All of the sorting algorithms we have considered so far have been **comparison sorts**. That is, we only use comparisons to determine the relative order of elements.

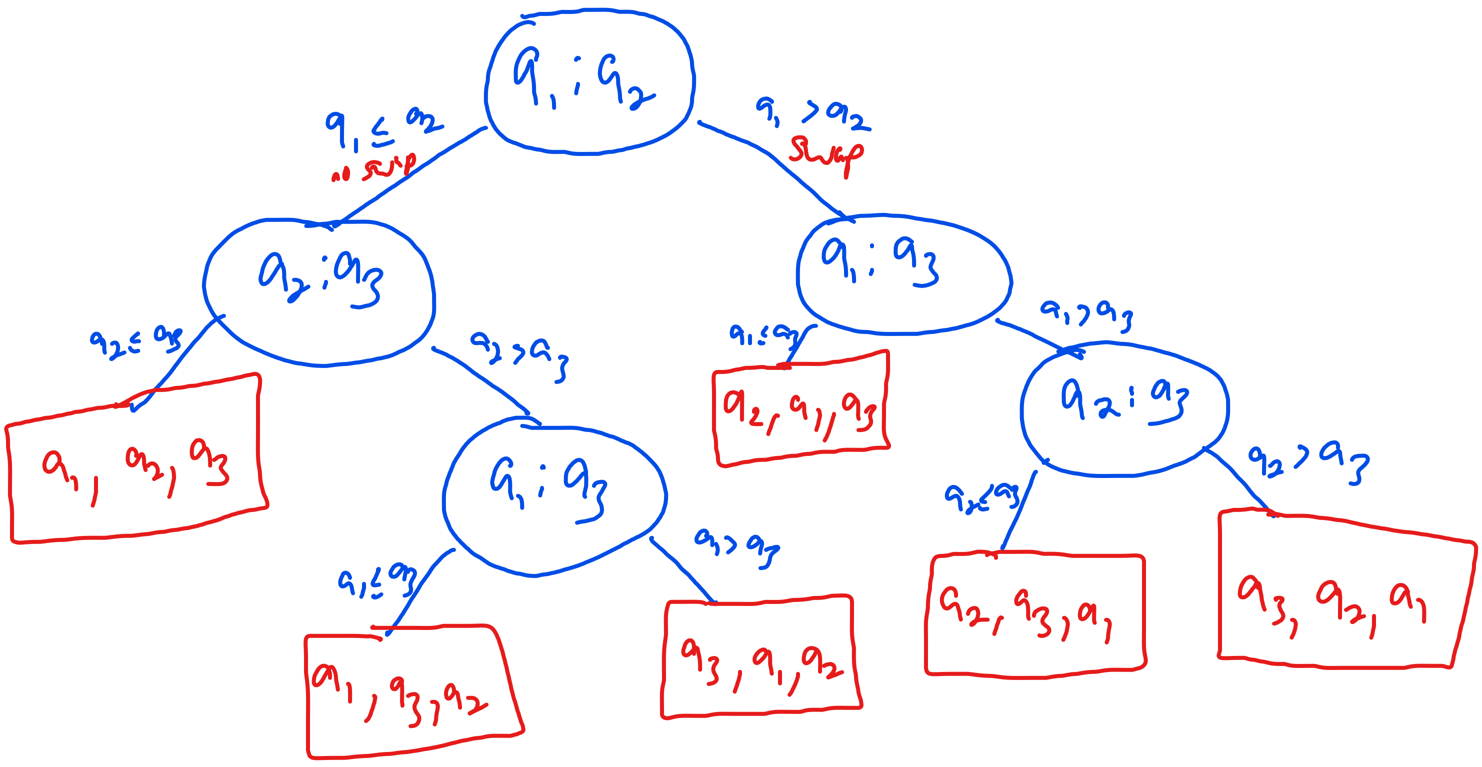
The best algorithm we have seen in terms of worst-case running time is $O(n \log n)$.

Can we do better? That is, is it possible to design a comparison-based sorting algorithm which has worst-case running time $o(n \log n)$ (e.g. $O(n)$ or $O(n \log \log n)$)?

We will use a **decision tree** to help us answer this question.

Insertion Sort

$n=3$, we want to sort $[a_1, a_2, a_3]$



We can construct a decision tree which models the execution of any comparison sorting algorithm.

