CSE 317: Design and Analysis of Algorithms
Introduction

Shahid Hussain

Week 1: **Fall 2025**

SMCS, IBA-Karachi

"An algorithm must be seen to be believed." – **Donald Ervin Knuth**

Topics to be covered

1. Oh-notations and mathematical preliminaries
2. Sorting and searching
3. Recursion and induction
4. Divide and conquer algorithms
5. Dynamic programming
6. Graphs algorithms
7. Greedy algorithms
8. Randomized algorithms
9. Computational complexity and hardness results
10. Algorithms to cope hardness
11. Introduction to parallel algorithms/computation

Books

- Class participation policy: Background reading for next session and active participation in class discussions.
- Lecture notes/slides will be made available online
  - *Algorithms*, by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani.
  - *Algorithm Design Manual* by Steven Skiena.
  - *Data Structures and Algorithms in Python*, by Michael Goodrich, Roberto Tamassia, and Michael Goldwasser.
- Supplemental reading:
  - *Introduction to algorithms*, by Thomas Cormen, Charles Leiserson, Ronald Rivest and Clifford Stein.
  - *Introduction to Analysis of Algorithms*, by Robert Sedgewick and Philippe Flajolet.

Introduction

Finding the maximum element

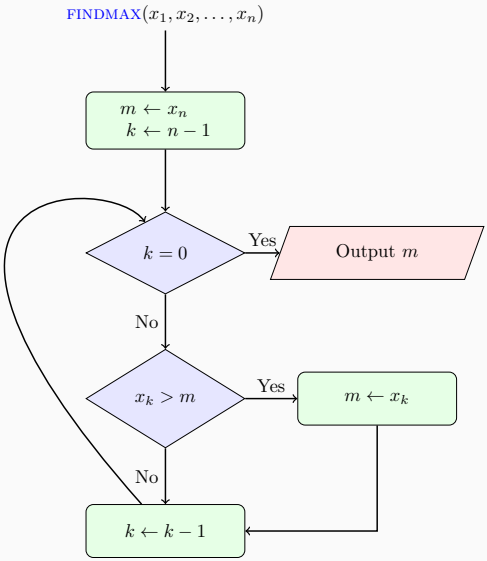
The `find max` problem

Consider the following simple algorithm to find the maximum element from a given sequence of numbers $x_1, x_2, \ldots, x_n$ (we can safely assume that these numbers are real and pair-wise distinct).

Finding the maximum element

Algorithm: FINDMAX
Input: A sequence of numbers $x_1, \ldots, x_n$, $n \geq 1$
Output: The maximum element from the input sequence

1. $m = x_k$
2. $k = n - 1$
3. **while** $(k \neq 0)$
4. **if** $x_k > m$
5. $m = x_k$
6. $k = k - 1$
7. **return** m



Finding the maximum element

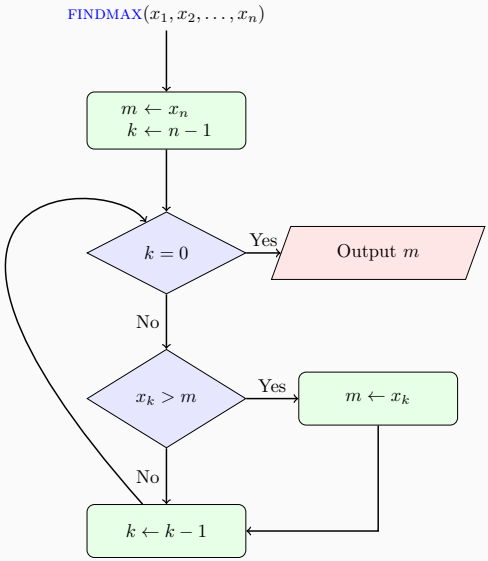
1. Let $m = 4$ and

x_1	x_2	x_3	x_4
2.3	7.1	4.0	5.9

2. We can trace the algorithm (flow chart) with this data. Following is the trace:

$$m \leftarrow \cancel{5.9}, \underline{7.1}$$
$$k \leftarrow \cancel{3}, \cancel{2}, \cancel{1}, \underline{0}$$

3. Is this algorithm correct?
4. We employ invariant. For the above algorithm we can place the invariant $m = \max(x_{k+1}, \ldots, x_n)$ before the first decision (the **if** statement) and it will always remain **true**.



