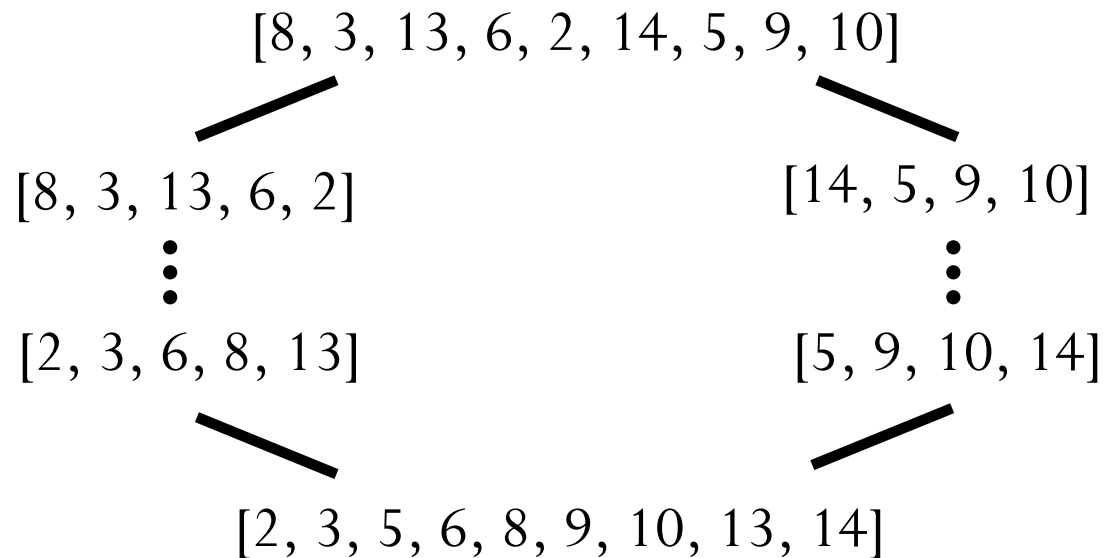  - Two famous ones: **merge sort** and **quick sort**.

13

# Outline

- Sorting Basics

- **Merge Sort**

- Quick Sort

- Comparison Sort Summary

# Merge Sort
## Algorithm

- Spilt array into two (roughly) equal subarrays.
- <u>Merge sort</u> each subarray recursively.
  - The two subarrays will be sorted.
- Merge the two sorted subarrays into a sorted array.

[8, 3, 13, 6, 2, 14, 5, 9, 10]

[8, 3, 13, 6, 2]      [14, 5, 9, 10]

⋮      ⋮

[2, 3, 6, 8, 13]      [5, 9, 10, 14]

[2, 3, 5, 6, 8, 9, 10, 13, 14]

# Merge Sort
Pseudo-code

```
void mergesort(int *a, int left, int
  right) {
    if (left >= right) return;
    int mid = (left+right)/2;
    mergesort(a, left, mid);
    mergesort(a, mid+1, right);
    merge(a, left, mid, right);
}
```

# Merge Two Sorted Arrays

- For example, merge A = (2, 5, 6) and B = (1, 3, 8, 9, 10).

- Compare the smallest element in the two arrays A and B and move the smaller one to an additional array C.

- Repeat until one of the arrays becomes empty.

- Then append the other array at the end of array C.

# Merge Two Sorted Arrays
## Implementation

- We actually do not "remove" element from arrays A and B.
  - We just keep a pointer indicating the smallest element in each array.
  - We "remove" element by incrementing that pointer.

```
i = j = k = 0;
while(i < sizeA && j < sizeB) {
  if(A[i]<=B[j]) C[k++]=A[i++];
  else C[k++]=B[j++];
}
if(i == sizeA) append(C,B,j);
else append(C,A,i);
```

Time complexity?

Time complexity is $O(sizeA + sizeB)$

# Merge Sort
Time Complexity

```
void mergesort(int *a, int left, int
  right) {
    if (left >= right) return;
    int mid = (left+right)/2;
    mergesort(a, left, mid);        T(N/2)
    mergesort(a, mid+1, right);     T(N/2)
    merge(a, left, mid, right);     O(N)
}
```

- Let $T(N)$ be the time required to merge sort $N$ elements.

- Merge two sorted arrays with total size $N$ takes $O(N)$.

Recursive relation: $T(N) = 2T(N/2) + O(N)$

How to solve the recurrence?

# Solve Recurrence: Master Method

- A "black box" for solving recurrence.

- However, there is an important assumption: all sub-problems have roughly **equal** sizes.

  - E.g., merge sort
  - Not applicable to unbalanced division.

