Sorting Lower Bounds
Linear sorting

CLRS 8.1 – 8.4
(+ some supplemental material)

Comparison-based sorting

- Recall – Sorting
  - input: A sequence of n values $x_1, x_2, \dots, x_n$
  - output: A permutation $y_1, y_2, \dots, y_n$ such that $y_1 \leq y_2 \leq \cdots \leq y_n$

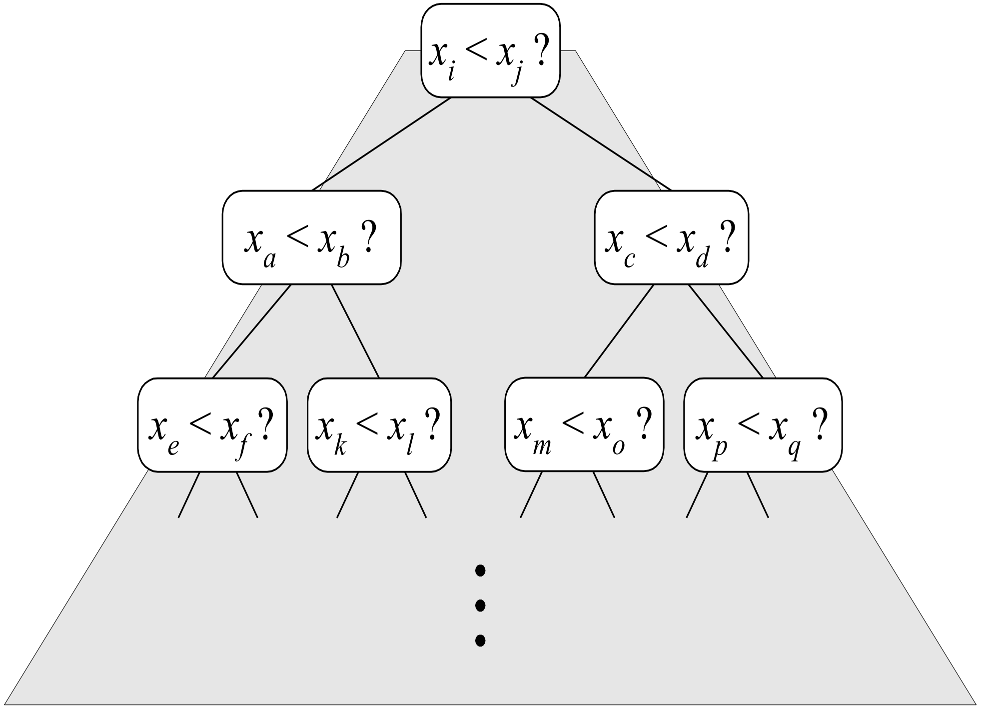
- **Many algorithms are comparison based**
  - they sort by making comparisons between pairs of objects
  - ex: selection-sort, insertion-sort, heap-sort, merge-sort, quick-sort, …
  - best so far runs in $\boldsymbol{O(n \log n)}$ time… can we do better?

- Let's derive a lower bound on the running time of any algorithm that uses comparisons to sort n elements