- One tree per input size n.
- The tree will contain *all* possible comparisons that could be executed for *any* input of size n.
- For one input, only one path to a leaf is executed.
- The running time is equal to the length of the path taken.
- Worst-case running time is the length of the longest path in the tree (the height of the tree).

Theorem: Any decision tree of a comparison sorting algorithm on an input of size n must have height $\Omega(n \log n)$.

There must be $n!$ leaves.

The tree is a binary tree because comparing is a binary operation. A binary tree of height h has at most 2^h leaves.

$\Rightarrow 2^h \geq n! \Rightarrow h \geq \log_2 n!$

$$n! = \underbrace{n(n-1)(n-2)\cdots\left(\tfrac{n}{2}\right)}\underbrace{\left(\tfrac{n}{2}-1\right)\cdots 1}$$

$$\geq \left(\tfrac{n}{2}\right)\left(\tfrac{n}{2}\right)\left(\tfrac{n}{2}\right)\cdots\left(\tfrac{n}{2}\right) 1 \cdot 1 \cdots\cdots 1 = \left(\tfrac{n}{2}\right)^{\left(\frac{n}{2}\right)}$$

$$\log_2 n! \geq \log_2 \left(\tfrac{n}{2}\right)^{\left(\frac{n}{2}\right)} = \tfrac{n}{2}\log_2 \tfrac{n}{2} = \Theta(n \log n)$$

$\log a^b = b \log a$

Recall we wanted to answer the following question.

- The best algorithm we have seen in terms of worst-case running time is $O(n \log n)$. Can we do better? That is, is it possible to design a comparison-based sorting algorithm which has worst-case running time $o(n \log n)$ (e.g. $O(n)$ or $O(n \log \log n)$)?

The theorem we proved implies that the answer to this question is no.

Corollary: Merge Sort is an asymptotically optimal comparison sorting algorithm.

Can we do better if we do $\overset{\text{not}}{\vee}$ use comparisons in our algorithm? Sometimes!

Counting Sort:

- Input: $A[1 \ldots \underline{n}]$, where each $A[j] \in \{1, 2, \ldots, \underline{k}\}$.
- Output: $\underline{B[1 \ldots n]}$, sorted.
- Auxiliary storage: $\underline{C[1 \ldots k]}$.

30,000 UTSA exam,
Each score is an
int between 0 & 100

Counting Sort

```
1. for $i=1$ to $k$
       $C[i]=0$                      } $O(k)$

2. for $j=1$ to $n$
       $C[A[j]]{+}{+}$               } $O(n)$
```

// $C[i]$ is how many times i occurs

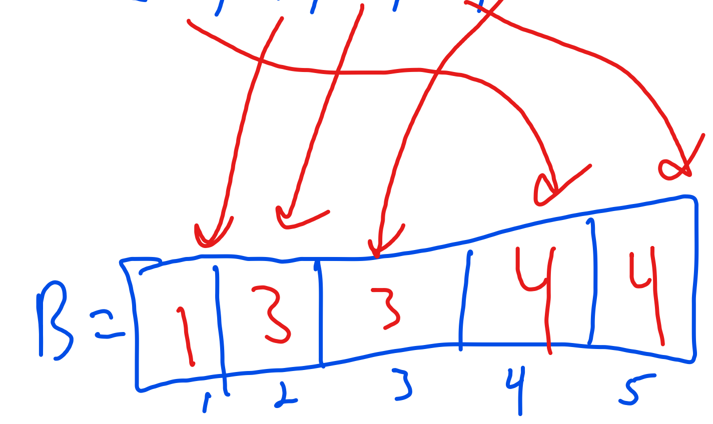
```
3. for $i=2$ to $k$
       $C[i] = C[i] + C[i-1]$        } $O(k)$
```

// $C[i]$ is how many times $\leq i$ occurs.

```
4. for $j=n$ to $1$
       $B[C[A[j]]] = A[j]$
       $C[A[j]]{-}{-}$               } $O(n)$
```

