# VE281

## Data Structures and Algorithms

Sorting

# Announcement

- Written homework four will be put online by this Friday.

- Due time: 11:40 am, Dec. 18th.

# Review

- Prim's Algorithm for MST
- Kruskal's Algorithm for MST
  - Repeatedly add the edge with the **smallest weight** that **does not cause a cycle** until no such edges exist.
- Sorting
  - Characteristics: time complexity, in-place, stable.
  - Basic types of sorting algorithms.
  - Insertion sort and selection sort.

# Outline

- Simple Sorting Algorithms and Heap Sort
- Merge Sort
- Quick Sort

# 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.

$$O(N^2)$$

- Time complexity?
- 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.

# Heap Sort

- Store items in a min heap. $O(N)$
- Call **`dequeueMin`** N times to extract the items in ascending order. $O(N \log N)$
- Heap sort is a type of selection sort.
- Time complexity?
  - $O(N \log N)$
- In place?
  - Yes.
- Stable?
  - No.

# Heap Sort Stability

- Input array: (3, e), (2, a), (3, b)
- After initializing the min heap: (2, a), (3, e), (3, b)
- First **dequeueMin** output: (2, a)
  - The remaining min heap: (3, b), (3, e).
- Second **dequeueMin** output: (3, b)
  - The remaining min heap: (3, e).
- Third **dequeueMin** output: (3, e)

Unstable!

- The sorted array: (2, a), (3, b), (3, e).

# Outline

- Simple Sorting Algorithms and Heap Sort
- Merge Sort
- Quick Sort

# Merge Sort

## Algorithm

- Spilt array into two roughly equal subarrays.
- Merge sort each subarray recursively.
  - The two subarrays will be sorted.
- Merge the two sorted subarrays into a sorted array.

```
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

- 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
## Example

- Merge A = (2, 5, 6) and B = (1, 3, 8, 9, 10).
- Repeatedly compare the smallest element in the two arrays A and B and move the smaller one to an additional array C:

  A = (2, 5, 6), B = (1, 3, 8, 9, 10), C = ()

  A = (2, 5, 6), B = (3, 8, 9, 10), C = (1)

  A = (5, 6), B = (3, 8, 9, 10), C = (1, 2)

  A = (5, 6), B = (8, 9, 10), C = (1, 2, 3)

  A = (6), B = (8, 9, 10), C = (1, 2, 3, 5)

  A = (), B = (8, 9, 10), C = (1, 2, 3, 5, 6)

  Sorted!
- Array A becomes empty, append B at the end

# Merge Two Sorted Arrays

## Implementation

- We actually do not “remove” element from arrays A and B.
- We 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) app
else append(C, A)
```

Time complexity is $\Theta(sizeA + sizeB)$

# Merge Sort

## Example

split

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

[8, 3, 13, 6, 2]

[14, 5, 9, 10]

[8, 3, 13]

[6, 2]

[14, 5]

[9, 10]

[8, 3]

[13]

[6]

[2]

[14]

[5]

