

Remember when ...

... I said: "Upper Bound of an Algorithm"

- ▶ An algorithm A has an **upper bound** of $f(n)$ for input of size n if there exists **no input** of size n such that A requires more than $f(n)$ time
- ▶ E.g., we know from prior courses that Quicksort and Bubblesort take no more time than $O(n^2)$, while Mergesort has an upper bound of $O(n \log n)$

... I said: "Upper Bound of a Problem"

- ▶ A problem has an **upper bound** of $f(n)$ if there exists **at least one** algorithm that has an upper bound of $f(n)$
 - ▶ I.e., there exists an algorithm with time/space complexity of at most $f(n)$ on **all** inputs of size n
- ▶ E.g., since **algorithm** Mergesort has worst-case time complexity of $O(n \log n)$, the **problem** of sorting has an upper bound of $O(n \log n)$

Remember when ...

... I said: "Lower Bound of a Problem"

- ▶ A problem has a **lower bound** of $f(n)$ if, for **any** algorithm A to solve the problem, there exists **at least one** input of size n that forces A to take at least $f(n)$ time/space
- ▶ This pathological input depends on the specific algorithm A
- ▶ E.g., reverse order forces Bubblesort to take $\Omega(n^2)$ steps
- ▶ Since **every** sorting algorithm has an input of size n forcing $\Omega(n \log n)$ steps, sorting problem has **time complexity lower bound** of $\Omega(n \log n)$
- ▶ To argue a lower bound for a problem, can use an **adversarial argument**: An algorithm that simulates **arbitrary** algorithm A to build a pathological input
 - ▶ Needs to be in some general (algorithmic) form since the nature of the pathological input depends on the specific algorithm A
 - ▶ Adversary has unlimited computing resources
- ▶ Can also **reduce** one problem to another to establish lower bounds

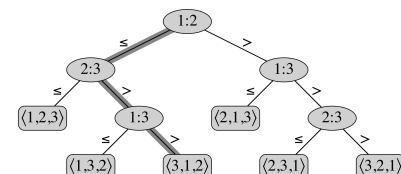
Comparison-Based Sorting Algorithms

- ▶ Our lower bound applies only to **comparison-based sorting algorithms**
 - ▶ The sorted order it determines is based **only** on comparisons between the input elements
 - ▶ E.g., Insertion Sort, Selection Sort, Mergesort, Quicksort, Heapsort
- ▶ What is **not** a comparison-based sorting algorithm?
 - ▶ The sorted order it determines is based on additional information, e.g., bounds on the range of input values
 - ▶ E.g., Counting Sort, Radix Sort

Decision Trees

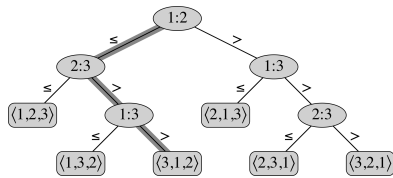
- ▶ A **decision tree** is a full binary tree that represents comparisons between elements performed by a particular sorting algorithm operating on a certain-sized input (n elements)
- ▶ **Key point:** a tree represents an algorithm's behavior on **all possible inputs** of size n
 - ▶ Thus, an adversarial argument could use such a tree to choose a pathological input
- ▶ Each internal node represents one comparison made by algorithm
 - ▶ Each node labeled as $i : j$, which represents comparison $A[i] \leq A[j]$
 - ▶ If, in the particular input, it is the case that $A[i] \leq A[j]$, then control flow moves to left child, otherwise to the right child
 - ▶ Each leaf represents a possible output of the algorithm, which is a permutation of the input
 - ▶ All permutations must be in the tree in order for algorithm to work properly

Example for Insertion Sort



- ▶ If $n = 3$, Insertion Sort first compares $A[1]$ to $A[2]$
- ▶ If $A[1] \leq A[2]$, then compare $A[2]$ to $A[3]$
- ▶ If $A[2] > A[3]$, then compare $A[1]$ to $A[3]$
- ▶ If $A[1] \leq A[3]$, then sorted order is $A[1], A[3], A[2]$

Example for Insertion Sort (2)



- ▶ Example: $A = [7, 8, 4]$
- ▶ First compare 7 to 8, then 8 to 4, then 7 to 4
- ▶ Output permutation is $\langle 3, 1, 2 \rangle$, which implies sorted order is 4, 7, 8
- ▶ What are worst-case inputs for this algorithm? What are not?