Running time: $O(n+k)$

$A = [4, 1, 3, 4, 3]$



$C : [1, 0, 2, 2]$

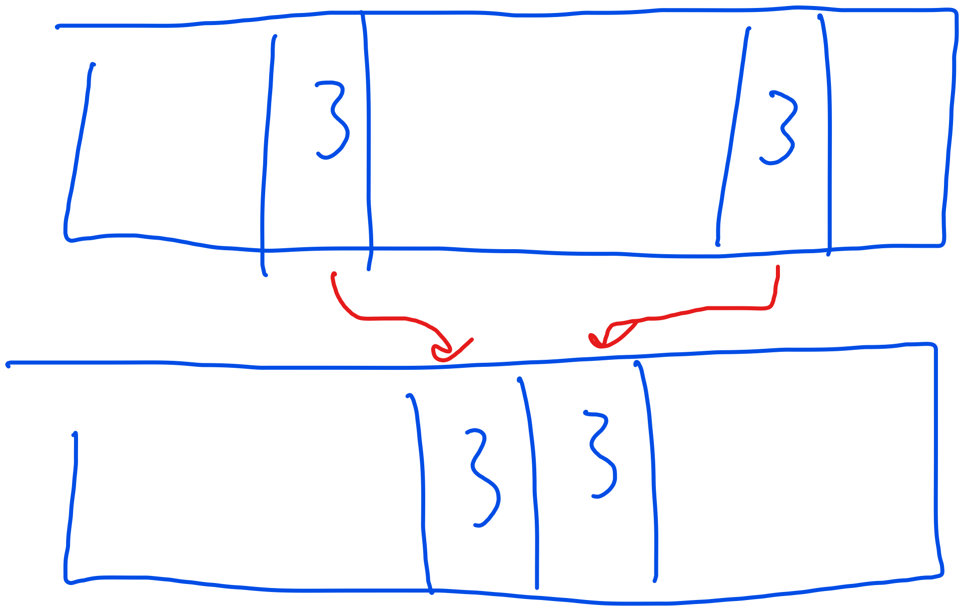
$$C' : \begin{array}{cccc} \overset{0}{\cancel{1}}, & 1, & \overset{\cancel{2}\ 1}{\cancel{3}}, & \overset{\cancel{4}\ 3}{\cancel{5}} \\ 1 & 2 & 3 & 4 \end{array}$$

Note that we did not ever directly compare any two elements of A.

Also note that if $k = O(n)$, then counting sort takes $\Theta(n)$ time (which is better than the $\Omega(n \log n)$ lower bound of comparison sorting algorithms.

In the worst case, k can be arbitrarily large, and therefore the running time of the algorithm can be arbitrarily large (with respect to n).

Counting sort is a **stable** sort. That is, if the same value appears multiple times, then their input order is preserved in the output array.



Radix Sort:

- Digit-by-digit sort.
- Idea: sort the input on the *least-significant digit* first (i.e. digits from right to left) using a stable sorting algorithm (like counting sort).

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
↑	↑	↑	

3 calls to counting sort, each is $O(n+10)$

$\Rightarrow O(n)$ overall.

Note that we can convert a decimal number to any other base (i.e. binary or hexadecimal) and sort these numbers instead (sorted order of binary/hex will be the same as sorted order for decimal).

Note that if we convert the number to binary, then the number of digits for radix sort increases, but the range of values for counting sort decreases ($k = 2$).

Note that if we convert the number to hexadecimal, then the number of digits for radix sort decreases, but the range of values for counting sort increases ($k = 16$).

What base should we use to minimize the running time of the algorithm?

We can view the problem as sorting n binary numbers of b bits each.

Consider some positive integer $r \leq b$. We can view each number as having b/r digits in the range $[0, 2^r - 1]$. Then the algorithm would make b/r calls to counting sort with $k = 2^r$.

What choice of r should we make to minimize the running time of the algorithm.

Counting sort takes $\Theta(n+k)$ time, and our $k = 2^r$ which implies each call to counting sort takes time $\Theta(n + 2^r)$.

Since there are b/r calls to counting sort, we have the overall running time is $\Theta(\frac{b}{r}(n+2^r))$. What choice of r minimizes this function?

So with $r = \log n$, we get that the running time of radix sort when the input is n numbers of b bits each is $\Theta(\frac{bn}{\log n})$.

Therefore, if the range of values is a constant (and therefore b is a constant), then the algorithm runs in $o(n)$ time.

If the range of values is from 0 to $n^d - 1$, then $b = d \log n$. The algorithm would have running time $\Theta(dn)$.

If the range of values is arbitrarily large, then the running time of radix sort is arbitrarily large.