CS 6212 DESIGN AND ANALYSIS OF ALGORITHMS

LECTURE: LOWER BOUND THEORY

Instructor: Abdou Youssef

OBJECTIVES OF THIS LECTURE

By the end of this lecture, you will be able to:

- Articulate the distinction between algorithm complexity and problem complexity
- Describe the notion of lower bounds of computational problems, and explain their significance
- Do the same for upper bounds
- Explicate the notation of big-O, big-Ω, and big-Θ in the context of complexity of computational <u>problems</u>
- Compute the lower bounds of several important problems
- Apply certain techniques to derive lower bounds for new problems
- Derive the achievable lower bound of sorting
- Model comparison-based algorithms, and appreciate the need for models in proving lower bounds

OUTLINE

- Complexity of algorithms vs. complexity of problems
- Definition and significance of lower bounds of computational problems – the big-Ω notation
- Derivation of lower bounds for a number of problems
- Definition of upper bounds and their significance
- Big-Θ, achievable lower bounds, and speed-optimal algorithms
- Derivation of the famous achievable lower bound of sorting

PERSPECTIVE

--SWITCHING ATTENTION FROM ALGORITHMS TO PROBLEMS--

- Up to now, we focused on complexity of algorithms
- Today, we switch our attention to complexity of **problems**
- Specifically, we will address the minimum amount of time/computation needed to solve a problem no matter which algorithm or technique we use
- Alternatively, we look for intrinsic limits of time on solving problems computationally, limits that no algorithms for a given problem can beat

PERSPECTIVE
-- LOWER BOUNDS --

- Informally, a function $l(n)$ is a *lower bound* for a problem P if every existing/future algorithm for P takes at least $c.\,l(n)$ time for some constant c
- Formally, $l(n)$ is called a *lower bound* for a problem P if every algorithm for P takes time $\boldsymbol{T(n) = \Omega(l(n))}$
- Recall the Ω notation: $T(n) = \Omega(l(n))$ if

  $\exists c, n_0$ such that $T(n) \geq c.\,l(n)\ \forall n \geq n_0$

- Notation: If $l(n)$ is a lower bound for problem P, denote that by:

  P is $\boldsymbol{\Omega(l(n))}$ or **P = $\boldsymbol{\Omega(l(n))}$** or **c.** $\boldsymbol{l(n) \leq P}$

LOWER BOUNDS (LB)
-- MULTIPLICITY OF LBs, INSIGHT, ANALOGIES --

- If $l_1(n)$ is a lower bound for P, and $l_2(n)$ is function where $l_2(n) \leq c'.l_1(n)$ for some constant c' and $\forall n \geq n_1$ for some n_1, i.e., $l_1(n) = \Omega(l_2(n))$, then clearly $l_2(n)$ is another lower bound for P (Why?)
- Analogy: If a reliable expert announces that every house in Washington costs at least $300K (i.e., $300K is a lower bound on house prices), then clearly every house in Washington costs at least $200K (i.e., $200K is another lower bound). In fact, every quantity < $300K is a lower bound
- Which is the more interesting lower bound?
    - The higher one (i.e., $l_1(n)$)
    - Analogy: When we know for sure that every house costs at least $300K, and then someone says that every house costs at least $200K, we respond: "of course, because we already know that it costs at least $300K!"
    - So the smaller lower bounds are not interesting, they give us no new information.
- So the search is for higher and higher lower bounds, reaching for the intrinsic complexity of the problem

LESSONS LEARNED SO FAR

- A lower bound (L.B.) for a computational problem P cannot be beaten by any algorithm for P: **an L.B. is a speed barrier**
- For any P, the search is for higher and higher LB's, reaching for the intrinsic complexity of P
- No LB for P can be > than the complexity of an algorithm for P

EXAMPLES OF LOWER BOUNDS
-- THE MINIMUM PROBLEM --

- Problem P: finding the minimum in an arbitrary input array A[1:n]
- Find some lower bounds on P
- P is clearly $\Omega(1)$, because simply reporting the output takes O(1).
- In fact, **the size of the output is always a lower bound** because simply reporting the output (never mind how long it takes to compute it) takes that much time
- P is $\Omega(n)$. Why?
  - Every input element must be examined at least once to get the right answer. Why?
  - Because if an algorithm systematically ignores an input in some fixed position, and the input is random (i.e., has no structure), then the algorithm will not always be correct (e.g., when the minimum is actually in the ignored position)

