CS202 – Data Structures

LECTURE-23

Sorting

Comparison based Sorting algorithms

Dr. Maryam Abdul Ghafoor
Assistant Professor
Department of Computer Science, SBASSE

Agenda

- Sorting Applications
- Views of Sorting
- Sorting Algorithms
  - Insertion Sort, Selection Sort, Merge Sort

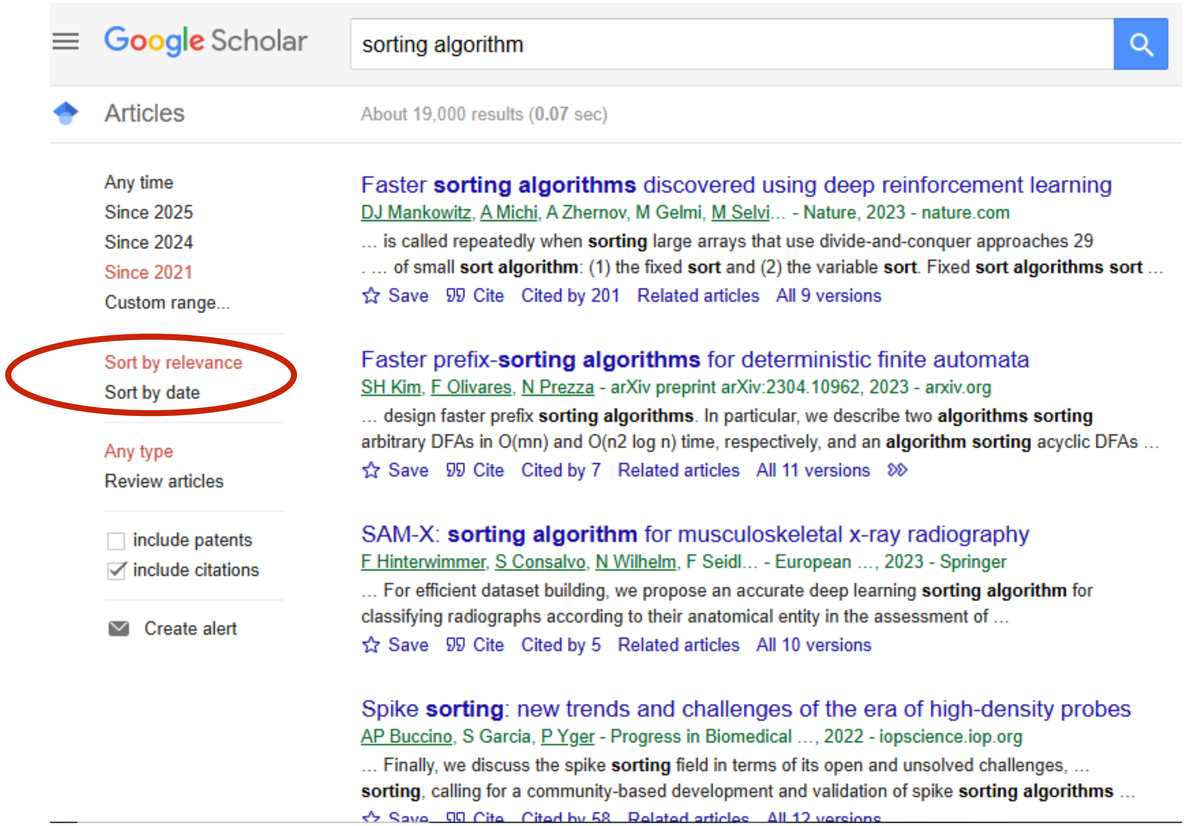
Quick Question?

- Given an arbitrary array of integers, how would you find the farthest pair and closest pair?
  - Farthest pair means a pair of integers with maximum difference between them
  - Closest pair means two integers with minimum possible difference between them.

Sorting

Why use Sorting?

- Used in a variety of applications
  - Emails are sorted by received date/time, last name, from, etc
  - Google sorts your search results by relevance



Why use Sorting?

- Sorting is the basic building block for many algorithms
  - Search preprocessing → speeds up search
  - Equivalent items are adjacent → allows rapid duplicate finding

- Sorting world records: http://sortbenchmark.org
  - In 2016, Tencent corporation proposed the fastest sorting algo
    - 37 TB of data in 60 secs (on one dataset)
  - In 2023, Mendelev proposed an algorithm for sorting 59.3K / Joules

Two views of Sorting

Permutations vs Inversions

Ordering Relation

- An ordering relation is a binary relation on a set that can be used to order(sort) the elements in the set.
- The ordering relation <= for the keys a, b and c has the following properties:
  - Law of Trichotomy: Exactly one of $a < b$, $a = b$, $b < a$ is true
  - Law of Transitivity: If $a < b$, and $b < c$, then $a < c$
- An ordering relation with the above properties is known as a "total order" (or linear order)

Example of an Ordering Relation: String length

Law of Trichotomy: Exactly one of the following is true

$$\text{len}(a) < \text{len}(b)$$
$$\text{len}(a) = \text{len}(b)$$
$$\text{len}(b) < \text{len}(a)$$