Analysis

Analysis of the find max algorithm

- How efficient and and how fast an algorithm is and not just if the algorithm is correct
- When analyzing we are often interested in many different resources an algorithm uses for example the amount of memory, amount of running time, and communication requirement
- However, in our case we are interested in finding the running time of this algorithm
- For this particular algorithm we traverse the whole sequence exactly once however not all parts of algorithm are executed the same number of times
- For example, in this FINDMAX algorithm we are interested that how many times the statement $m \leftarrow x_k$ is executed

Analysis of the find max algorithm

- Assume that L denotes the number of times the statement $m \leftarrow x_k$ is executed
- We can perform following kinds of analysis i.e.,
  1. the minimum value of L (the best-case)
  2. the maximum value of L (the worst-case)
  3. the average value of L (the average-case)
- In our example:
  1. best-case value of $L = 0$ when x_n is the maximum number
  2. worst-case value of $L = n - 1$ when $x_n < x_{n-1} < \cdots < x_1$
  3. average-case value of L must be somewhere between 0 and $n - 1$
- Performing average-case analysis is one of the most important part of analysis of algorithms however it is often difficult as well

Analysis of the find max algorithm

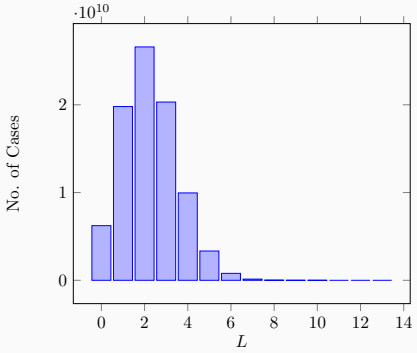
- For the n elements $x_1, x_2, \ldots, x_n$ we have $n!$ different permutations of these numbers
- For $n = 3$, assume (without loss of generality) the numbers are $1, 2, 3$, there are six different permutations:

Permutation			L
1	2	3	0
1	$\underline{3}$	2	1
2	1	3	0
2	$\underline{3}$	1	1
$\underline{3}$	1	2	1
$\underline{3}$	$\underline{2}$	1	2

$$\begin{aligned} \Pr[L = 0] &= 2/6 \\ \Pr[L = 1] &= 3/6 \\ \Pr[L = 2] &= 1/6 \end{aligned}$$

Analysis of the find max algorithm

- So for $n = 3$ the sequence is $2, 3, 1$ corresponding to number of cases with $L = 0, 1, 2$ respectively
- Similarly, for $n = 4$, we can see the sequence is $6, 11, 6, 1$ for $L = 0, 1, 2, 3$
- For $n = 5$ the sequence is $24, 50, 35, 10, 1$ for $L = 0, 1, 2, 3, 4$ and so on
- Following figure shows the number of cases with $n = 15$
- We can clearly see that average is clustered in the beginning of the values

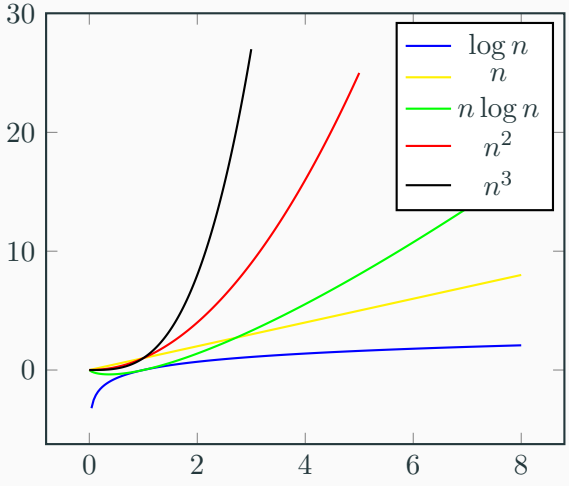


Time Complexity

Time complexity

- Time in an important resource while designing algorithms
- It is meaningless to say that an Algorithm A, when presented with input x, runs in time y seconds
- We, therefore, measure time in number of elementary operations such as arithmetic operations (addition and subtraction), comparisons, etc

Growth of functions



The big omicron (big oh)

- Let f and g be two functions of n from nonnegative integers to nonnegative integers
- We say that $f(n)$ is in big-oh of $g(n)$, i.e., $f(n) = O(g(n))$, if and only if, there exists constants c and n_0 such that

$$f(n) \leq c \cdot g(n), \qquad \text{for all } n \geq n_0$$

- Consequently, if $\lim_{n\to\infty} f(n)/g(n)$ exists, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty \text{ implies } f(n) = O(g(n))$$