# Solve Recurrence: Master Method

- Recurrence: $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$
  - Base case: $T(n) \leq constant$ for all sufficiently small n.
  - $a$ = number of recursive calls (integer $\geq 1$)
  - $b$ = input size shrinkage factor (integer $> 1$)
  - $O(n^d)$: the runtime of merging solutions. $d$ is real value $\geq 0$.
  - a, b, d are independent of n.
- <u>Claim</u>:

base doesn't matter

$$T(n) = \begin{cases} O(n^d \log n) & if \ a = b^d \\ O(n^d) & if \ a < b^d \\ O(n^{\log_b a}) & if \ a > b^d \end{cases}$$

base matters!

# Example of Merge Sort

Recurrence:   $T(n) \leq aT\left(\dfrac{n}{b}\right) + O(n^d)$

Claim:   $T(n) = \begin{cases} O\left(n^d \log n\right) & if\ a = b^d \\ O\left(n^d\right) & if\ a < b^d \\ O\left(n^{\log_b a}\right) & if\ a > b^d \end{cases}$

- $a = 2, b = 2, d = 1 \ \Rightarrow b^d = a$
- $T(n) = O(n \log n)$

# What are $a, b, d$ for Binary Search?

Recurrence: $\quad T(n) \leq aT\left(\dfrac{n}{b}\right) + O(n^d)$

Claim: $\quad T(n) = \begin{cases} O\left(n^d \log n\right) & if \ a = b^d \\ O\left(n^d\right) & if \ a < b^d \\ O\left(n^{\log_b a}\right) & if \ a > b^d \end{cases}$

**A.** a $=2$, b $= 2$, d $= 0$  **B.** a $=1$, b $= 2$, d $=0$

**C.** a $=2$, b $= 2$, d $= 1$  **D.** a $=1$, b $= 2$, d $=1$

# Merge Sort
## Characteristics

- Not in-place
  - For efficient merging two sorted arrays, we need an auxiliary $O(N)$ space.
  - Recursion needs up to $O(\log N)$ stack space.

- Stable if **`merge()`** <span style="color:red">**maintains**</span> the relative order of equal keys.

# Divide-and-Conquer Approach

- Merge sort uses the **divide-and-conquer** approach.

- Recursively **breaking** down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly.
  - For merge sort, split an array into two and sort them respectively.

- The solutions to the sub-problems are then **combined** to give a solution to the original problem.
  - For merge sort, merge two sorted arrays.

25

# Outline

- Sorting Basics

- Merge Sort

- **Quick Sort**

- Comparison Sort Summary

# Quick Sort
*Algorithm*

Another divide-and-conquer approach to sort

- Choose an array element as **pivot**.
- Put all elements < pivot to the left of pivot.
- Put all elements ≥ pivot to the right of pivot.
- Move pivot to its correct place in the array.

} **partition()**

- Sort left and right subarrays recursively (not including pivot).

```
void quicksort(int *a, int left,
  int right) {
    int pivotat; // index of the pivot
    if(left >= right) return;
    pivotat = partition(a, left, right);
    quicksort(a, left, pivotat-1);
    quicksort(a, pivotat+1, right);
}
```

# Choice of Pivot

- If your input is random, you can choose the **first** element.
  - But this is very bad for presorted input.

- A better strategy: **randomly** pick an element from the array as pivot.
  - <u>Claim</u>: **for any input**, the average running time is $O(n \log n)$.
    - <u>Note</u>: average is over random choice of pivots made by the algorithm, **not** on the input.

# Partitioning the Array

- Once pivot is chosen, swap pivot to the beginning of the array.
- When another array B is available, scan original array A from left to right.
  - Put elements < pivot at the left end of B.
  - Put elements ≥ pivot at the right end of B.
  - The pivot is put at the remaining position of B.
  - Copy B back to A.

A | 6 | 2 | 8 | 5 | 11 | 10 | 4 | 1 | 9 | 7 | 3 |

B | 2 | 5 | 4 | 1 | 3 | 6 | 7 | 9 | 10 | 11 | 8 |

# In-Place Partitioning the Array

1. Once pivot is chosen, swap pivot to the beginning of the array.

2. Start counters `i=1` and `j=N-1`.

3. Increment `i` until we find element `A[i]>=pivot`.
   - `A[i]` is the leftmost item $\geq$ pivot.

4. Decrement `j` until we find element `A[j]<pivot`.
   - `A[j]` is the rightmost item $<$ pivot.

5. If `i<j`, swap `A[i]` with `A[j]`. Go back to step 3.

6. Otherwise, swap the first element (pivot) with `A[j]`.

# In-Place Partitioning the Array
## Example

i [yellow]          j [blue]

A | 6 | 2 | **8** | 5 | 11 | 10 | 4 | 1 | 9 | 7 | **3** |

A | 6 | 2 | 3 | 5 | **11** | 10 | 4 | **1** | 9 | 7 | 8 |

A | 6 | 2 | 3 | 5 | 1 | **10** | **4** | 11 | 9 | 7 | 8 |

A | 6 | 2 | 3 | 5 | 1 | **4** | **10** | 11 | 9 | 7 | 8 |

- Now, `j < i`, swap the first element (pivot) with `A[j]`.