Law of Transitivity: If $\text{len}(a) < \text{len}(b)$ and $\text{len}(b) < \text{len}(c)$, then $\text{len}(a) < \text{len}(c)$

Valid sorts for [“cows”, “set”, “going”, “the”] for the ordering relation above:

- [“the”, “set”, “cows”, “going”] → **unstable**
- [“set”, “the”, “cows”, “going”] → **Stable**

Under this relation, “the” is considered = to “set”, since len(“the”) = len(“set”)

Sorting Definition and Views

Goal: Taking a list of objects which could be stored in a linear order, e.g., numbers $(a_0, a_1, \ldots, a_{n-1})$ and select the permutation of list of objects such that

$$a'_0 \leq a'_1 \leq \cdots \leq a'_{n-1}$$

A sort is a permutation (or re-arrangement) of a sequence of keys that puts them in non-decreasing order (ascending order)

Sorting: An Alternate View

An ***inversion*** is a pair of elements that are out of order with respect to '<'


Gabriel Cramer

Given a permutation of n elements (a_0, a_1, ..., a_{n-1}) , an inversion is defined as a pair of entries which are reversed, i.e., (aj, ak) forms an inversion if j < k but aj > ak

Ref: Bruno Preiss, Data Structures and Algorithms

Goal: Given a sequence of elements with X inversions, perform a sequence of operations that reduces inversions to 0

Rank the following lists based on the number of inversions required to sort them, from most to least.

1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95

1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99

22 20 81 38 95 84 99 12 79 44 26 87 96 10 48 80 1 31 16 92

To what degree are these three lists unsorted?

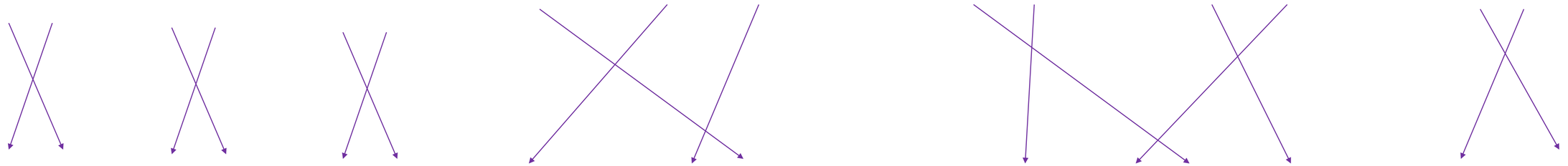
1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95

1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99

22 20 81 38 95 84 99 12 79 44 26 87 96 10 48 80 1 31 16 92

Number of Pairs Out of Order?

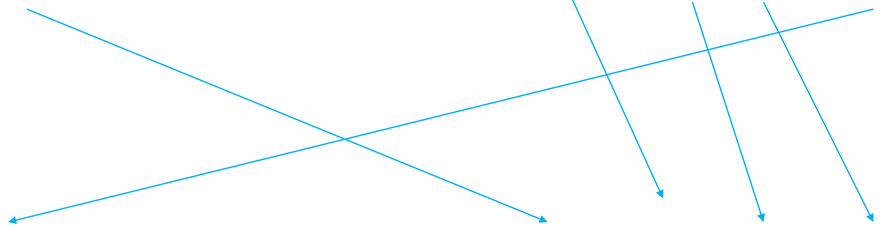
1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95



1 12 16 25 26 33 35 42 45 56 58 67 74 75 81 83 86 88 95 99

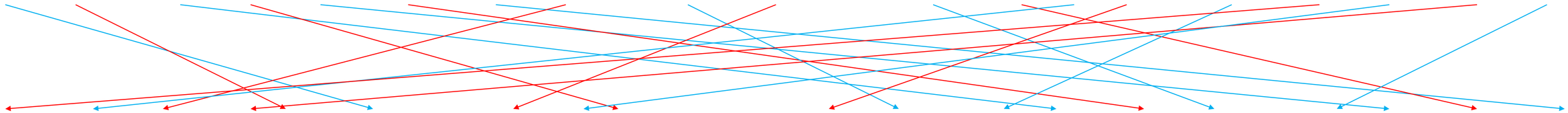
Number of Pairs Out of Order?

1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99



1 17 21 23 24 27 32 35 42 45 47 57 66 69 70 76 85 87 95 99

22 20 81 38 95 84 99 12 79 44 26 87 96 10 48 80 1 31 16 92



1 10 12 16 20 22 26 31 38 44 48 79 80 81 84 87 92 95 96 99

How many inversion(pairs out of order) should be removed to sort the list L, L = [1, 3, 5, 4, 2, 6]

0

at least 1

0%

5

0%

at most 4

0%

6

0%

Sorting – Inversion View

Example: (1, 3, 5, 4, 2, 6)

(1, 3) (1, 5) (1, 4) (1, 2) (1, 6)
(3, 5) (3, 4) (3, 2) (3, 6)
(5, 4) (5, 2) (5, 6)
(4, 2) (4, 6)
(2, 6)

Key Observation:

Given any list of n numbers, there are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs of numbers. For a random ordering, approximately half of all pairs, or $\frac{1}{2}\binom{n}{2} = \frac{n(n-1)}{4} = \mathbf{O}(n^2)$ inversions exists.