EXAMPLES OF LOWER BOUNDS
-- THE MINIMUM PROBLEM IN A HEAP --

- Problem P: finding the minimum in a min-heap H
- Find some lower bounds on P
- P is clearly $\Omega(1)$, for reasons stated earlier
- Is P $\Omega(n)$? Why or Why not?
  - No, n is not a lower bound for this problem when the input is a min-heap
  - Because, indeed, the minimum is at the root, and so we can find it in O(1) time, without having to examine any other element in the heap
- So, is input size always a lower bound?
  - No, not when the input has some structure relevant to the problem
  - Yes, when the input is arbitrary/random in structure

EXAMPLES OF LOWER BOUNDS
-- CONSTRUCTION OF A HEAP--

- Problem P: Constructing a min-heap H from an arbitrary input array A[1:n]
- Find some lower bounds on P
- Is P $\Omega(n)$?
  - Yes, because the output size is n
- Is there a bigger lower bound than n for this problem?
  - No, because we know of an algorithm that constructs a heap in O(n)
- Since n is a lower bound and there is an algorithm that does it in O(n), we say that the lower bound n is ***achievable*** (more on that later)

EXAMPLES OF LOWER BOUNDS
-- CONSTRUCTION OF A BST-

- Problem P: Constructing a Binary Search tree from an arbitrary input array A[1:n]
- Find some lower bounds on P
- Is P $\Omega(n)$?
  - Yes, because the output size is n
- Is there a bigger lower bound than n for this problem?
  - Possibly: we don't know of any algorithms that constructs a BST in O(n) time
- Is P $\boldsymbol{\Omega(n \log n)}$**?** (We'll see later)

EXAMPLES OF LOWER BOUNDS
-- SEARCHING: IN A RANDOM ARRAY --

- Problem P: Searching for a value a in a random input array A[1:n]
- Is Searching $\boldsymbol{\Omega(n)}$**?**
  - Yes, because every input element has to be examined since the input has no structure relevant to the problem
- Is there a bigger lower bound than n for this problem?
  - No, because scanning algorithm (from start to end of the array) takes O(n) time, and so no lower bound can be bigger than that
- So n is an achievable lower bound for the random search problem

EXAMPLES OF LOWER BOUNDS
-- SEARCHING: IN A SORTED ARRAY --

- Problem P: Searching for a value a in a sorted input array A[1:n]
- Is Searching $\Omega(n)$?
  - No, because the *binary search* algorithm (a well-known search algorithm) takes O$(\log n)$, which is less than n
- Is Searching $\boldsymbol{\Omega(\log n)}$**?**
  - Possibly, since we don't know of any search algorithm in a sorted arrays that takes less than $\log n$
  - But is it? **Yes** (the proof is an exercise)

EXAMPLES OF LOWER BOUNDS
-- MERGING TWO SORTED ARRAYS --

- Problem P: Merging two sorted arrays $A[1{:}n]$ and $B[1{:}m]$
- Is there a bigger lower bound than $(n+m)$?
    - No, because we have a merging algorithm (which we developed early in the semester) which takes $O(n+m)$, and so no lower bound can be bigger than that
- Is Merging $\boldsymbol{\Omega(n+m)}$**?**
    - Possibly: We don't know of any merging algorithm that merges in less time
    - But is it? **Yes** (the proof is left as an exercise)

LESSONS LEARNED SO FAR

- A lower bound (L.B.) for a computational problem P cannot be beaten by any algorithm for P: **an L.B. is a speed barrier**
- For any P, the search is for higher and higher LB's, reaching for the intrinsic complexity of P
- No LB for P can be > than the complexity of an algorithm for P
- The size of the output is always a lower bound
- The size of the input may be a lower bound: if the input has no structure relevant to the problem

