Sorting Conclusions

Sorting So Far

Insertion sort:

- Easy to code
- Fast on small inputs (less than ~50 elements)
- Fast on nearly-sorted inputs
- $O(n^2)$ worst case
- $O(n^2)$ average (equally-likely inputs) case
- $O(n^2)$ reverse-sorted case

Sorting So Far

- Merge sort:
  - Divide-and-conquer:
    - Split array in half
    - Recursively sort subarrays
    - Linear-time merge step
  - O(n lg n) worst case
  - Doesn’t sort in place

Sorting So Far

- Heap sort:
  - Uses the very useful heap data structure
    - Complete binary tree
    - Heap property: parent key > children's keys
  - O(n lg n) worst case
  - Sorts in place
  - Fair amount of shuffling memory around

Sorting So Far

- Quick sort:
  - Divide-and-conquer:
    - Partition array into two subarrays, recursively sort
    - All of first subarray < all of second subarray
    - No merge step needed!
  - O(n lg n) average case
  - Fast in practice
  - O(n^2) worst case
    - worst case on sorted input
    - Address this with randomized quicksort

How Fast Can We Sort?

- First, an observation: all of the sorting algorithms so far are *comparison sorts*
  - The only operation used to gain ordering information about a sequence is the pairwise comparison of two elements
  - Comparisons sorts must do at least n comparisons *(why?)*
  - What do you think is the best comparison sort running time?

Comparison-based sorting

- Sorted order is determined based **only** on a comparison between input elements
  - A[i] < A[j]
  - A[i] > A[j]
  - A[i] = A[j]
  - A[i] ≤ A[j]
  - A[i] ≥ A[j]
- Do any of the sorting algorithms we've looked at use additional information?

Comparison-based sorting

- Sorted order is determined based **only** on a comparison between input elements
  - A[i] < A[j]
  - A[i] > A[j]
  - A[i] = A[j]
  - A[i] ≤ A[j]
  - A[i] ≥ A[j]
- Do any of the sorting algorithms we've looked at use additional information?
  - No
  - All the algorithms we've seen are comparison-based sorting algorithms

Comparison-based sorting

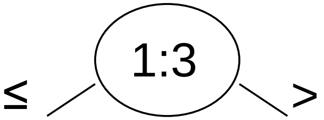
- Sorted order is determined based **only** on a comparison between input elements
  - A[i] < A[j]
  - A[i] > A[j]
  - A[i] = A[j]
  - A[i] ≤ A[j]
  - A[i] ≥ A[j]
- Can we do better than O(n log n) for comparison based sorting approaches?

Decision Trees

- Abstraction of any comparison sort.
- Represents comparisons made by
  - a specific sorting algorithm
  - on inputs of a given size.
- Abstracts away everything else: control and data movement.
  - We’re counting *only* comparisons.
- Each node is a pair of elements being compared
- Each edge is the result of the comparison (< or >=)
- Leaf nodes are the sorted array

Decision-tree model

- *Full* binary tree representing the comparisons between elements by a sorting algorithm
- Internal nodes contain indices to be compared



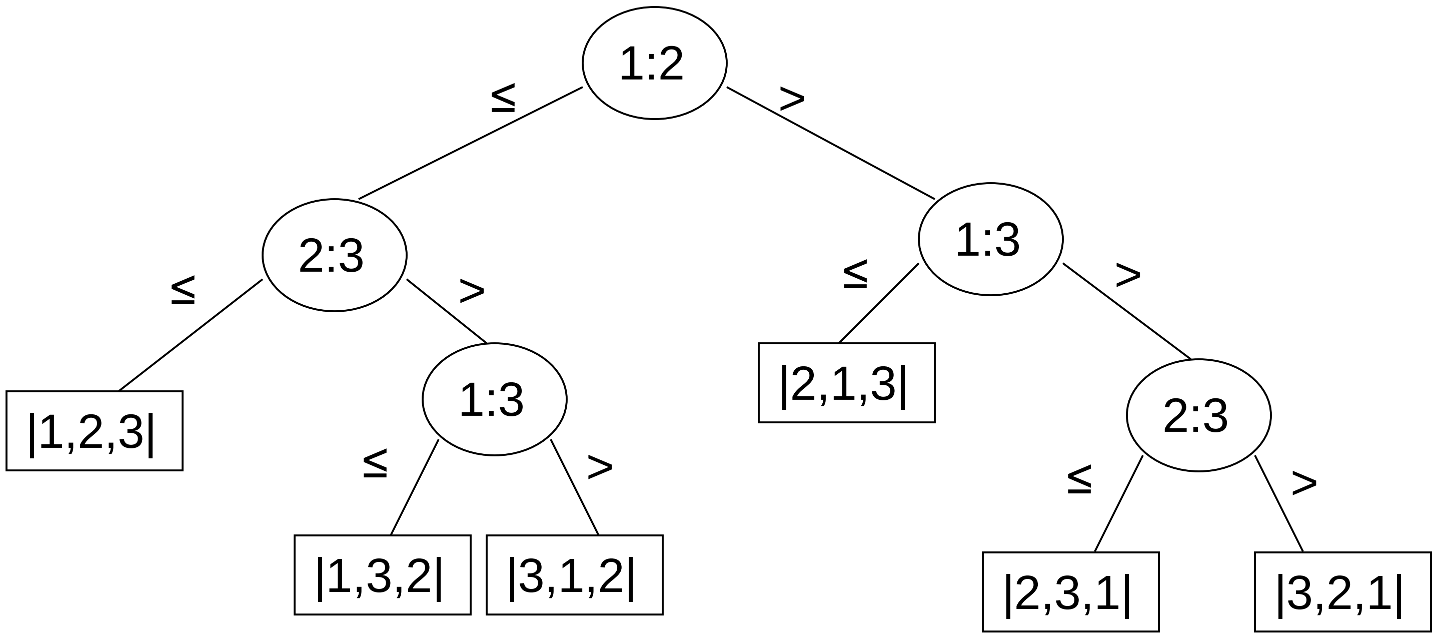
- Leaves contain a complete permutation of the input