Memory Efficiency in Sorting

- **In-place sorting algorithms**
  - require the allocation of at most **O(1)** additional memory
  - Use fixed amount of extra memory, independent of the input size.

- Out-of-place sorting algorithms
  - Require the allocation of secondary array (or data structure) of size proportional to the input
  - Use O(n) additional memory

Trade-offs

- Memory vs Performance
- Algorithm Complexity
- Stability

Building Blocks – moveMin()

Input: An unsorted array A of k numbers.

Output: An array where minimum number is at the first position, rest remain at their positions.

What is the computational complexity?

Building Blocks – Insert()

Input: An array ***A*** of k numbers with first k-1 sorted in ascending order and the last number is out of place.

Output: An array ***A***[1…..k] sorted in ascending order

What is the computational complexity?

Selection Sort

Selection Sort (SS)

- Invariant: SS maintains a list S, which remains sorted at each step of the algorithm

```
Algorithm
  (a) Start with an empty list S and the input list I
  (b) for (i=0; i<n; i++){
        find x, the smallest value in I
        remove x from I
        append x to the end of S
      }
```

Question?

- If l is a linked list, what is the worst-case complexity for finding the minimum?

- What if l is an array?

When I is an array or a linked list…

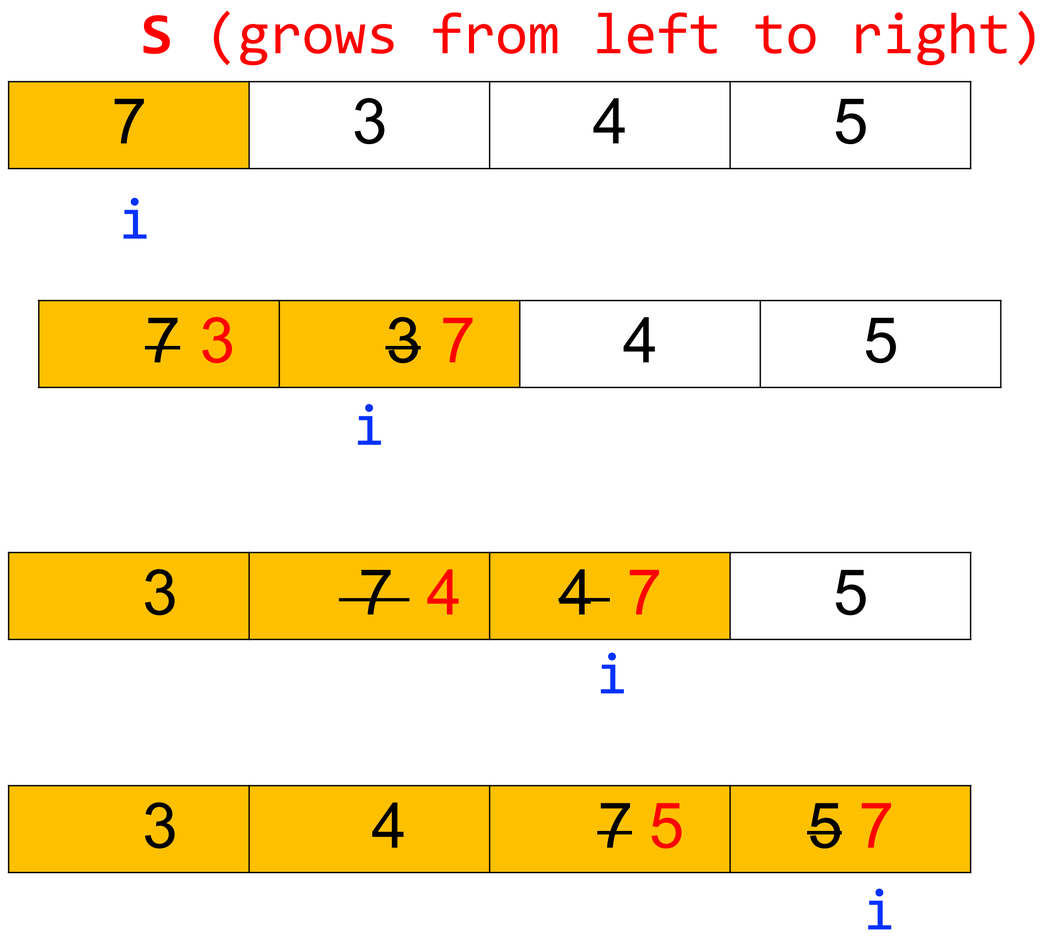
- Searching for the minimum always takes linear time
- Thus, it takes $\Theta(n^2)$ time even in the best case!
- Good news: it can be done in-place too

Example

1. I = {7, 3, 4, 5}	S = { }
2. I = {7, 4, 5}	S = {3}
3. I = {7, 5}	S = {3, 4}
4. I = {7}	S = {3, 4, 5}
5. I = { }	S = {3, 4, 5, 7}

An in-place implementation of Selection Sort

```
void sort(Vector<int> & vec) {
  int n = vec.size();
  for (int i = 0; i < n; i++){
       int ind = findMin(vec, i+1);
       int tmp = vec[ind];
       vec[ind] = vec[i];
       vec[i] = tmp;
  }
}

 // finds the index of min element
int findMin(Vector<int> &vec, int j){
   int currMinIdx = j;
   for (;j < vec.size();j++) {
       if(vec[j] < vec[currMinIdx])
           currMinIdx = j;
   }
   return currMinIdx;
}
```



What is the computational cost of selection sort?