Counting comparisons

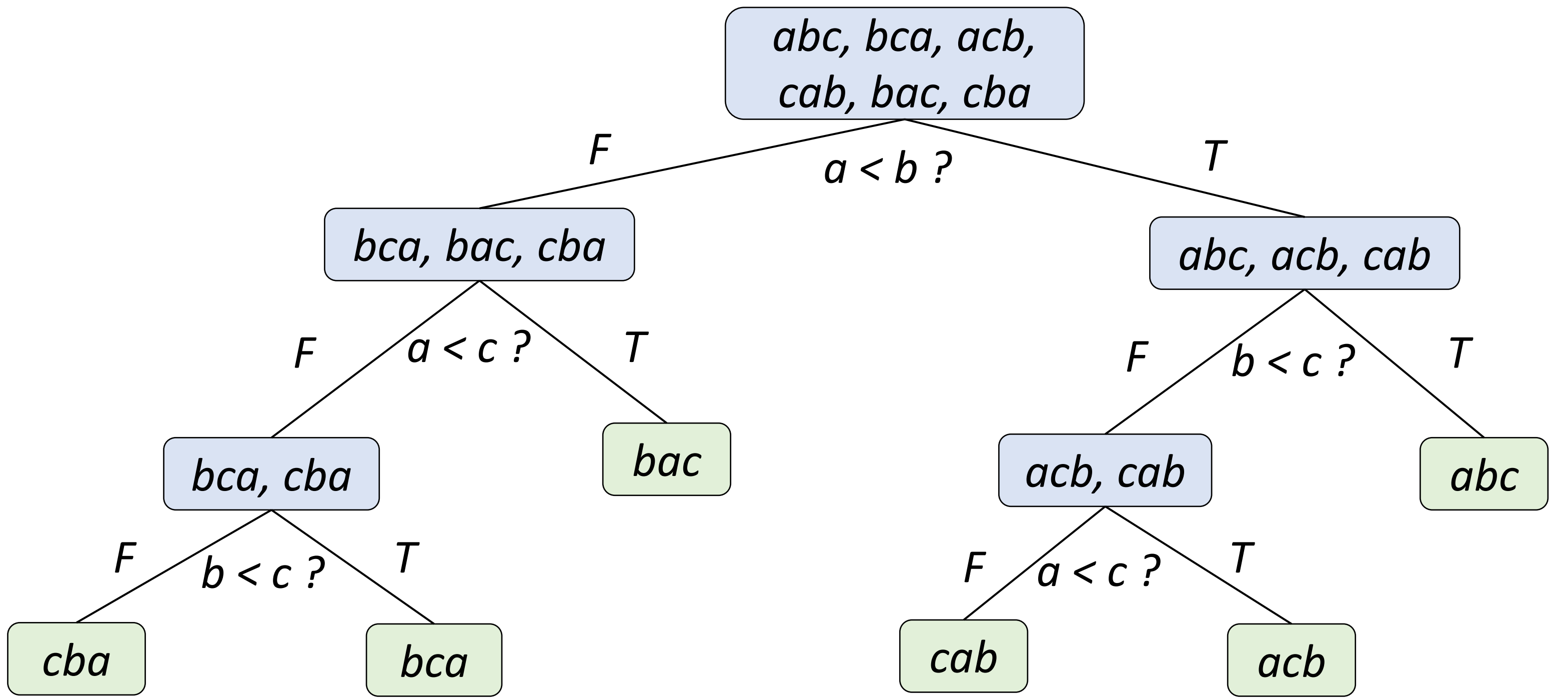
A decision tree represents every sequence of comparisons that an algorithm might make on an input of size n

- **each possible run of the algorithm corresponds to a root-to-leaf path**
- at each internal node a comparison $x_i < x_j$ is performed and branching made
- nodes annotated with the orderings consistent with the comparisons made so far
- leaf contains result of computation (a total order of elements)