The big omega

- Let f and g be two functions of n from nonnegative integers to nonnegative integers
- We say that $f(n)$ is in big-omega of $g(n)$, i.e., $f(n) = \Omega(g(n))$, if and only if, there exists constants c and n_0 such that

$$f(n) \geq c \cdot g(n), \qquad \text{for all } n \geq n_0$$

- Consequently, if $\lim_{n\to\infty} f(n)/g(n)$ exists, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0 \text{ implies } f(n) = \Omega(g(n))$$

- Informally, $f(n) = \Omega(g(n))$ if and only if $g(n) = O(f(n))$

The big theta

- Let f and g be two functions of n from nonnegative integers to nonnegative integers
- We say that $f(n)$ is in big-theta of $g(n)$, i.e., $f(n) = \Theta(g(n))$, if and only if, there exists constants c_1, c_2, and n_0 such that

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \qquad \text{for all } n \geq n_0$$

- Consequently, if $\lim_{n\to\infty} f(n)/g(n)$ exists, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \text{ implies } f(n) = \Theta(g(n))$$

- Importantly, $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

The small omicron (small oh)

- Let f and g be two functions of n from nonnegative integers to nonnegative integers
- We say that $f(n)$ is in small-omicron of $g(n)$, i.e., $f(n) = o(g(n))$, if and only if, there exists a constant n_0 such that for all possible values of $c > 0$

$$f(n) < c \cdot g(n), \qquad \text{for all } n \geq n_0$$

- Consequently, the $\lim_{n\to\infty} f(n)/g(n)$ exists and equal to zero i.e.,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \text{ implies } f(n) = o(g(n)).$$

The small omega

- Let f and g be two functions of n from nonnegative integers to nonnegative integers
- We say that $f(n)$ is in small-omega of $g(n)$, i.e., $f(n) = \omega(g(n))$, if and only if, there exists a constant n_0 such that for all possible values of $c > 0$

$$f(n) > c \cdot g(n), \qquad \text{for all } n \geq n_0$$

- Consequently, the $\lim_{n\to\infty} f(n)/g(n)$ is infinite i.e.,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \text{ implies } f(n) = \omega(g(n)).$$

Some examples

1. Let $f(n) = 10n^3 + 20n$ then $f(n) = O(n^3)$ also $f(n) = \Omega(n^3)$ therefore $f(n) = \Theta(n^3)$
2. In general, let $f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$, then $f(n) = \Theta(n^k)$
3. Let $f(n) = \log n^2$ then $f(n) = 2 \log n$ therefore $f(n) = \Theta(\log n)$
4. For any fixed constant k, $\log n^k = \Theta(\log n)$
5. Any constant function is $O(1)$, $\Omega(1)$, and $\Theta(1)$
6. $2^n = \Theta(2^{n+1})$, this is an example of *many* functions that satisfy $f(n) = \Theta(f(n+1))$

More examples

1. Consider the series $\sum_{j=1}^{n} \log j$, clearly:

$$\sum_{j=1}^{n} \log j \leq \sum_{j=1}^{n} \log n = O(n \log n)$$

2. We also prove that $\sum_{j=1}^{n} \log j = \Omega(n \log n)$, therefore:

$$\sum_{j=1}^{n} \log j = \Theta(n \log n)$$

Problems

True or False?

1. $n^3 = O(n^2)$
2. $n \log n = O(n\sqrt{n})$
3. $n^2(1+\sqrt{n}) = O(n^2 \log n)$
4. $\log n + \sqrt{n} = O(n)$
5. $\sqrt{n} \log n = O(n)$
6. $\log n = O(1/n)$
7. $n + \sqrt{n} = O(\sqrt{n} \log n)$

Prove/disprove

1. $n^2 - 3n - 18 = \Omega(n)$
2. $2^n = O(n!)$
3. $n! = \Omega(2^n)$
4. $n^{\log n} = O((\log n)^n)$
5. $2^{(1+O(1/n))^2} = 2 + O(1/n)$

Searching

Linear Search

- Suppose we have a sequence of n numbers $X = \langle x_1, x_2, \ldots, x_n \rangle$, $n \geq 1$
- And we want *to search* for an element x in X
- A simple *search* algorithm is to scan the sequence X from left to right, and return the index of the first element x_i such that $x_i = x$, if such an element exists, and 0 otherwise
- Clearly, for a sequence of length n, the worst-case running time of this algorithm is $\Theta(n)$
- This algorithm is called LINEARSEARCH
- **Exercise:** Implement LINEARSEARCH in your favorite programming language
- **Exercise:** What is the average-case running time of LINEARSEARCH?