O(n)

0%

O(log n)

0%

O(n^2)

0%

O(n log n)

0%

What is the Computational Cost for Selection Sort?

$$n + (n-1) + (n-2) + \ldots + 1$$

$$= n(n+1)/2$$

$$\in \Theta(n^2)$$

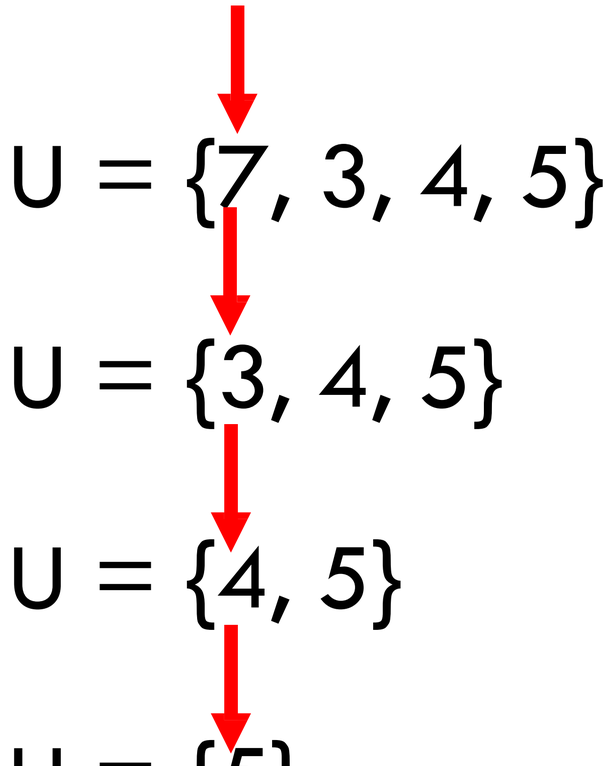
Insertion Sort

Insertion Sort (IS)

- Invariant: Algorithm maintains a list S, which remains sorted at each step of the algorithm

```
Algorithm
  (a) Start with an empty list S and an unsorted list U
  (b) for (each item x in U){
                  insert x into S in sorted order
      }
```

Example



U = {7, 3, 4, 5}

U = {3, 4, 5}

U = {4, 5}

U = {5}

U = { }

S = { }

S = { 7}

S = {3,7}

S = {3, 4, 7}

S = {3, 4, 5, 7}

S = {3, 4, 5, 7}

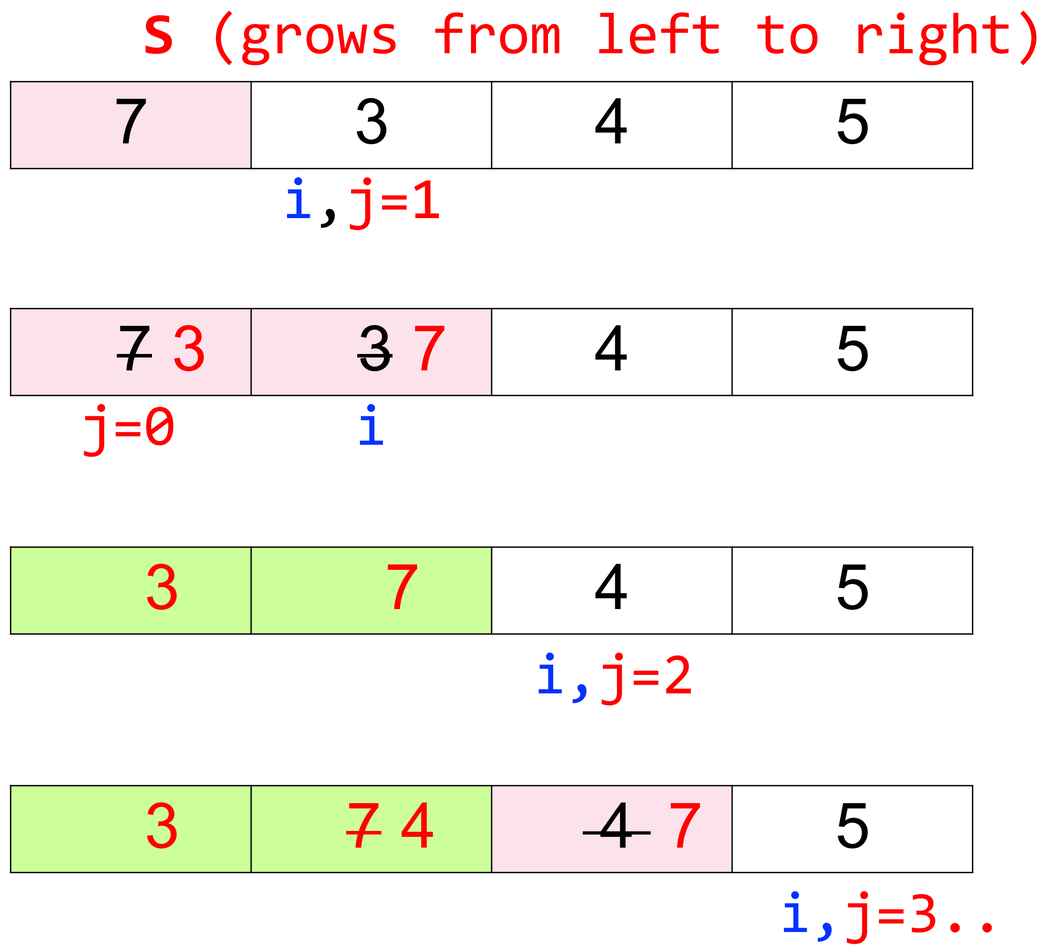
Question?