[3, 12, 7] ⇨ | 1,3,2 | ⇨ [3, 7, 12]

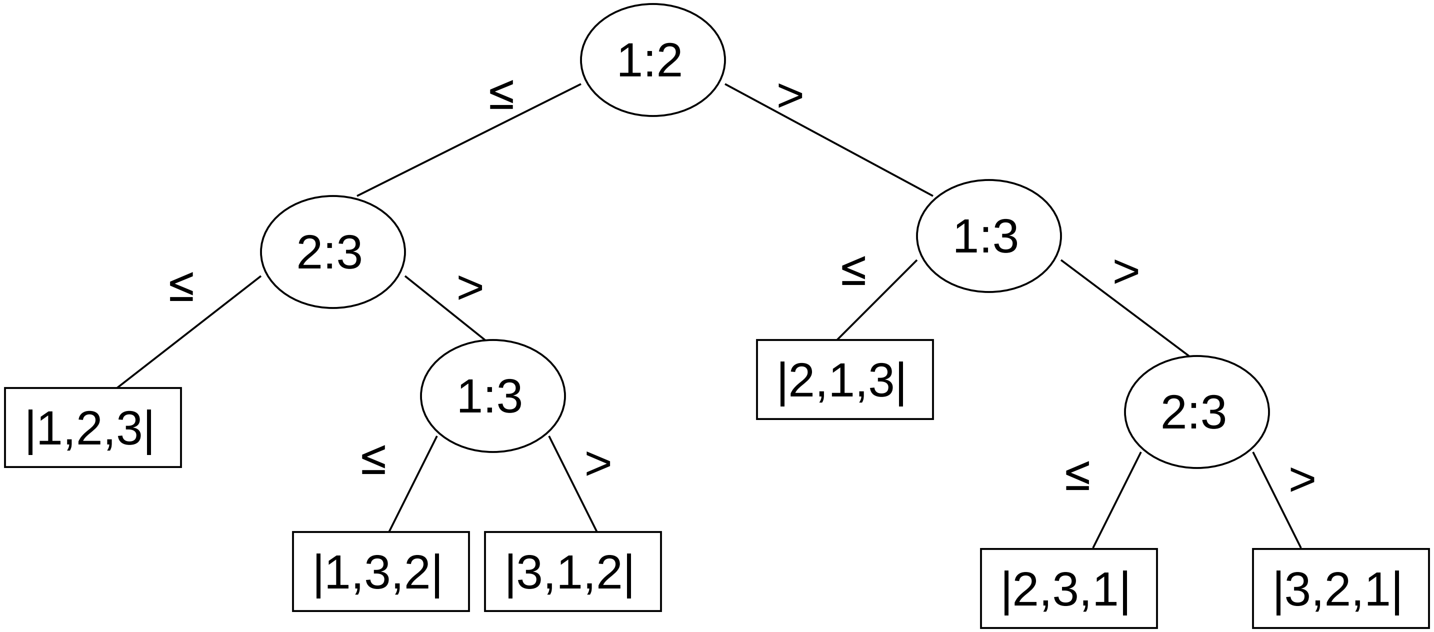
[7, 3, 12] ⇨ | 2,1,3 | ⇨ [3, 7, 12]

- Tracing a path from root to leave gives the correct reordering/permutation of the input for an input