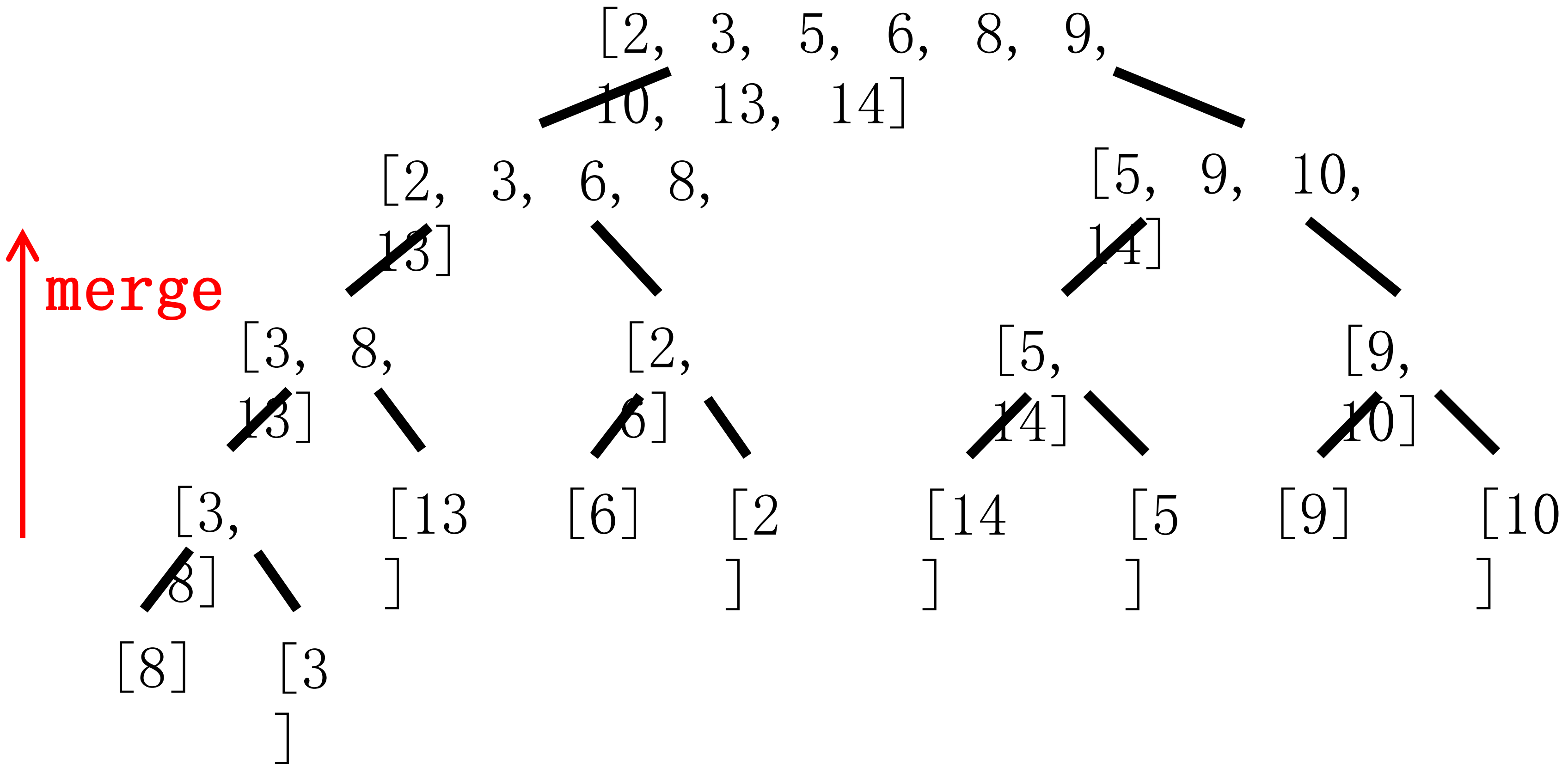
[9]

[10]

[8]

[3]