- If S is a linked list, what is the worst-case complexity for finding the right position?

- What if S is an array?

An in-place implementation of Insertion Sort

```
void sort(Vector<int> & vec) {
  int n = vec.size();
  // already-sorted section grows 1 at a
    // time from left to right
  for (int i = 1; i < n; i++) {
   insert(vec, i);
  }
}
void insert(vector<int> &vec, int j){
  while (j > 0 && vec[j-1] > vec[j]) {
 // keep swapping this item with
 // its left neighbor if it is smaller
 // than the left neighbor
    int tmp = vec[j-1]
    vec[j-1] = vec[j];
    vec[j] = tmp;
     j--;
  }
}
```



Concept Check!

PollEv

Which of these arrays does InsertionSort sort the slowest? What about the fastest?

[2, 6, 1, 2, 3, 4]

[1, 2, 3, 4, 5, 6]

[6, 5, 4, 3, 2, 1]

[3, 2, 1, 5, 7, 1]

Which of these arrays does Insertion Sort sort the slowest?

[2, 6, 1, 2, 3, 4]

0%

[1, 2, 3, 4, 5, 6]

0%

[6, 5, 4, 3, 2, 1]

0%

[3, 2, 1, 5, 7, 1]

0%

Which of these arrays does Insertion Sort sort the fastest?

[2, 6, 1, 2, 3, 4]

0%

[1, 2, 3, 4, 5, 6]

0%

[6, 5, 4, 3, 2, 1]

0%

[3, 2, 1, 5, 7, 1]

0%

With sorted input, can we do better than quadratic time?

1. U = {1, 2, 3, 5}	S = { }
2. U = {2, 3, 5}	S = {1}
3. U = {3, 5}	S = {1, 2}
4. U = {5}	S = {1, 2, 3}
5. U = { }	S = {1, 2, 3, 5}

- Insertion sort takes $O(n)$ time if the list is completely sorted!
- In general, the running time is proportional to the number of inversions (or comparisons) need to find the right spot

Concept Check!

PollEv

- How can we efficiently check if the input is already sorted?

How can we efficiently check if the input is already sorted?

Nobody has responded yet.

Hang tight! Responses are coming in.

Can We use a BST to represent S?

- Yes!
  - We can take an item from U and do an insert in the BST, which takes $O(logn)$ on average
  - Note: In CS, we do not associate this version with Insertion Sort

- Key:
  - the data structure you use can make a huge difference!