UPPER BOUNDS

- A function $u(n)$ is called an *upper bound* for (or on) a problem P if there is an algorithm for P that takes O$(u(n))$ time. We denote that by:

  P is O$(u(n))$ or **P =O$(u(n))$** or **P** $\leq c.u(n)$

- For example, $O(n^2)$ is an upper bound on Sorting, because there is a sorting algorithm (e.g., insertion sort) that takes $O(n^2)$ time.
- Also, $O(n \log n)$ is another upper bound on Sorting because heapsort takes $O(n \log n)$ time
- Clearly, if $u(n)$ is an upper bound for P, then any function larger than $u(n)$ is also an upper bound for P
- Why call it an upper bound then?
  - Because when we know that there is an O$(u(n))$ algorithm for P, then no one should bother to design a new algorithm that takes more time than O$(u(n))$ since no one in their right mind should use the slower algorithm!
- So, we're always aiming for smaller and smaller upper bounds (i.e., faster and faster algorithms) for a given problem

“MOTION” OF UPPER BOUNDS

- We always aim for smaller and smaller upper bounds (i.e., faster and faster algorithms) for a given problem
- Until when?
  - Until we develop an algorithm that achieves a known lower bound for the problem
  - At that point, there is no room for faster algorithms by definition of lower bounds.

UPPER AND LOWER BOUNDS
-- RELATION, SIGNIFICANCE, AND IMPLICATIONS --

- If $l(n)$ is a lower bound for P, and $u(n)$ is an upper bound on P, then

$$c_1 . l(n) \leq \mathrm{P} \leq c_2 . u(n)$$

- That implies that $l(n) \leq u(n)$ up to a constant factor
- That also implies that
  - There is an algorithm that takes $\mathrm{O}(u(n))$, and any future algorithm we aim to design for P should take less time than $u(n)$
  - Every algorithm for P will take at least $l(n)$ time
  - We should look for higher lower bounds than $l(n)$, if any
- The motion (or pressure) is always to **push** $l(n)$ up and $u(n)$ down
- What if they meet, i.e., $l(n) = u(n)$ up to a constant factor?

THE THETA Θ NOTATION

- What if $l(n) = u(n)$ up to a constant factor? Let $f(n) = l(n) = u(n)$
- **Definition**: We say that a problem P is $\Theta(f(n))$ if

  P is $\Omega(f(n))$ and P is $O(f(n))$

- That is, P is $\Theta(f(n))$ if $f(n)$ is at once a lower bound and an upper bound for P
- As an upper bound, it means there is an algorithm A for P that takes $O(f(n))$ time
- As a lower bound, it means that no algorithm can take less time than $f(n)$
- Therefore, algorithm A is the fastest possible for P, and we have it!
- So, we have a Theta situation whenever an algorithm for a problem P achieves a known lower bound for P
- Implications: Algorithm A is optimal (in speed), and there is no room for improvement (in speed or in lower bounds), i.e., P is *closed*!

LESSONS LEARNED SO FAR

- A lower bound (L.B.) for a computational problem P cannot be beaten by any algorithm for P: **an L.B. is a speed barrier**
- For any P, the search is for higher and higher LB's, reaching for the intrinsic complexity of P
- No LB for P can be > than the complexity of an algorithm for P
- The size of the output is always a lower bound
- The size of the input may be a lower bound: if the input has no structure relevant to the problem
- The push is always towards higher LBs and smaller upper bounds (faster algorithms) – until they meet
- When an algorithm for a problem P achieves a known LB for P, the algorithm is optimal in speed, and there is no room for improvement (in speed or LB), i.e., P is *closed*!