Searching in Sorted Data

- Often, we need to search for an element in a sorted sequence
- That is, we are provided with a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$ such that $x_1 \leq x_2 \leq \cdots \leq x_n$
- So, searching for an element x in X can be bit easier if we exploit the nature of the data
- For example, consider the following sequence $X = \langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \rangle$
- Suppose we want to search for $x = 7$ in X
- We can start from the middle of the sequence, i.e., $x_5 = 5$
- Since $x_5 < 7$, we can ignore the first half of the sequence, i.e., x_1, x_2, x_3, x_4, x_5
- This significantly reduces the *search space* for all possible places where x can be in X

A Not So Simple Searching Algorithm (Binary search)

Algorithm: BINARYSEARCH

Input: A sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$, $n \geq 1$, s.t. $x_1 \leq x_2 \leq \cdots \leq x_n$, and an element x to search for

Output: Returns j, if $x = x_j$, $1 \leq j \leq n$, and 0 otherwise

1. $l \leftarrow 1$, $h \leftarrow n$, $j \leftarrow 0$
2. **while** $(l \leq h)$ **and** $(j = 0)$
3. $\quad m \leftarrow \lfloor (l + h)/2 \rfloor$
4. $\quad$ **if** $x = x_m$ **then** $j \leftarrow m$
5. $\quad$ **else if** $x < x_m$ **then** $h \leftarrow m - 1$
6. $\quad$ **else** $l \leftarrow m + 1$
7. **return** j

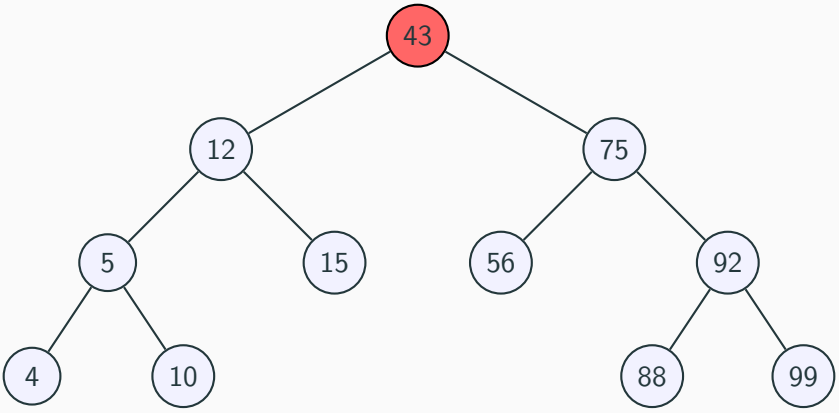
Analysis of Binary Search Algorithm

- Suppose the sequence X has n elements, $n \geq 1$
- And we are searching for x s.t. x is at least the max element in X
- In this case, after the first iteration $\lfloor n/2 \rfloor$ elements will remain in X
- After the second iteration, $\lfloor \lfloor n/2 \rfloor /2 \rfloor = \lfloor n/2^2 \rfloor$ elements will remain in X
- Suppose we have k iterations, there is only 1 element in X then

$$\lfloor n/2^k \rfloor = 1 \implies 1 \leq n/2^{k-1} < 2 \implies 2^{k-1} \leq n < 2^k \implies k - 1 \leq \log_2 n < k$$

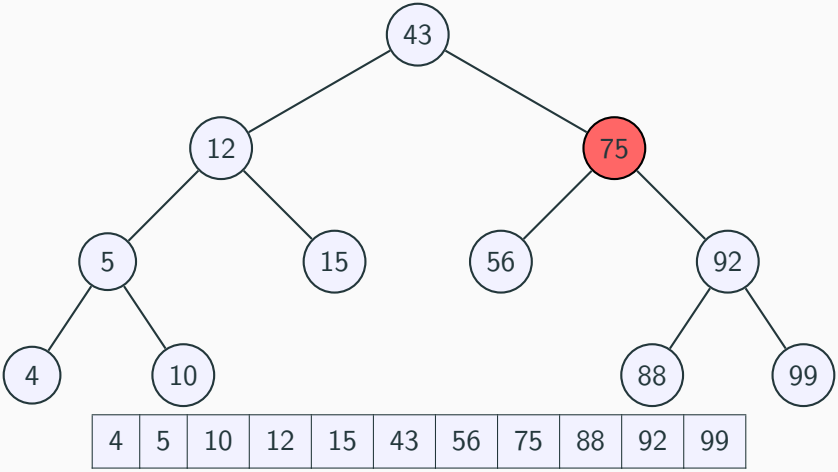
- So, the number of iterations is $\Theta(\log n)$