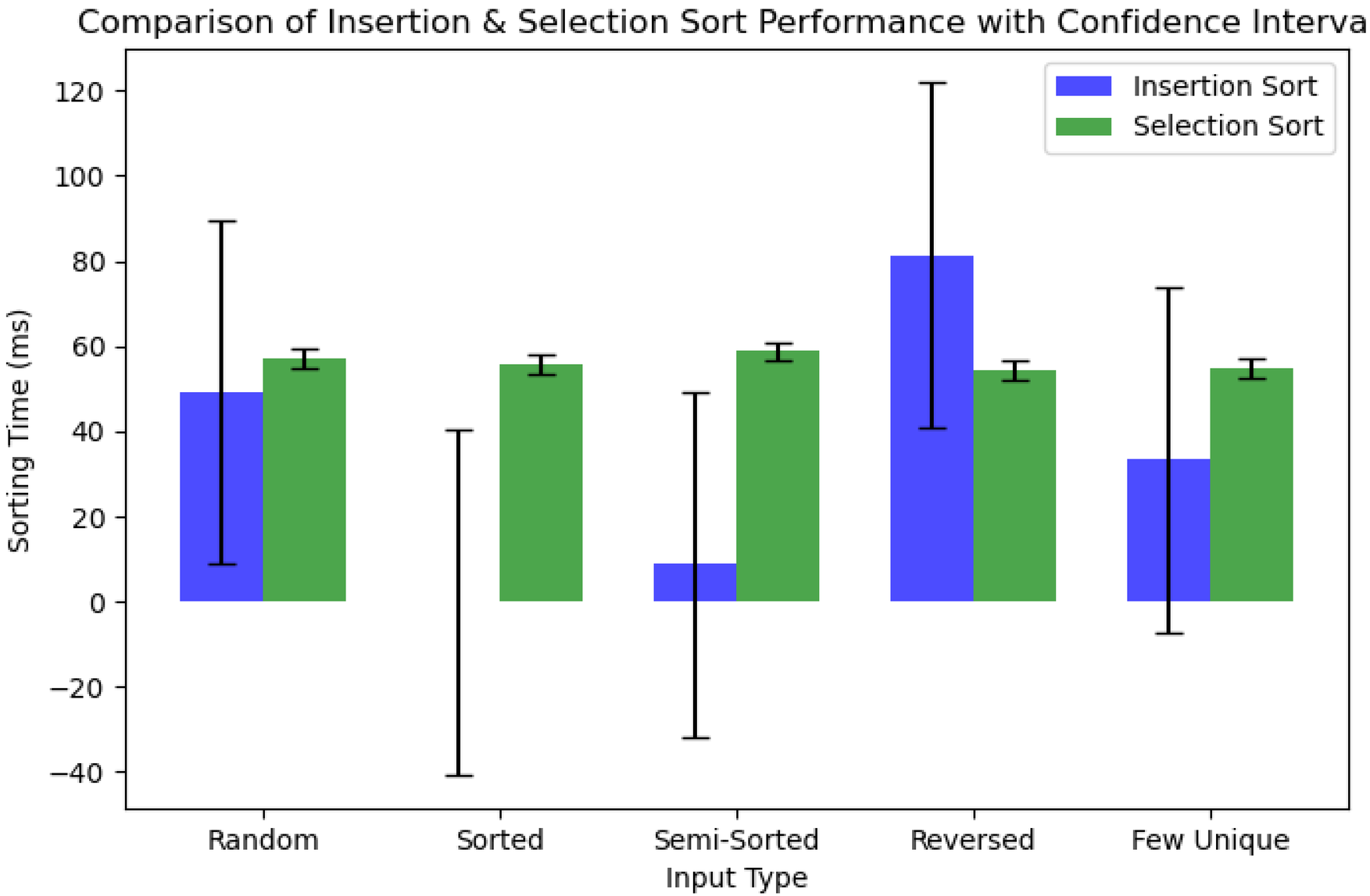
Computational Cost

Array Size = 5000

Range of numbers = 1 – 10000

Input Type	Insertion Sort(ms)	Selection Sort(ms)
Random	49.23	57.12
Sorted	0.0311	55.55
Semi Sorted	8.65	58.70
Reversed	81.22	54.11
Few Unique	33.37	54.77

Comparison of Computational Cost



Question?

- Suppose **selection sort** on an array with 800 elements takes **x ms.** How long would selection sort take on an array with 400 elements?

x/4 ms

For a quadratic function halving the input size will reduce the time to quarter.

Time Required to Sort the Two Lists

- Suppose selection sort on an array with 800 elements takes = x ms
- Sorting two arrays of size 400 will take = x/4 + x/4

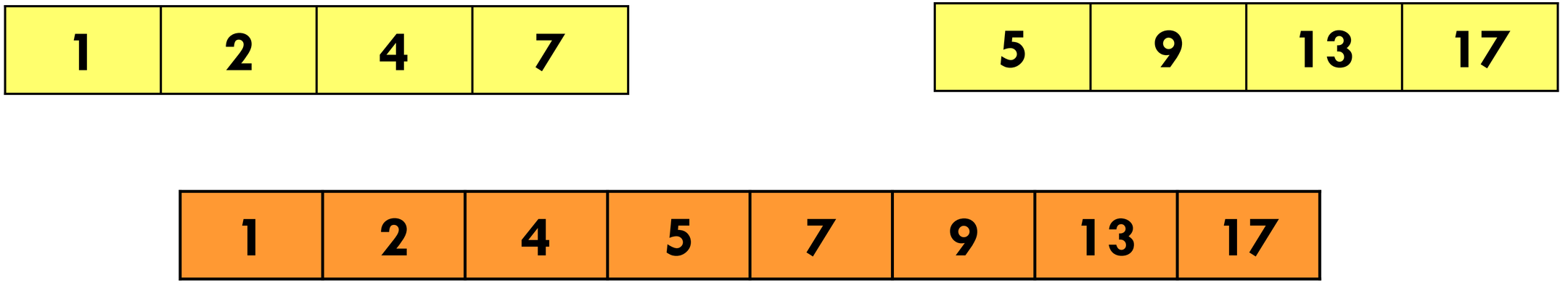
  = x/2

Sorting smaller arrays speed up!

Divide-and-conquer algorithms take advantage of the fact that smaller inputs can be sorted much faster.

Question?

- Suppose we have two sorted lists. How can we efficiently merge them into a (larger) sorted list?



- Key insight: We can merge two sorted lists in linear time!