7/13

Proof of Lower Bound

- ▶ Length of path from root to output leaf is number of comparisons made by algorithm on that input
- ▶ Worst-case number of comparisons = length of longest path = **height** h
- ⇒ Adversary chooses a deepest leaf to create worst-case input
- ▶ Number of leaves in tree is $n!$ = number of outputs (permutations)
- ▶ A binary tree of height h has at most 2^h leaves
- ▶ Thus we have $2^h \geq n! \geq \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$
- ▶ Take base-2 logs of both sides to get

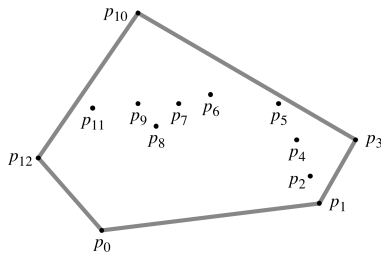
$$h \geq \lg \sqrt{2\pi n} + (1/2) \lg n + n \lg n - n \lg e = \Omega(n \log n)$$

- ⇒ **Every** comparison-based sorting algorithm has **some** input that forces it to make $\Omega(n \log n)$ comparisons
- ⇒ Mergesort and Heapsort are *asymptotically optimal*

8/13

Another Lower Bound: Convex Hull

- ▶ Use sorting lower bound to get lower bound on **convex hull** problem:
 - ▶ Given a set $Q = \{p_1, p_2, \dots, p_n\}$ of n points, each from $\mathbb{R}^2$, output $\text{CH}(Q)$, which is the smallest convex polygon P such that each point from Q is on P 's boundary or in its interior

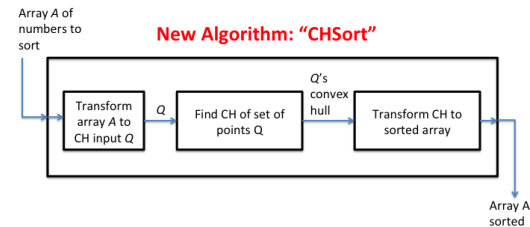


Example output of CH algorithm: ordered set $\langle p_{10}, p_3, p_1, p_0, p_{12} \rangle$

9/13

Another Lower Bound: Convex Hull (2)

- ▶ We will **reduce** the problem of sorting to that of finding a convex hull
- ▶ I.e., given any instance of the sorting problem $A = \{x_1, \dots, x_n\}$, we will transform it to an instance of convex hull such that the time complexity of the new algorithm sorting will be no more than that of convex hull

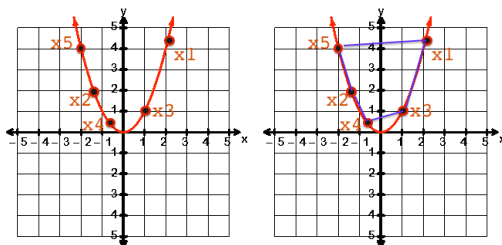


- ▶ The reduction: transform A to $Q = \{(x_1, x_1^2), (x_2, x_2^2), \dots, (x_n, x_n^2)\}$
- ⇒ Takes $O(n)$ time

10/13

Another Lower Bound: Convex Hull (3)

E.g., $A = \{2.1, -1.4, 1.0, -0.7, -2.0\}$,
 $\text{CH}(Q) = \langle (-1.4, 1.96), (-2, 4), (2.1, 4.41), (1, 1), (-0.7, 0.49) \rangle$



- ▶ Since the points in Q are on a parabola, all points of Q are on $\text{CH}(Q)$
- ▶ How can we get a sorted version of A from this?

11/13

Another Lower Bound: Convex Hull (4)

- ▶ CHSort yields a sorted list of points from **(any)** A
- ▶ Time complexity of CHSort: time to transform A to Q + time to find CH of Q + time to read sorted list from CH
- ⇒ $O(n)$ + time to find CH + $O(n)$
- ▶ If time for convex hull is $o(n \log n)$, then sorting is $o(n \log n)$
 - ⇒ Since that cannot happen, we know that convex hull is $\Omega(n \log n)$

12/13