A decision tree model

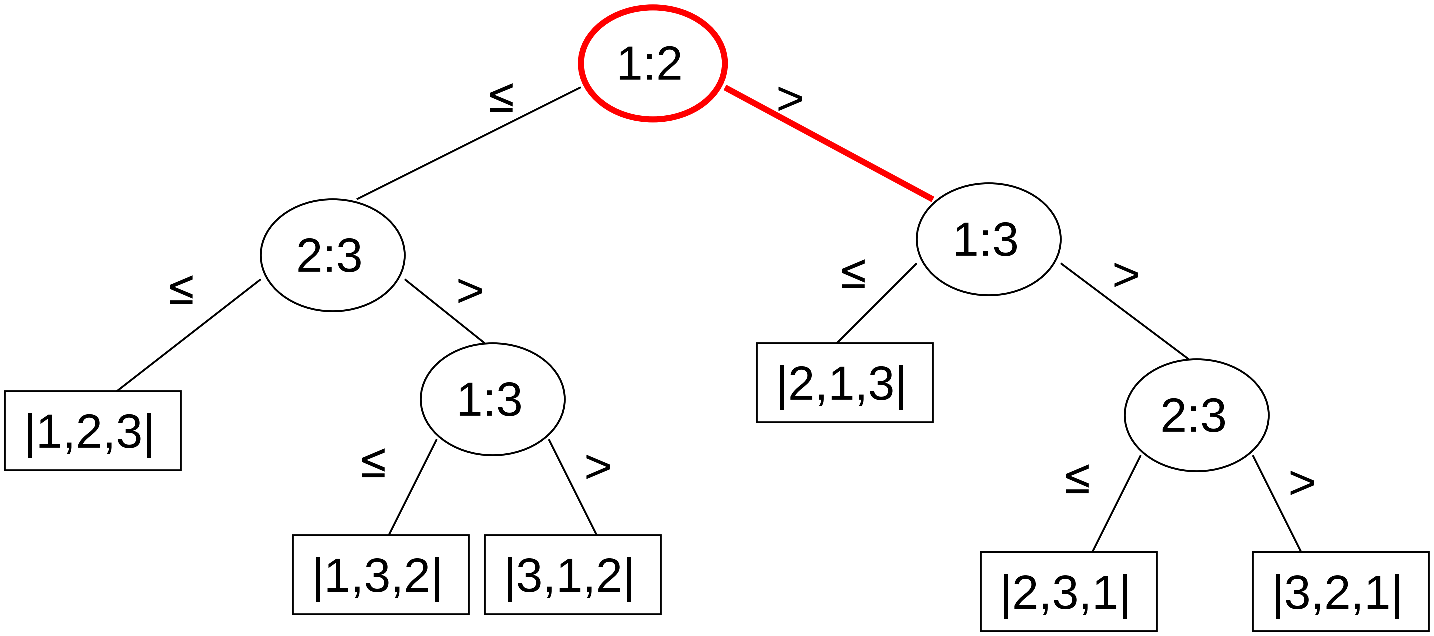


A decision tree model



[12, 7, 3]

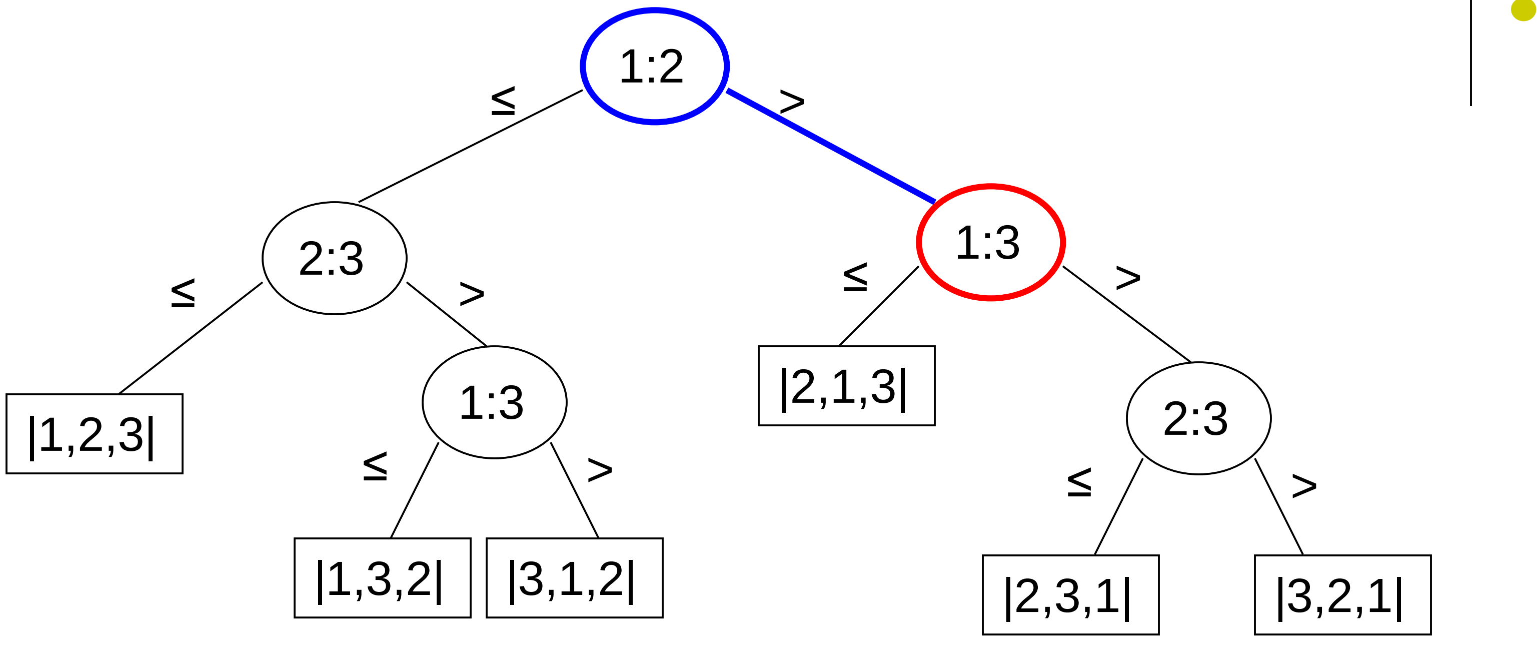
A decision tree model



[12, 7, 3]

Is 12 ≤ 7 or is 12 > 7?