Algorithm

- Let L1 and L2 be two sorted lists
- Let L be an empty list to merge these into

```
while( i < L1.size() or j < L2.size()){
    if(j == L2.size())
        L[k++] = L1[i++];
    else  if(i == L1.size())
        L[k++] = L2[j++];
    else
        L[k++] = (L1[i]   < L2[j]) ? L1[i++] : L2[j++]
```

Merge Sort

Mergesort

- Proposed by John Von Neuman in 1945
- It is a recursive divide and conquer algorithm
- High-level idea
  - Divide the problem into smaller parts
  - Independently solve the parts
  - Combine the solutions to get the overall solution

Merge Sort

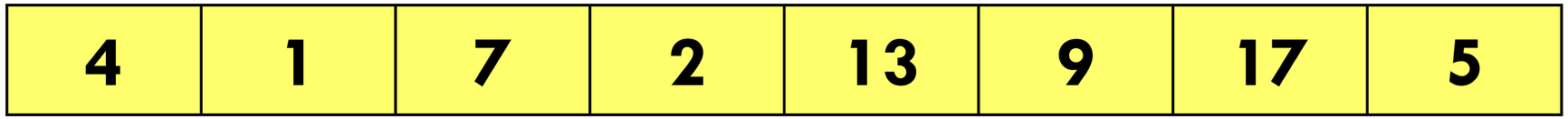
- Base case:
  - An empty or length-1 list is already sorted

- Recursive case:
  - Break each list in half and recursively sort (merge sort) each half
  - Merge them back into a single sorted list

Algorithm

1. Start with an unsorted list I of n items
2. Break I into two halves I_1 and I_2 where I_1 has $\lfloor n/2 \rfloor$ items and I_2 has $\lceil n/2 \rceil$ items
3. Sort I_1 recursively, yielding S_1
4. Sort I_2 recursively, yielding S_2
5. Merge S_1 and S_2 into a sorted list