LOWER BOUND ON SORTING (1)
-- A TIGHT BOUND AND WHAT TO SHOW--

- **Theorem**: Sorting is $\Theta(n \log n)$
- Proof:
  - Need to show that Sorting is $O(n \log n)$ and Sorting is $\Omega(n \log n)$
  - Well, we have $O(n \log n)$ sorting algorithms (e.g., heapsort and mergesort)
  - So it remains to show that Sorting is $\Omega(n \log n)$

LOWER BOUND ON SORTING (2)
-- WHAT WE NEED TO SHOW SPECIFICALLY --

- **Theorem:** Sorting is $\Theta(n \log n)$
- Proof (Continued):
  - Show that Sorting is $\Omega(n \log n)$
  - For that, we need to show that every sorting algorithm takes at least $\Omega(n \log n)$ time
  - We can limit ourselves to **comparison-based** sorting algorithms

LOWER BOUND ON SORTING (3)
-- DILEMMA --

- Proof (Continued):
  - Need to show that **<u>every</u>** comparison-based sorting algorithm takes at least $\Omega(n \log n)$ time
  - **<u>Dilemma</u>**: The above goal presents some serious challenges
    - Not all possible sorting algorithms have been designed yet
    - There could potentially be an infinite list of sorting algorithms: we can never finish proving that each one of them takes at least $\Omega(n \log n)$ time

LOWER BOUND ON SORTING (4)
-- AN ANALOGY => INSPIRATION FOR A WAY OUT --

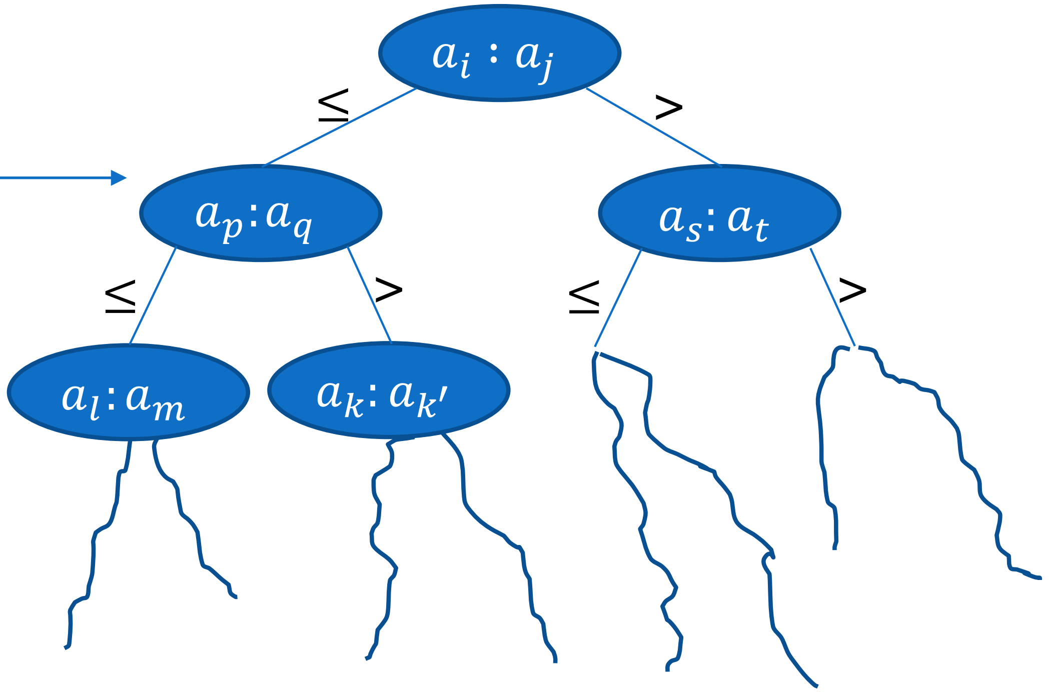
- A similar dilemma is when we're asked to prove that **"all humans are mortal"**
  - Not all humans are here (yet) to check
  - There is potentially an infinite list of humans to check, so we'll never finish examining their mortality one by one
- Is there a way out?
  - Yes, **Modeling:** need a (biological) model that captures the essential and common characteristics of the human body
  - Then we can focus on that model and prove that "the model will die one day"

LOWER BOUND ON SORTING (5)
-- MODELING, BUT WHAT MODEL? --

- Proof (Continued):
  - Need to show that every comparison-based sorting algorithm takes at least $\Omega(n \log n)$ time
  - How: Need a model (of every comparison-based sorting algorithm)
  - Once we have such a model, we'll prove that the model algorithm takes at least $\Omega(n \log n)$ time
  - So we have a methodological way out
  - But what would that model be like?