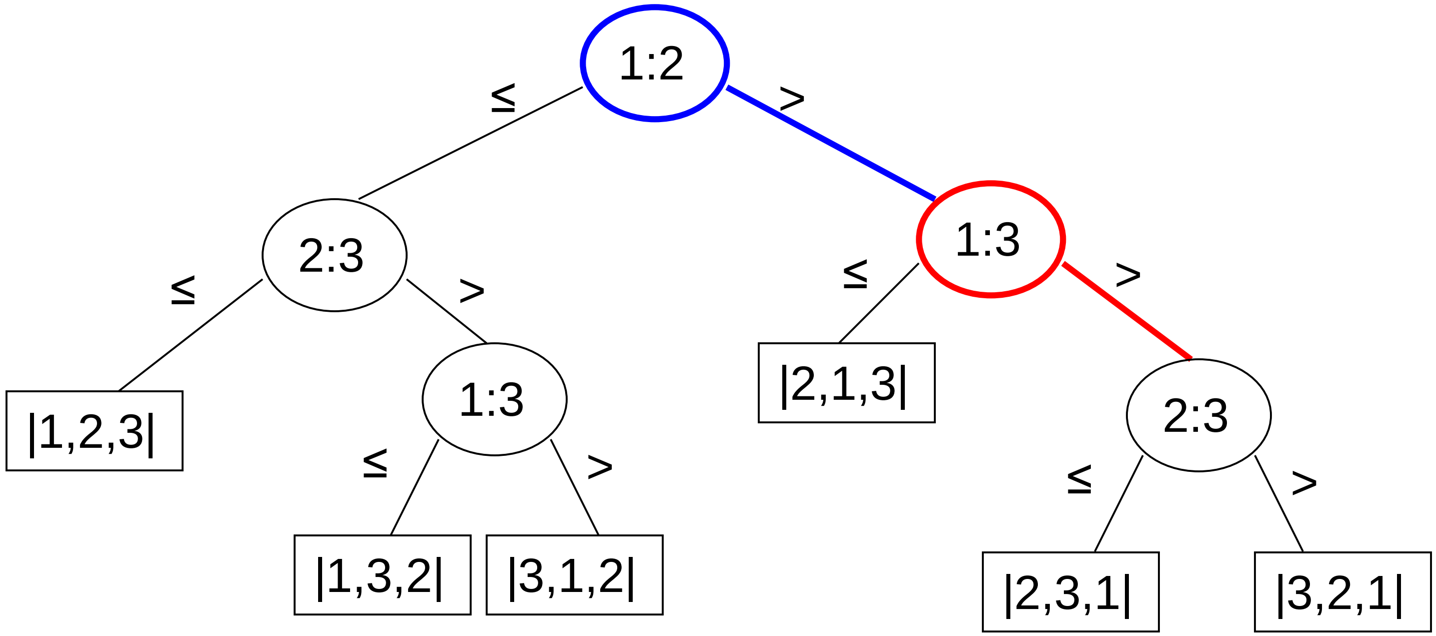
A decision tree model



[12, 7, 3]

Is 12 ≤ 3 or is 12 > 3?

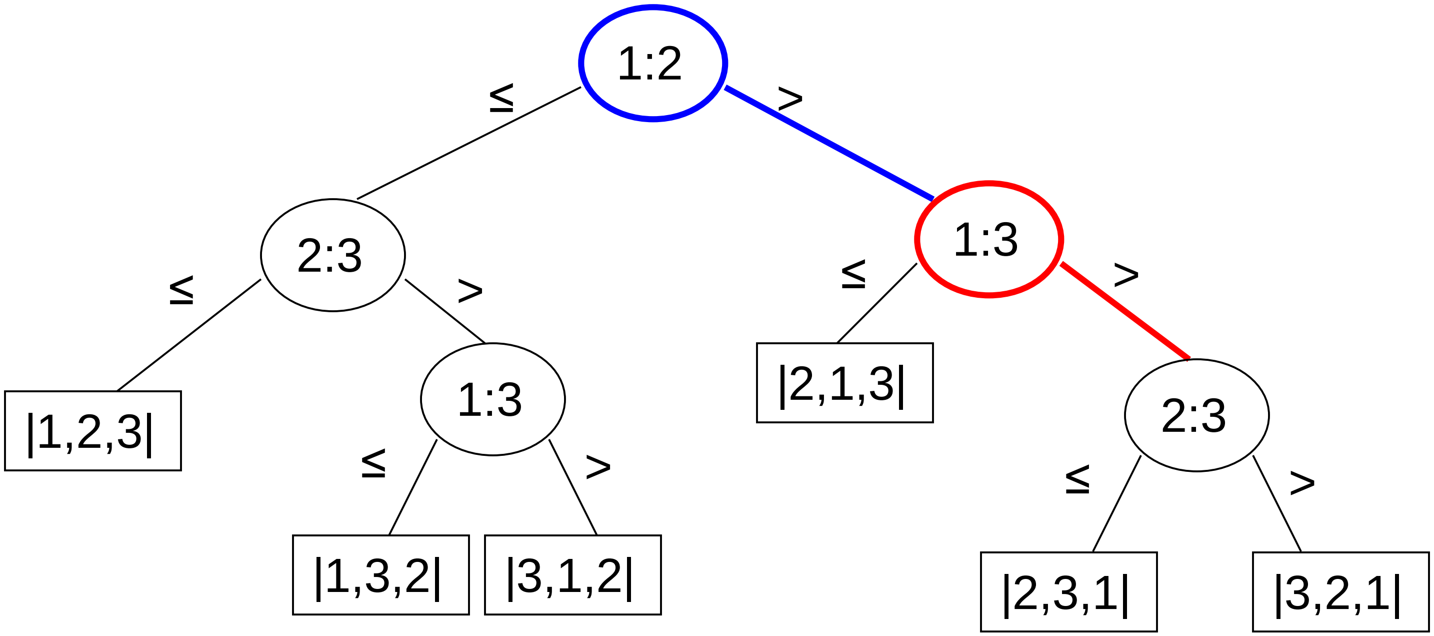
A decision tree model



[12, 7, 3]

Is 12 ≤ 3 or is 12 > 3?