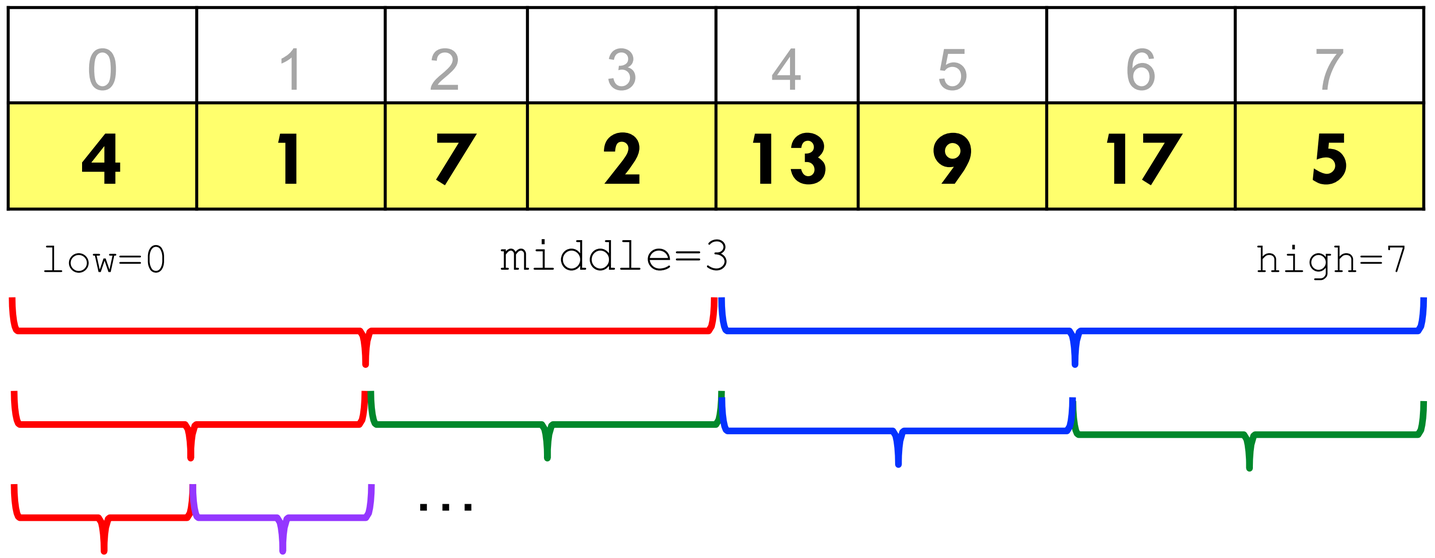
Example



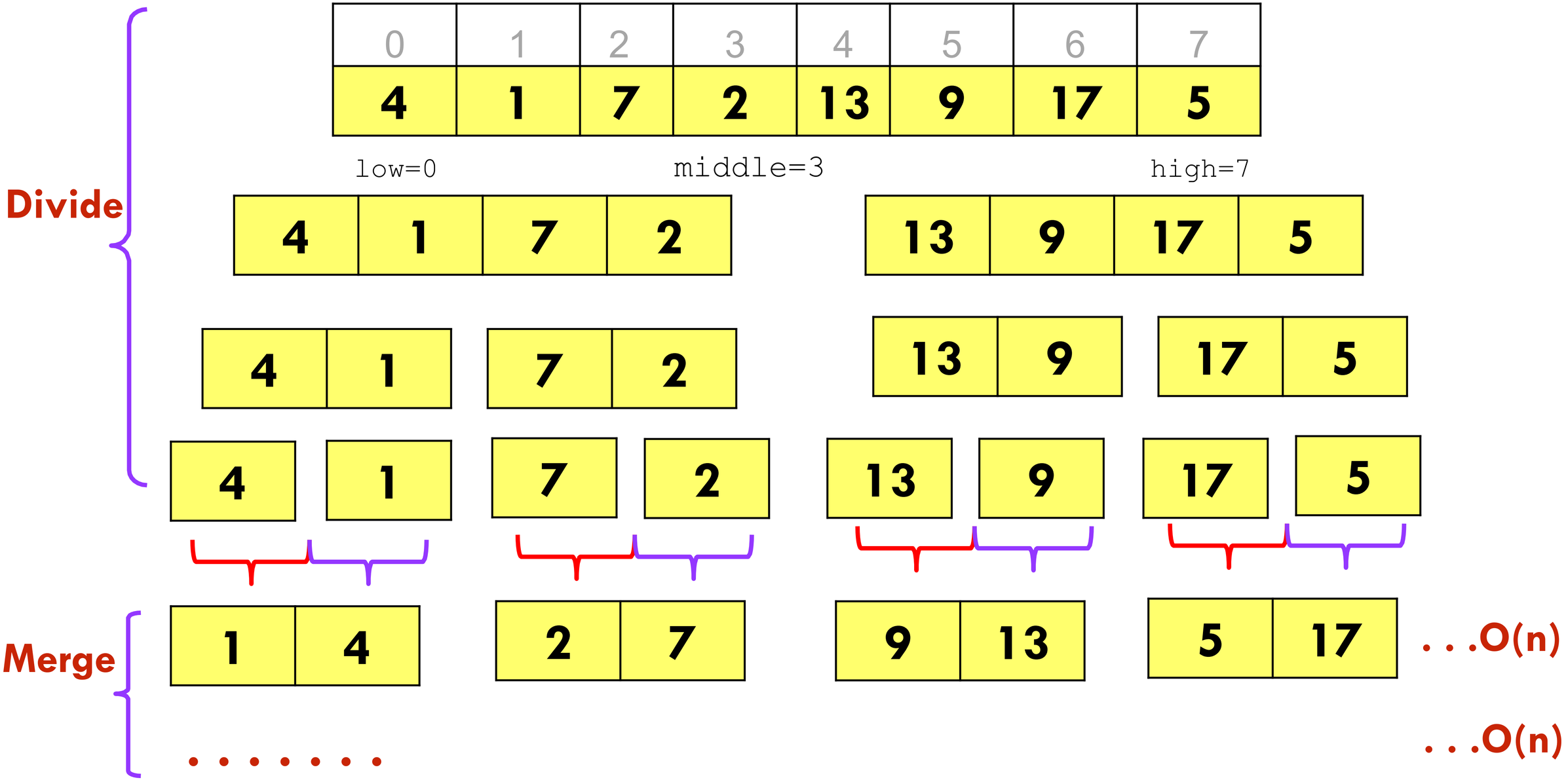
Merge Sort

```
/* low refers to the left index and high to the right index of array s to be sorted */
void mergeSort(int s[], int low, int high)
{
        if (low < high)
        {
                // Same as (low+high)/2, but avoids overflow for large low and high values
                int middle = low + (high-low)/2;

                // Sort first and second halves
                mergeSort(s, low, middle);
                mergeSort(s, middle+1, high);

                // Merge the sorted lists
                merge(s, low, middle, high);
        }
}
```

```
/* low refers to the left index and high to the right index of array s to be sorted */
void mergeSort(int s[], int low, int high)
{
        if (low < high)
        {
                // Same as (low+high)/2, but avoids overflow for large low and high values
                int middle = low + (high-low)/2;

                // Sort first and second halves
                mergeSort(s, low, middle);
                mergeSort(s, middle+1, high);

                // Merge the sorted lists
                merge(s, low, middle, high);
        }
}
```



Example


0 1 2 3 4 5 6 7
4 1 7 2 13 9 17 5
low=0
middle=3
high=7
Divide
4 1 7 2
13 9 17 5
4 1
7 2
13 9
17 5
4
1
7
2
13
9
17
5
Merge
1 4
2 7
9 13
5 17
. . .O(n)
. . .O(n)
. . . . . .

Time Complexity

- Number of levels is in $O(\log_2 n)$
- O(n) time per level for merging
- Thus, merge sort runs in $O(n \log_2 n)$

Merge Sort Summary

- Recursively sort left and right half of input, then merge result back into one sorted sequence.
- Divide step: easy (just split in half and recurse)
- Conquer step: hard (merge sorted sequences)
- **O(n log n)** sorting algorithm
  - This is better than Selection Sort!

Quicksort

Quick Sort

- Choose a "pivot" element
- Group your elements into three groups:
  - Less than pivot
  - Equal to pivot
  - Greater than pivot
- Recursively sort (quick sort) the less than and greater than groups
- Concatenate the three sorted groups back together again

Questions