# Merge Sort
## Example



# 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);
    mergesort(a, mid+1, right);
    merge(a, left, mid, right);
}
```

$T(N/2)$

$T(N/2)$

$\Theta(N)$

- Let $T(N)$ be the time required to merge sort $N$ elements.
- Merge two sorted arrays with total size $N$ takes $\Theta(N)$.

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

# Merge Sort

## Time Complexity

- $\boxed{\text{Recursive relation: } T(N) = 2T(N/2) + \Theta(N)}$

- We can just solve $T(N) = 2T(N/2) + N$.
- $T(N) = 2T(N/2) + N \ = 2[2T(N/4) + N/2] + N$

$$= 4T(N/4) + 2N \ = 4[2T(N/8) + N/4] + 2N$$

$$= 8T(N/8) + 3N \ = \cdots \ = 2^k T\left(N/2^k\right) + kN$$

$$= \cdots \ = NT(1) + N\log_2 N$$

- $T(1)$ is a constant.
- $T(N) = \Theta(N \log N)$
  - Both average case and worst case.

# 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()`** is stable.
- Suitable for parallel implementation.

# Merge Sort

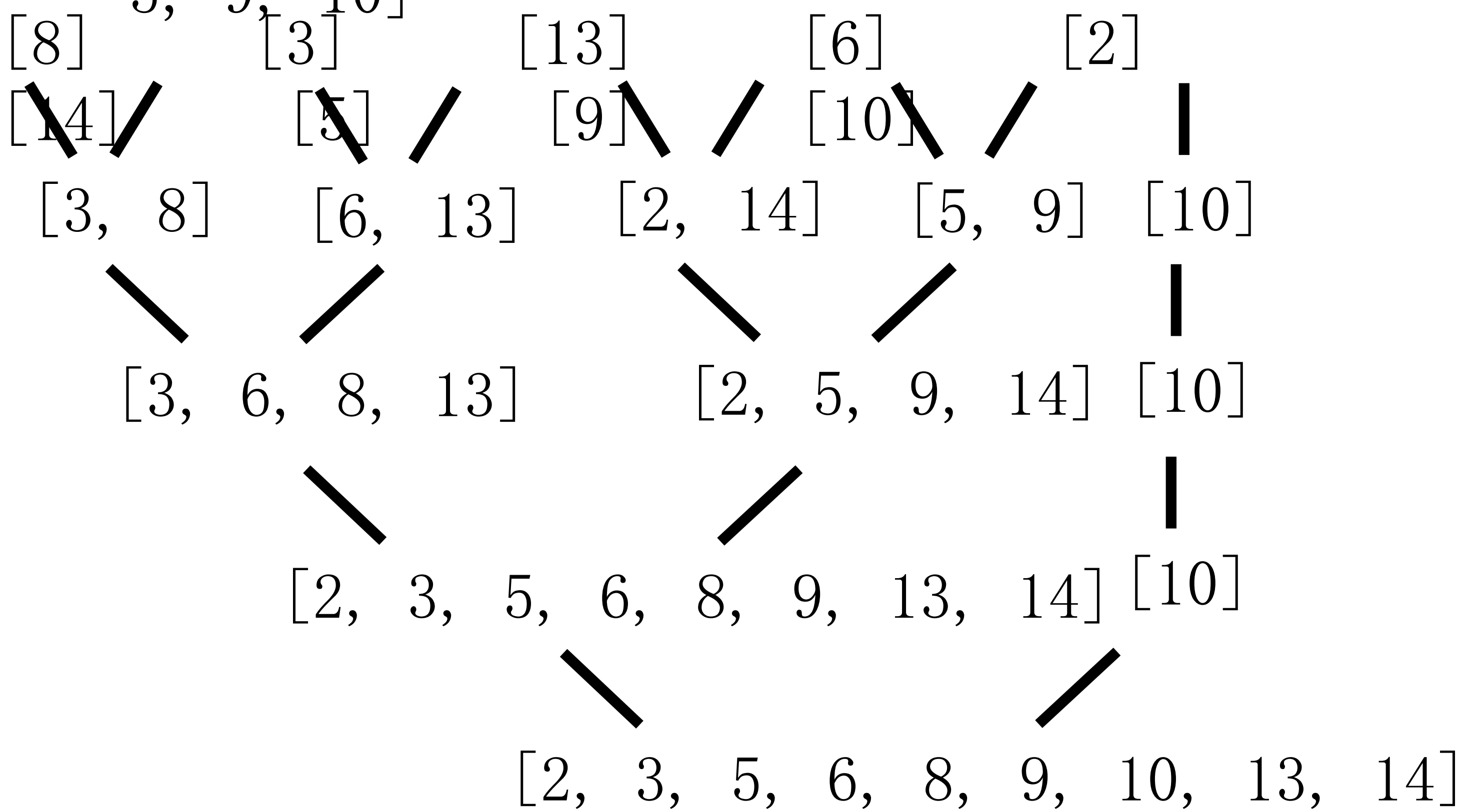
## Non-Recursive Version

- Consider the original array as N subarrays of size 1.
- Scan through array to perform pairwise merging to produce (roughly) N/2 sorted subarrays of 2 elements.
- Scan through array to perform pairwise merging to produce (roughly) N/4 sorted subarrays of 4 elements.
- etc.
- Scan through array to perform final merging of two sorted subarrays to produce a sorted array of N elements.

# Merge Sort

## Non-Recursive Version Example

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



# Merge Sort

## Summary

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

# Outline

- Simple Sorting Algorithms and Heap Sort
- Merge Sort
- Quick Sort

# Quick Sort

## Algorithm

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

**partition()**

```
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

- The ideal pivot divides the array into two equal-sized partitions.
  - This requires the pivot to be the **median** value of all items in the array.
  - However, computing/finding the median requires the array to be sorted in the first place!
- If your input is random, you can choose the first element, but this is very bad for presorted input.
- You can also randomly pick a pivot in the array.

# Choice of Pivot

- Median of three heuristic: randomly select three items and choose their median as pivot.
  - Randomness does not matter. Choosing the first, last, and middle items works the same.
- Worst case pivot divides the array into partitions of size 0 and size $N - 1$.
- If choosing the pivot randomly, we have $2/N$ probability to choose the worst case pivot.
- With median of three heuristic, 0 probability to choose the worst case pivot.

# 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 ≥ 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

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 |