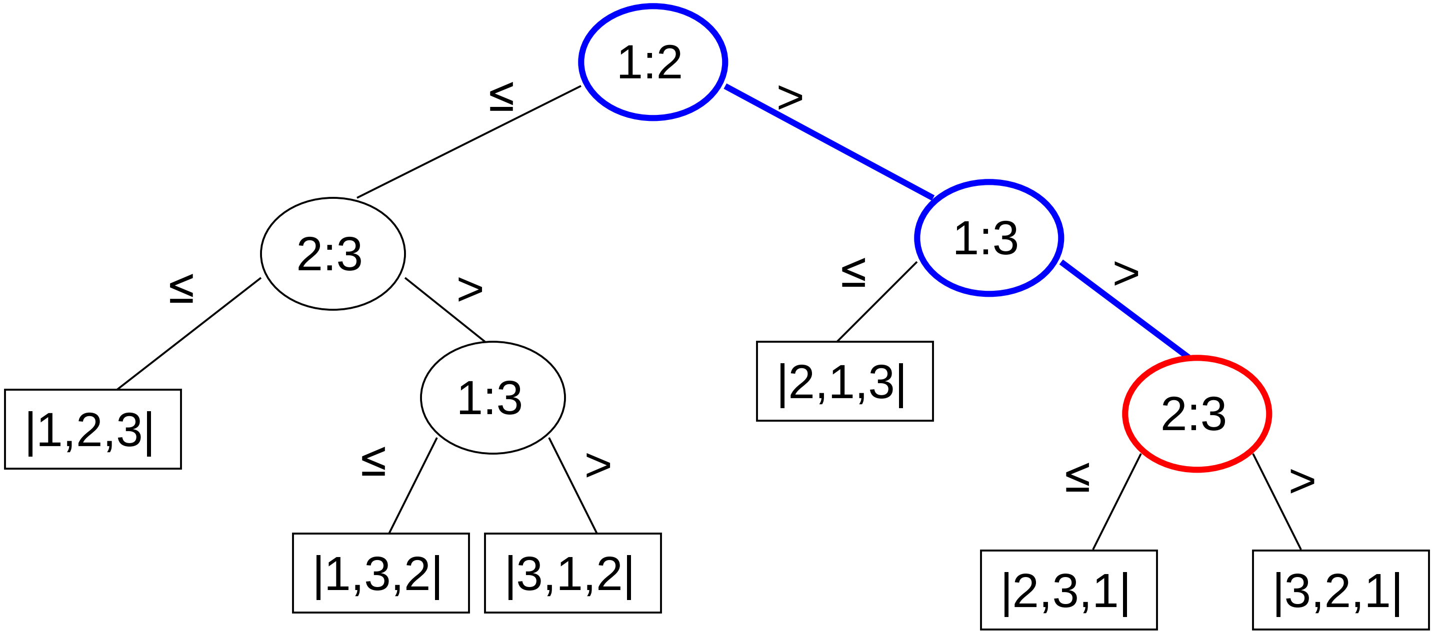
A decision tree model



[12, 7, 3]

Is 12 ≤ 3 or is 12 > 3?