Binary Search Visualization (searching for 100)



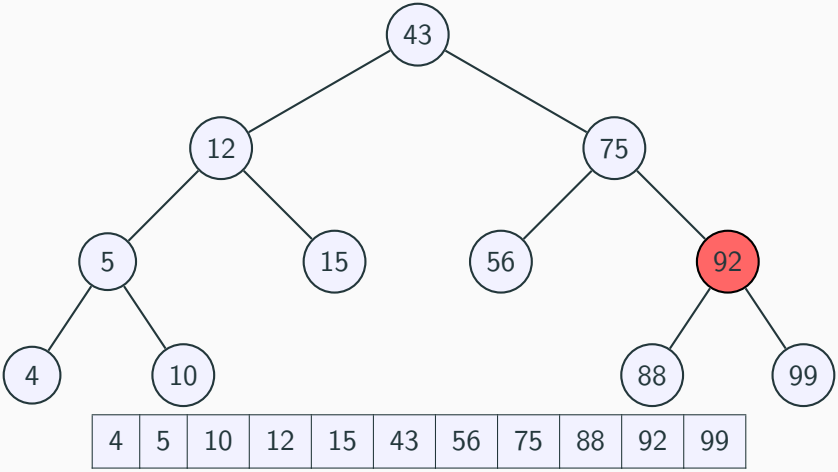
1-st iteration: no. of comparisons: 1

Binary Search Visualization (searching for 100)



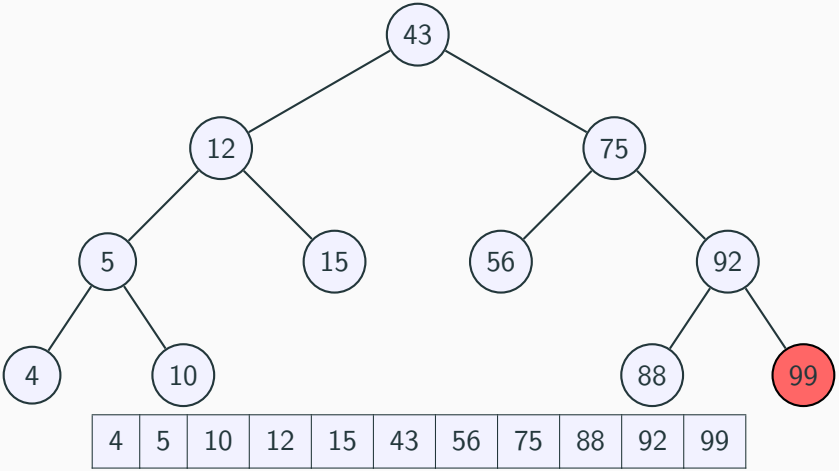
2-nd iteration: no. of comparisons: 2

Binary Search Visualization (searching for 100)



3-rd iteration: no. of comparisons: 3

Binary Search Visualization (searching for 100)



4-th iteration: no. of comparisons: 4

Sorting

Sorting

- Sorting is one of *the most fundamental problems* in computer science
- It is used in many applications
- For example, in a compiler, we may want *to sort* the variables by their scope
- In a spreadsheet, we may want *to sort* the rows by the value of a column, etc.

A Naïve Sorting (Selection Sort)

- Suppose we have a sequence of n numbers $X = \langle x_1, x_2, \ldots, x_n \rangle$, $n \geq 1$
- And we want *to sort* this sequence in *ascending order*
- That is, we want to rearrange the elements of X such that $x_1 \leq x_2 \leq \cdots \leq x_n$
- A simple *sorting* algorithm is to scan the sequence X from left to right, and find the smallest element x_i in X
- Then we swap x_1 with x_i
- Then we scan the sequence X from left to right, and find the smallest element x_j in X except x_1
- Then we swap x_2 with x_j
- We continue this process until the sequence X is sorted