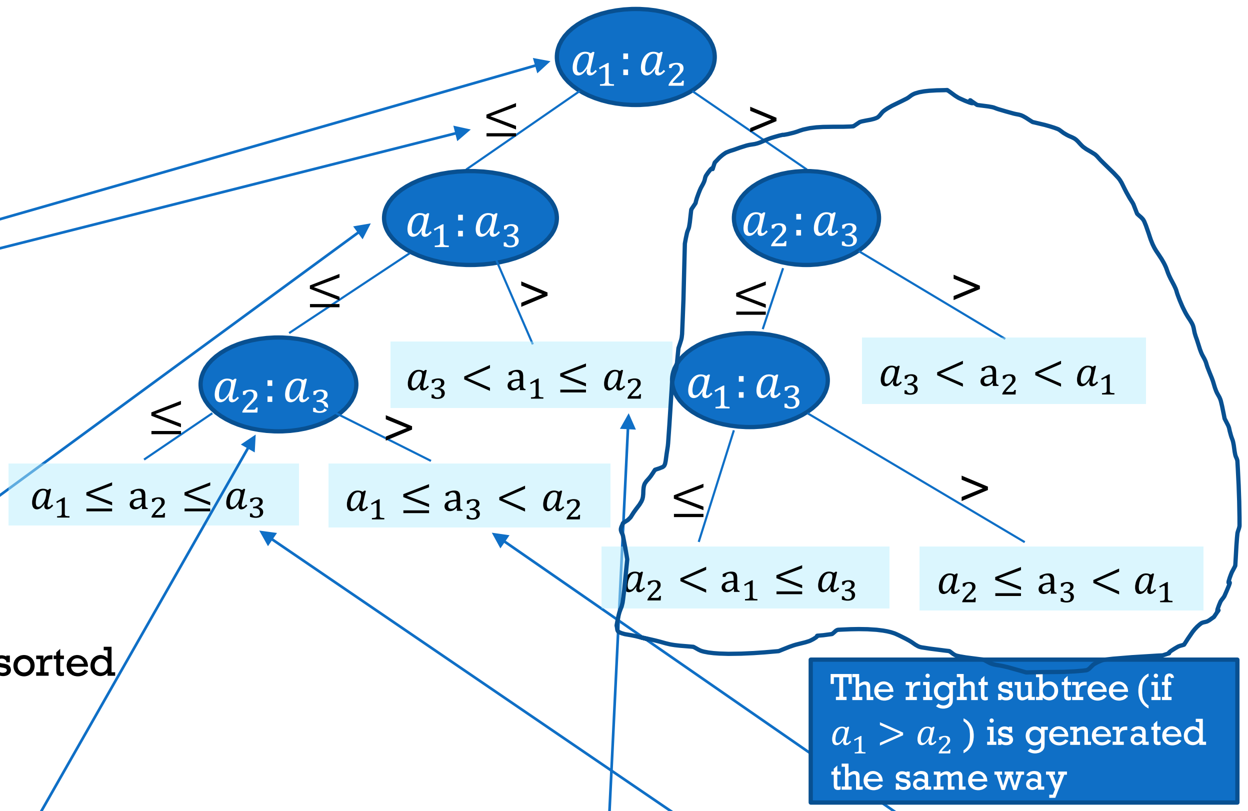
LOWER BOUND ON SORTING (5)
-- MODEL OF SORTING ALGORITHMS --

- Proof (Continued):
  - But what would that model be like?
  - The model is a **Comparison Tree**
  - It is a binary tree of comparison nodes
  - Every node is a comparison between two numbers in the input array
  - Take one arbitrary sorting algorithm
    1. The algorithm will pick one particular pair of numbers to compare
    2. Depending on the outcome of the comparison, the algorithm will pick another specific pair of numbers to compare,
    3. The last step is repeated until the algorithm "gathers enough information" to know how to order the elements
  - This will be illustrated on a specific sorting algorithm next