Decision Tree Example

- Algorithm: insertion sort
- Instance (n = 3): the numbers a, b, c



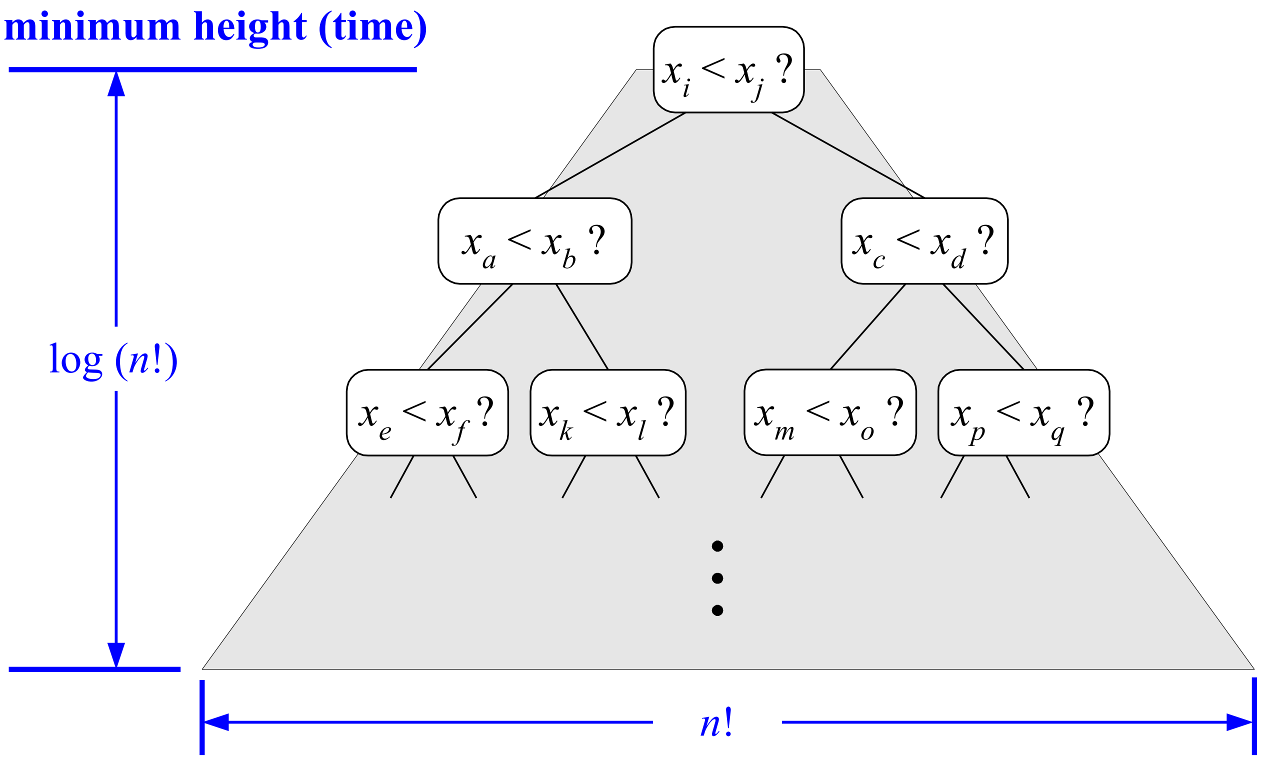
Height of Decision Tree

Theorem: Any decision tree sorting n elements has height $\Omega(n \log n)$.

Proof: There are $n!$ leaves. A binary tree of height h has at most 2^h leaves. So

$$\begin{aligned} 2^h &\geq n! \\ h &\geq \log(n!) \\ &\geq c \cdot \log(n^n) \\ &= c \cdot n \log(n) \end{aligned}$$

Thus, $h \in \Omega(n \log n)$.



Corollary: Any sorting algorithm that uses only comparisons takes $\Omega(n \log n)$ in the worst case.

Linear time sorting

Any comparison-based sorting algorithm runs in $\Omega(n \log n)$ time in the worst case.

To achieve linear-time sorting of *n* elements:

- (!!!) Assume **keys** are **integers** in the **range** $[\mathbf{0}, \boldsymbol{k}]$
- We can use other operations instead of comparisons
- We can sort in linear time when k is small enough
  - Note: we cannot assume this for just any problem with integers !!!

Some sorting algorithms which are **not** comparison-based

- Counting sort
- Radix sort
- Bucket sort

Counting sort

Input: array $A[1 \dots n]$ of integers, each in the range $[0, k]$

A: 8, 3, 4, 8, 12, 20, 8 …..