Naïve Sorting Visualization

$$\langle 5, 2, 4, 6, 1, 3 \rangle$$

$$\langle 5, 2, 4, 6, \boxed{1}, 3 \rangle \implies \langle 1, 2, 4, 6, 5, 3 \rangle, \quad \text{comparisons: } 5, \text{swaps: } 1$$

$$\langle 1, \boxed{2}, 4, 6, 5, 3 \rangle \implies \langle 1, 2, 4, 6, 5, 3 \rangle, \quad \text{comparisons: } 4, \text{swaps: } 1$$

$$\langle 1, 2, 4, 6, 5, \boxed{3} \rangle \implies \langle 1, 2, 3, 6, 5, 4 \rangle, \quad \text{comparisons: } 3, \text{swaps: } 1$$

$$\langle 1, 2, 3, 6, 5, \boxed{4} \rangle \implies \langle 1, 2, 3, 4, 5, 6 \rangle, \quad \text{comparisons: } 2, \text{swaps: } 1$$

$$\langle 1, 2, 3, 4, \boxed{5}, 6 \rangle \implies \langle 1, 2, 3, 4, 5, 6 \rangle, \quad \text{comparisons: } 1, \text{swaps: } 1$$

Insertion Sort (with Binary Search)

- There are better sorting algorithms than SELECTIONSORT
- Consider the following algorithm
- Suppose we want to sort sequence of n numbers $X = \langle x_1, x_2, \ldots, x_n \rangle$, $n \geq 1$
- We start with an empty sequence Y
- We scan the sequence X from left to right, and insert each element $x_i \in X$ into Y such that Y remains sorted
- We can use modified BINARYSEARCH to find the position of x_i in Y
- This algorithm is called INSERTIONSORT
- **Exercise:** Implement INSERTIONSORT in your favorite programming language

Insertion Sort

Algorithm: INSERTIONSORT

Input: A sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$, $n \geq 1$

Output: A sequence $Y = \langle y_1, y_2, \ldots, y_n \rangle$, $n \geq 1$, such that $y_1 \leq y_2 \leq \cdots \leq y_n$

1. $Y \leftarrow \langle\rangle$
2. **for** $i \leftarrow 1$ **to** n
3. $\quad j \leftarrow \text{BINARYSEARCH}(Y, x_i)$
4. $\quad Y \leftarrow \text{INSERT}(Y, j, x_i)$
5. **return** Y

Time complexity: Above algorithm runs in $\Theta(n \log n)$ time

Theoretical Lower Bound for Sorting

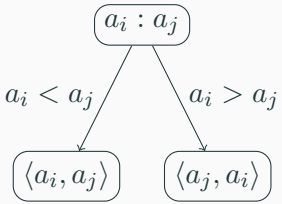
- Most sorting algorithms we will see in this class are *comparison based* sorting algorithms
- That is, we *compare* elements of the given sequence to find their relative order
- Suppose we have sequence of n elements, then there are $n!$ different ways to arrange these elements
- However, only of these $n!$ arrangements is sorted in ascending order
- We can use a *decision tree* model to find a lower bound on the number of comparisons required to sort a sequence of n elements

Lower Bound on Comparison-Based Sorting Algorithms

- **Input:** a sequence of elements $\langle a_1, a_2, \ldots, a_n \rangle$ from some totally ordered set (assuming all elements are distinct for simplicity)
- **Output:** a permutation $\langle a_{i_1}, a_{i_2}, \ldots, a_{i_n} \rangle$ of the input sequence such that $a_{i_1} < a_{i_2} < \cdots < a_{i_n}$
- Let us denote by P_n all permutations of numbers $\{1, 2, \ldots, n\}$
- In comparison-based sorting algorithm; we compare two elements $a_i : a_j$
- There are two scenarios: either $a_i < a_j$ or $a_i > a_j$

Lower Bound on Comparison-Based Sorting Algorithms

- Consider a decision tree T with $n!$ leaves (terminal nodes), each leaf corresponds to a permutation of the input sequence
- Each nonterminal node of T is labeled with a pair of elements $a_i : a_j$, $1 \leq i, j \leq n$, $i \neq j$



Lower Bound on Comparison-Based Sorting Algorithms

Lemma

For any rooted binary tree D with n leaves, the height of D is at least $\log_2 n$.

- Let $\mathcal{A}$ be an arbitrary comparison-based sorting algorithm to sort $\langle a_1, a_2, \ldots, a_n \rangle$
- Let $T_{\mathcal{A}}$ be the decision tree corresponding to $\mathcal{A}$
- Since $T_{\mathcal{A}}$ has $n!$ leaves, by the above lemma, the height of $T_{\mathcal{A}}$ is at least $\log_2 n! = \Theta(n \log n)$
- Therefore, $\mathcal{A}$ must do at least $\Omega(n \log n)$ comparisons