LOWER BOUND ON SORTING (6)
-- ILLUSTRATION OF COMPARISON TREES --

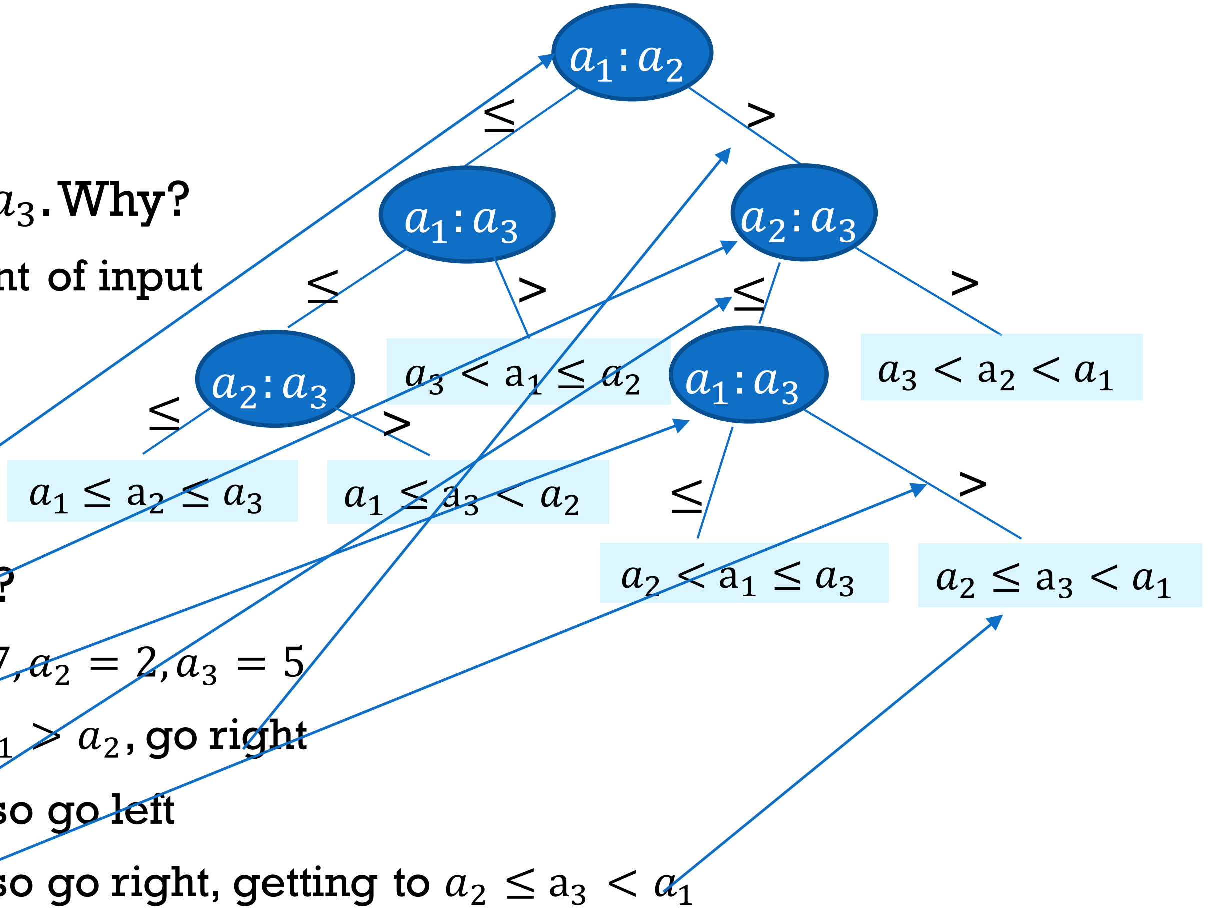
- We will show the comparison tree model of **mergesort** for sorting input of n=3 elements: a_1, a_2, a_3
- Partition input to 2 parts: $[a_1, a_2]$, $[a_3]$
- Partition $[a_1, a_2]$ into 2 halves: $[a_1]$, $[a_2]$
- Each of the two halves is sorted
- Now now merge $[a_1]$ and $[a_2]$
  - Compare a_1 with a_2
  - If $a_1 \leq a_2$:
    - Mergesort moves a_1 to output, then append a_2 to output, so $[a_1 \leq a_2]$ is sorted
    - $[a_3]$, being a single-number part, is already sorted
    - Now merge $[a_1 \leq a_2]$ with $[a_3]$:
      - Compare a_1 with a_3
        - If $a_1 \leq a_3$: move a_1 to output, and then compare a_2 with a_3: depending on outcome, we get $a_1 \leq a_2 \leq a_3$ or $a_1 \leq a_3 < a_2$
        - If $a_1 > a_3$: move a_3 to output, and then move $[a_1 \leq a_2]$ to end of output, getting $a_3 < a_1 \leq a_2$