A | **4** | 2 | 3 | 5 | 1 | 6 | 10 | 11 | 9 | 7 | 8 |

# In-Place Partitioning the Array
## Time Complexity

1. Once pivot is chosen, swap pivot to the beginning of the array.

2. Start counters **i=1** and **j=N-1**.

3. Increment **i** until we find element **A[i]>=pivot**.

4. Decrement **j** until we find element **A[j]<pivot**.

5. If **i<j**, swap **A[i]** with **A[j]**. Go back to step 3.

6. Otherwise, swap the first element (pivot) with **A[j]**.

- Scan the entire array no more than twice.
- Time complexity is $O(N)$, where $N$ is the size of the array.

# Quick Sort
## Time Complexity

```
void quicksort(int *a, int left,
  int right) {
    int pivotat; // index of the pivot
    if(left >= right) return;
    pivotat = partition(a, left, right);
    quicksort(a, left, pivotat-1);
    quicksort(a, pivotat+1, right);
}
```

$O(N)$

$T(LeftSz)$

$T(RightSz)$

- Let $T(N)$ be the time needed to sort $N$ elements.
  - $T(0) = c$, where $c$ is a constant.
- Recursive relation:

$$T(N) = T(LeftSz) + T(RightSz) + O(N)$$

  - $LeftSz + RightSz = N - 1$

# Quick Sort
## Worst Case Time Complexity

- Recursive relation:

$$T(N) = T(LeftSz) + T(RightSz) + O(N)$$

- Worst case happens when each time the pivot is the smallest item or the largest item

  - $T(N) = T(N-1) + T(0) + O(N)$

    $\leq T(N-1) + T(0) + dN$

    $\leq T(N-2) + 2T(0) + d(N-1) + dN$

    $\cdots$

    $\leq T(0) + NT(0) + d + 2d + \cdots + d(N-1) + dN$

    $= O(N^2)$

# Quick Sort
Best Case Time Complexity

- Recursive realtaion:
$$T(N) = T(LeftSz) + T(RightSz) + O(N)$$

- Best case happens when each time the pivot divides the array into two equal-sized ones.
  - $T(N) = T((N-1)/2) + T((N-1)/2) + O(N)$
  - The recursive relation is similar to that of merge sort.
  - $T(N) = O(N \log N)$

# Quick Sort
## Average Time Complexity

- Average time complexity of quick sort can be proved to be $O(N \log N)$.
  - Assume **randomly** pick an element from the array as pivot.
  - **Note**: average is over random choice of pivots made by the algorithm, **not** on the input.
  - The claim holds for any input.

# Quick Sort
Other Characteristics

- In-place?
  - In-place partitioning.
  - Worst case needs $O(N)$ stack space.
  - Average case needs $O(\log N)$ stack space.
    - "Weakly" in-place.

- Not stable.

# Quick Sort
Summary

- Like merge sort, quick sort is a divide-and-conquer algorithm.

- Merge sort: easy division, complex combination.
- Quick sort: complex division (partition with pivot step), easy combination.

- Insertion sort is faster than quick sort for small arrays.
  - Terminate quick sort when array size is below a threshold. Do insertion sort on subarrays.

# Outline

- Sorting Basics

- Merge Sort

- Quick Sort

- **Comparison Sort Summary**

# Comparison Sorts
Summary

| | Worst Case Time | Average Case Time | In Place | Stable |
|---|---|---|---|---|
| Insertion | $O(N^2)$ | $O(N^2)$ | Yes | Yes |
| Selection | $O(N^2)$ | $O(N^2)$ | Yes | No |
| Bubble | $O(N^2)$ | $O(N^2)$ | Yes | Yes |
| Merge Sort | $O(N \log N)$ | $O(N \log N)$ | No | Yes |
| Quick Sort | $O(N^2)$ | $O(N \log N)$ | Weakly | No |

For comparison sort, is $O(N \log N)$ the best we can do in the **worst case**?

# Comparison Sorts
## Worst Case Time Complexity

- Theorem: A sorting algorithm that is based on pairwise comparisons must use $\Omega(N \log N)$ operations to sort in the worst case.