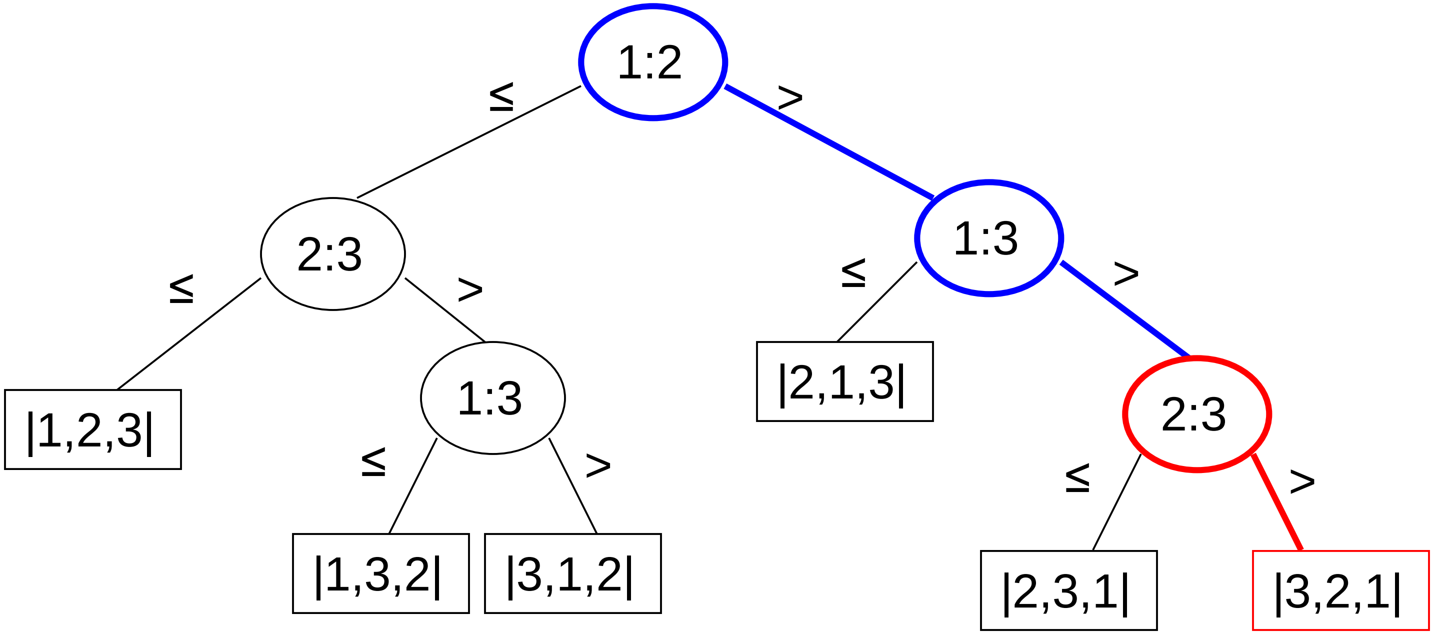
A decision tree model



[12, 7, 3]

Is 7 ≤ 3 or is 7 > 3?

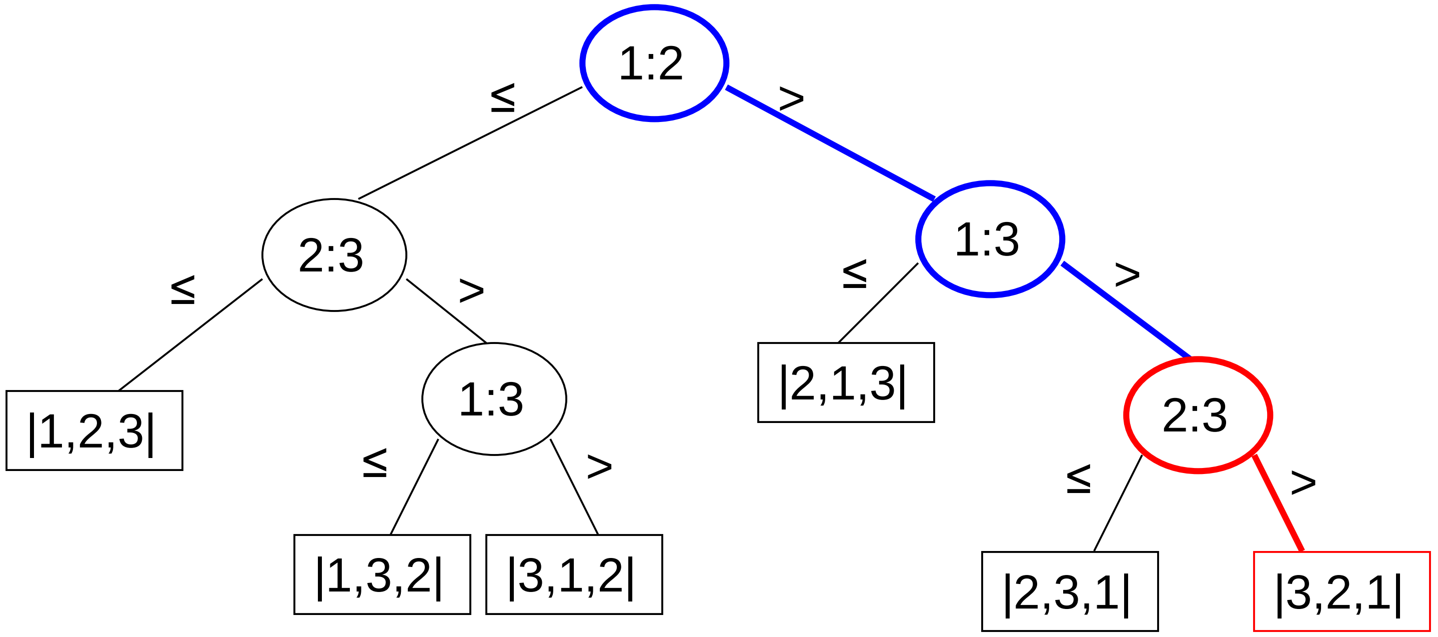
A decision tree model



[12, 7, 3]

Is 7 ≤ 3 or is 7 > 3?

