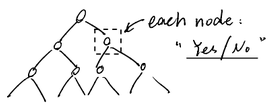
Sorting

if: Comparison-Based ⇒ Optimally O(nlogn) time
(Merge Sort / Heap Sort)

Decision Tree



∴ Binary Tree with n nodes has height $\Omega(\log n)$

Decision Tree for Binary Search

Since Binary Search has (n+1) outputs (n elements + 1 "failure")

∴ Decision tree has at least (n+1) nodes.

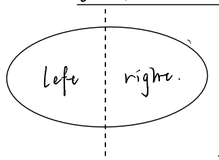
⇒ Height: $\Omega(\log(n+1)) = \Omega(\log n)$

∴ BS algorithm has $\Omega(\log n)$ running time.

"Decision Tree's height is related to # outputs"

Example: n coins., at most, 1 coins is heavier/lighter.

We wanna to find that coin



Algorithm: Split coins into 3 sets: C_1, C_2, C_3.

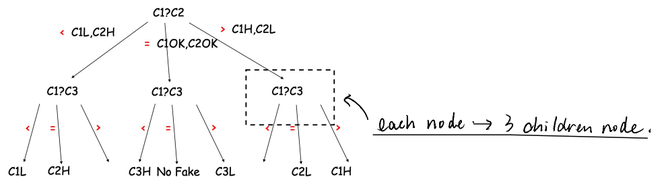
With 2 comparisons We have 7 possible outcomes:

C_1H / C_1L, C_2H / C_2L, C_3H / C_3L, Same

if Same: Stop and return none. (No defective coins)

else: recursively call algorithm on the set containing defective coin

Running Time?



each node → 3 children node.

Decision Tree has (2n+1) possible outcomes

(i.e. each coin is either "heavier", "lighter", or "normal"

⇒ Decision Tree has height of at least $\log_3(2n+1)$

∴ "At least" $\log_3(2n+1)$ Comparisons

Actually, since we use 2 comparison ⇒ reduce the size of problem by 1/3.

∴ $T(n) = T(n/3) + 2 = 2\log_3 n$

Back to "Comparison-Based Sorting"

for n-element array. "Decision Tree" has (n!) possible one put.

∴ Height is at least $\log(n!) = \Theta(n\log n)$.

∴ running time: $T(n) = \Omega(n\log n)$

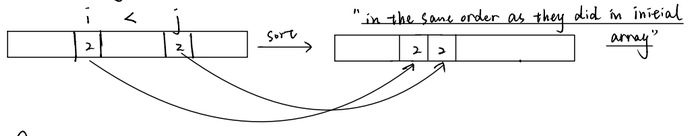
How can we do better?

non Comparison-based sorting algorithm

e.g. Counting Sort ⇒ sort array with element ranging from [1, k]

```
CountSort(A[1..n], B, k):
for i ← 1 to k do:
    C[i] ← 0.
for i ← 1 to n do:
    C[A[i]] ← C[A[i]] + 1
for i ← 2 to k do:
    C[i] ← C[i] + C[i-1]      ← "Prefix Sum"
for i ← n to 1 do:
    B[C[A[i]]] ← A[i]
    C[A[i]] ← C[A[i]] - 1
```

A sorting algorithm is "Stable" if



Counting Sort is Stable

- Running Time: $\Theta(n+k)$
- Space: $\Theta(n+k)$

Range Queries: "We want to know How many elements are there in range (a, b]"

```
range_query(A[1..n], a, b]:
    for i ← 1 to k do:
        C[i] ← 0.
    for i ← 1 to n do:
        C[A[i]] ← C[A[i]] + 1
    for i ← 2 to k do:
        C[i] ← C[i] + C[i-1]
    return C[b] - C[a]
```

← Same part as in "Counting Sort"

Radix Sort

2329	2720	2720	2329	2329
5457	5355	2329	5355	2720
3657	3436	3436	3436	3436
5839	5457	5839	5457	3657
3436	3657	5355	3657	5355
2720	2329	5457	2720	5457
5355	5839	3657	5839	5839

```
for i ← 1 to d
    use "counting sort" to sort array A on digit i
```

Correctness

proof by induction:

Assume the numbers are correctly sorted on low-order i-1 digits then:

On i digit:

1. if two numbers are different on digit i, they will be correctly sorted by i low-order digits.
2. if two numbers are same on digit i

Based on the "stability" of "Counting Sort" they are in the same order after running "Counting order" on digit i.

⇒ they are correctly sorted by their low-order digits i.

∴ The algorithm can correctly sort the numbers

running time $T(n) = \Theta(d(n+k))$

where: d: # digits of number; n: # integers; k: range

space required: $\Theta(n+k)$

	Insertion sort	Merge sort	Quicksort	Heapsort	Radix sort
Running time	$\Theta(n^2)$	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(d(n+k))$
Randomized	No	No	Yes	No	No
Working space	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(n+k)$
Comparison-based	Yes	Yes	Yes	Yes	No
Stable	Yes	Yes	No	No	Yes
Other Properties					
Cache performance	Good	Good	Good	Bad	Bad
Parallelization	No	Excellent	Good	No	No