Main idea: Use a **counting** array **C**

First, store frequency of integers in A at their matching index in C

- Ex: $\boldsymbol{C}[\boldsymbol{i}] = \boldsymbol{x}$ if there are $\boldsymbol{x}$ total elements in $\boldsymbol{A}$ with value **equal to** $\boldsymbol{i}$

C	0	0	0	1	1	0	0	0	3	0	0	
	0	1	2	3	4	5	6	7	8	9	10	…. k

Next, combine so that C stores sorted **rank** (number of items before) at indices

- Ex: $\boldsymbol{C}[\boldsymbol{i}] = \boldsymbol{x}$ if there are $\boldsymbol{x}$ total elements in $\boldsymbol{A}$ with value **less than or equal to** $\boldsymbol{i}$

C	0	0	0	1	2	2	2	2	5	5	5	
	0	1	2	3	4	5	6	7	8	9	10	…. k

Build a **sorted array B** which will be returned

- Use rank to determine where the element belongs in B
- For each integer $\boldsymbol{a} \in A$, its rank in $\boldsymbol{B}$ is $\boldsymbol{C}[\boldsymbol{a}]$ … so put $\boldsymbol{a}$ at location $\boldsymbol{B}[\boldsymbol{C}[\boldsymbol{a}]]$
- Decrease $\boldsymbol{C}[\boldsymbol{a}]$ to update rank of future duplicate values

B	3	4	8	8	8	….				
	1	2	3	4	5	6	7	8	….	n

Counting sort

Input: array $A[1 \ldots n]$ of integers, each in the range $[0, k]$

Q: How efficient is this?

Q: Is it in-place?

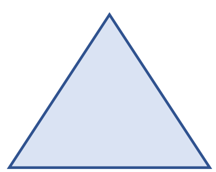
```
COUNTING-SORT(A, n, k)
let B[1 : n] and C[0 : k] be new arrays
for i = 0 to k
    C[i] = 0
for j = 1 to n
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
// Copy A to B, starting from the end of A.
for j = n downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1    // to handle duplicate values
return B
```

https://algorithm-visualizer.org/divide-and-conquer/counting-sort

Notable properties of counting sort

- **Run time:**
  - $\boldsymbol{O(n+k)}$
  - $\boldsymbol{O(n)}$ **when** $\boldsymbol{k = O(n)}$
  - Ex: if all integers are in the range $[0, 100n]$, then counting sort is $O(n)$
  - Ex: if all integers are in the range $[0, n^5]$, then counting sort is $O(n^5)$
  - Ex: if all integers are in the range $[0, 2^n]$, then counting sort is $O(2^n)$
- It is **stable**: numbers with the same value appear in the output array in the same order as they do in the input array
  - Important when we are **sorting multiple times** based on different attributes
  - Ex: sort a list of names by first name, then sort by last name
- Ex: Unsorted sequence (**B**, **b**, a, c). Suppose B = b and a < b < c.
  - Stable sorted: (a, **B, b**, c)
  - Unstable sorted: (a, **b, B**, c)

In general, we can choose two:



Radix sort

Input: array $A[1 \dots n]$ of n integers where

- each integer is represented as d keys: $\boldsymbol{x_d x_{d-1} \dots x_2 x_1}$
- $\boldsymbol{x_d}$ is the most significant key/dimension; $\boldsymbol{x_1}$ is the least significant key/dimension
- all $\boldsymbol{nd}$ keys are in the range $[\mathbf{0}, \boldsymbol{k}]$

```
RADIX-SORT(A, d)
for i = 1 to d
    use a stable sort to sort array A on digit i
```

Here, we represent an integer key in base 10 (so, all keys are in the range [0,9].
In this case, $d = 3$.

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Notable properties of radix sort

- **Run time:**
  - $\boldsymbol{O(d(n+k))}$
  - $O(n)$ when d is constant and $k = O(n)$

- In the last example, we represented integers in base 10.
  - Suppose our maximum integer is N, and $N = O(n^c)$ for a constant c
  - Then the number of keys needed for each integer is $\boldsymbol{d = \log_{10} N = O(\log n)}$
  - Total run time: $O(n \log n)$

- We can do better!! What if we represented integer keys in base n?
  - Suppose our maximum integer is N, and $N = O(n^c)$ for a constant c
  - Then the number of keys needed for each integer is $\boldsymbol{d = log_n N = c = O(1)}$
  - Total run time: $O(n)$

- Ex: if all integers are in the range $[0, n^3]$, then representing the integers in base n allows radix sort to run in $O(n)$ time