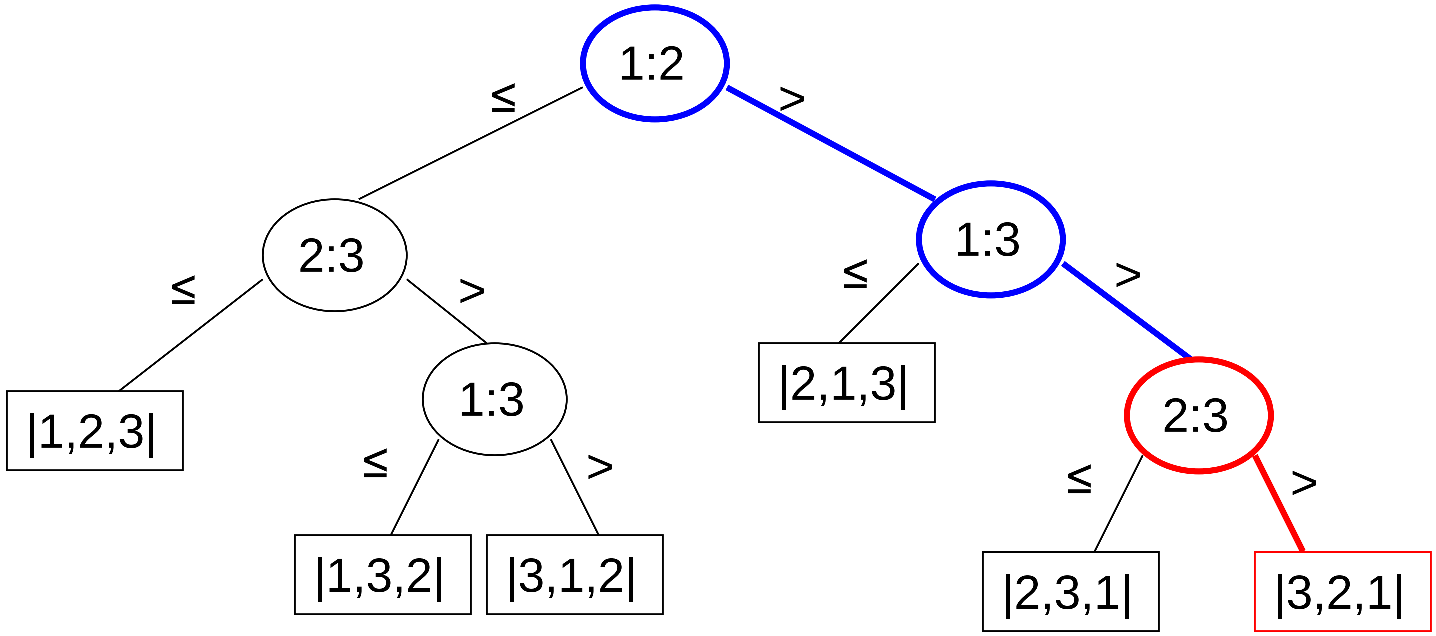
A decision tree model



3, 2, 1

[12, 7, 3]

A decision tree model

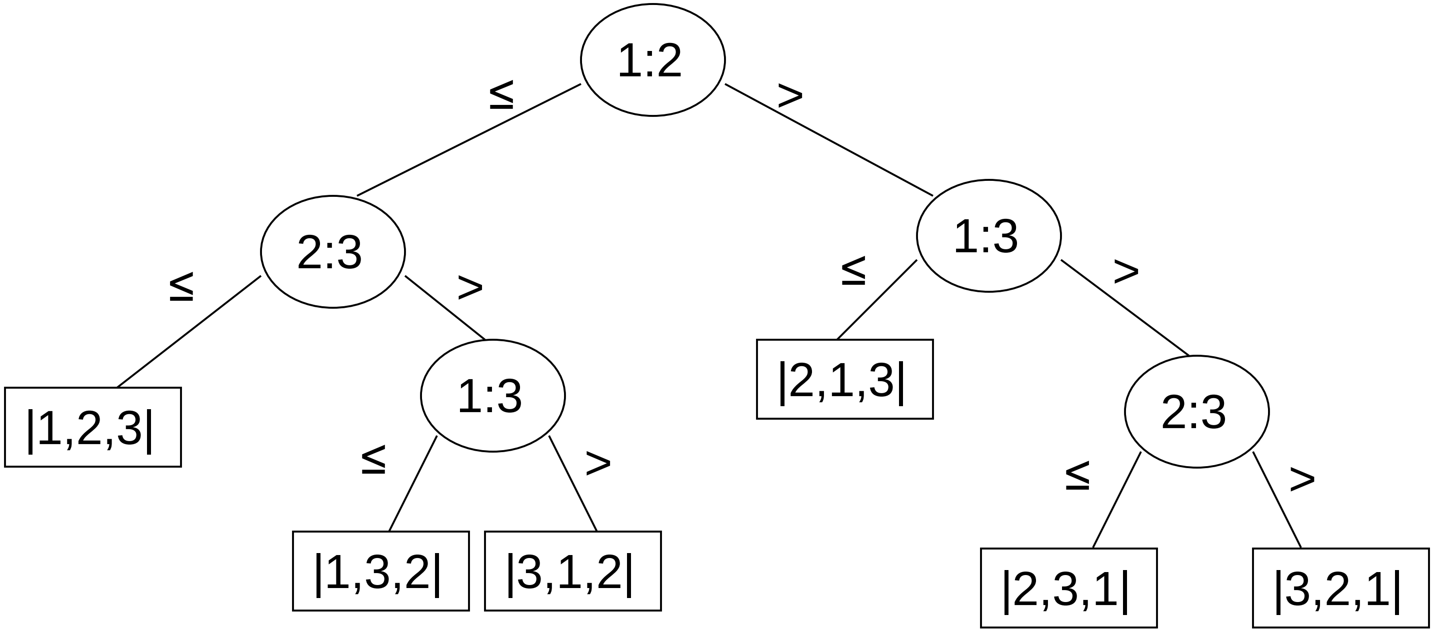


3, 2, 1

[12, 7, 3]