LOWER BOUND ON SORTING (7)

-- OBSERVATIONS ABOUT THE COMPARISON TREE --

- How many leaves?
- 6: Why? What is the significance of 6?
- It is 3!, the number of permutations of a_1, a_2, a_3. Why?
    - Because the output can be any re-arrangement of input
- For input of length n, we have **n! leaves**
- The **tree** can be viewed as the **code of the whole sorting algorithm**
- How about a run/execution for a given input?
    - Try executing on the input **7, 2, 5**, i.e., $a_1 = 7, a_2 = 2, a_3 = 5$
    - Start at root, compare $a_1 : a_2$, i.e., 7:2, we get $a_1 > a_2$, go right
    - Now compare $a_2 : a_3$, i.e., 2:5, we get $a_2 \leq a_3$, so go left
    - Now compare $a_1 : a_3$, i.e., 7:5, we get $a_1 > a_3$, so go right, getting to $a_2 \leq a_3 < a_1$
    - That is, $2 \leq 5 < 7$ which is sorted
- So a **run is like tracing a path from the root to a leaf** (corresponding to the sorted order)



LOWER BOUND ON SORTING (8)

-- RECAP, AND REST OF THE PROOF--

1. Recap:
   a. Every sorting algorithm is modeled by a comparison tree (C.T.)
   b. The tree has n! leaves
   c. The tree is like the code for the algorithm
   d. An execution of the algorithm is like tracing a path from the root to a leaf
2. The (W.C) time of the algorithm is the length of the longest path from root to leaves
3. Therefore, the time of the algorithm is the height of the comparison tree: $\boldsymbol{T(n) = h}$
4. **Lemma**: In any binary tree of N leaves and height h, we have $N \leq 2^h$, i.e., $\boldsymbol{h \geq \log N}$.
   - Proof: $N \leq$ [# leaves in the completely filled binary tree (of height h)]=$2^h \Rightarrow N \leq 2^h \Rightarrow \log N \leq h$.

Using Stirling's formula

5. (4) and (1.b) $\Rightarrow$ the height h of the C.T. $\geq \log N = \log n! \geq \frac{1}{2} n \log n$, i.e., $\boldsymbol{h \geq \frac{1}{2} n \log n}$
6. (3) and (5) $\Rightarrow$ time of the arbitrary sorting algorithm is $\boldsymbol{T(n) = h \geq \frac{1}{2} n \log n = c.n \log n}$
7. Conclusion: $\boldsymbol{T(n) = \Omega(n \log n)}$. Q.E.D.

LESSONS LEARNED SO FAR

- A lower bound (L.B.) for a computational problem P **is a speed barrier**
- For any P, the search is for higher and higher LB's, up to the intrinsic complexity of P
- No LB for P can be greater than the complexity of an algorithm for P
- The size of the output is always a lower bound
- The size of the input may be a lower bound: if the input has no structure relevant to the problem
- The push is always to higher LBs and smaller upper bounds (faster algorithms) – until they meet
- When an algorithm for a problem P achieves a known LB for P, the algorithm is optimal in speed, and there is no room for improvement (in speed or LB), i.e., P is *closed*!
- Sorting is $\Theta(n \log n)$
- Proofs for lower bounds can present serious challenges, which can me surmounted using **modeling** of the algorithms for the problem at hand
- A good model for a class of algorithms can serve as a code for each of those algorithms and can conveniently offer the execution time of the modeled algorithm