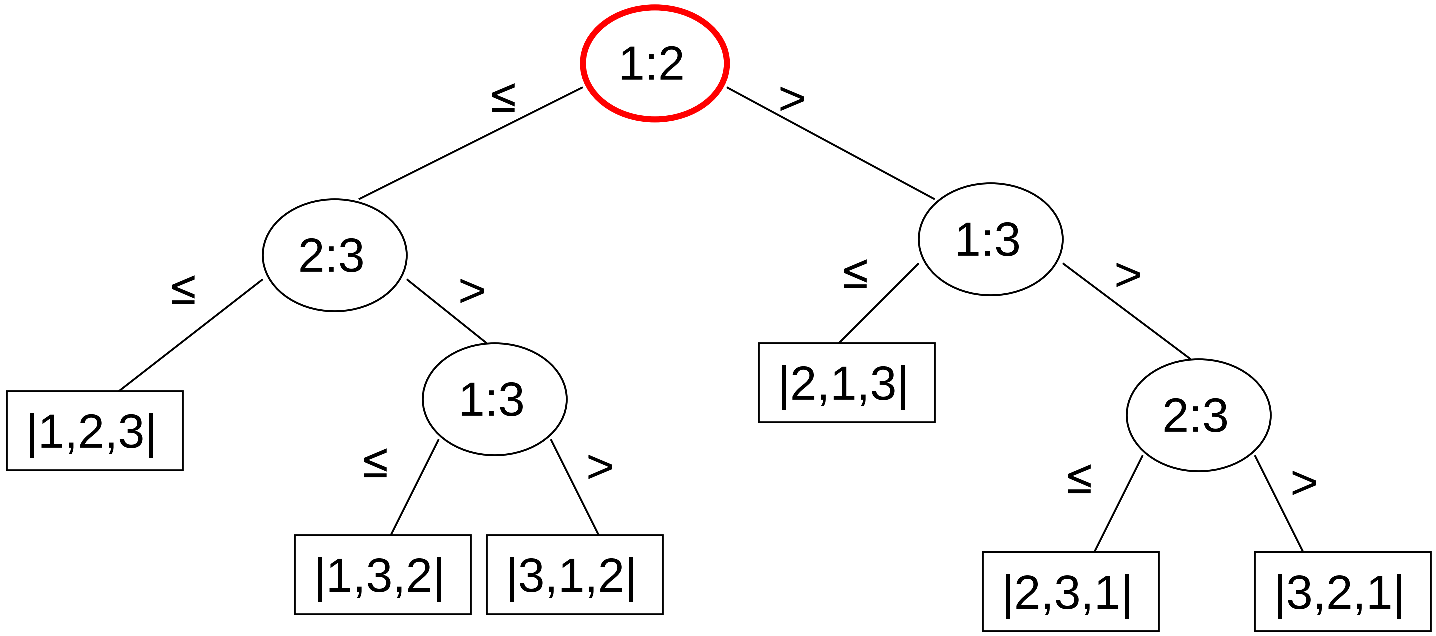
[3, 7, 12]

A decision tree model



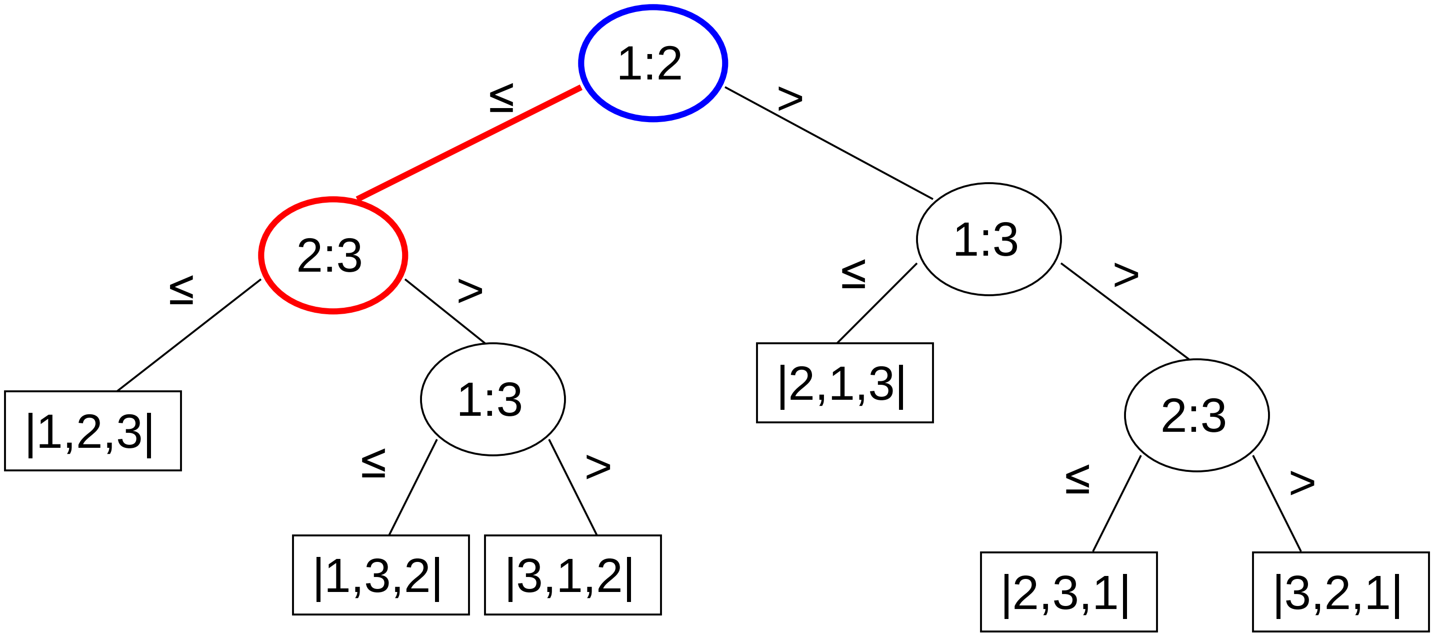
[7, 12, 3]

A decision tree model



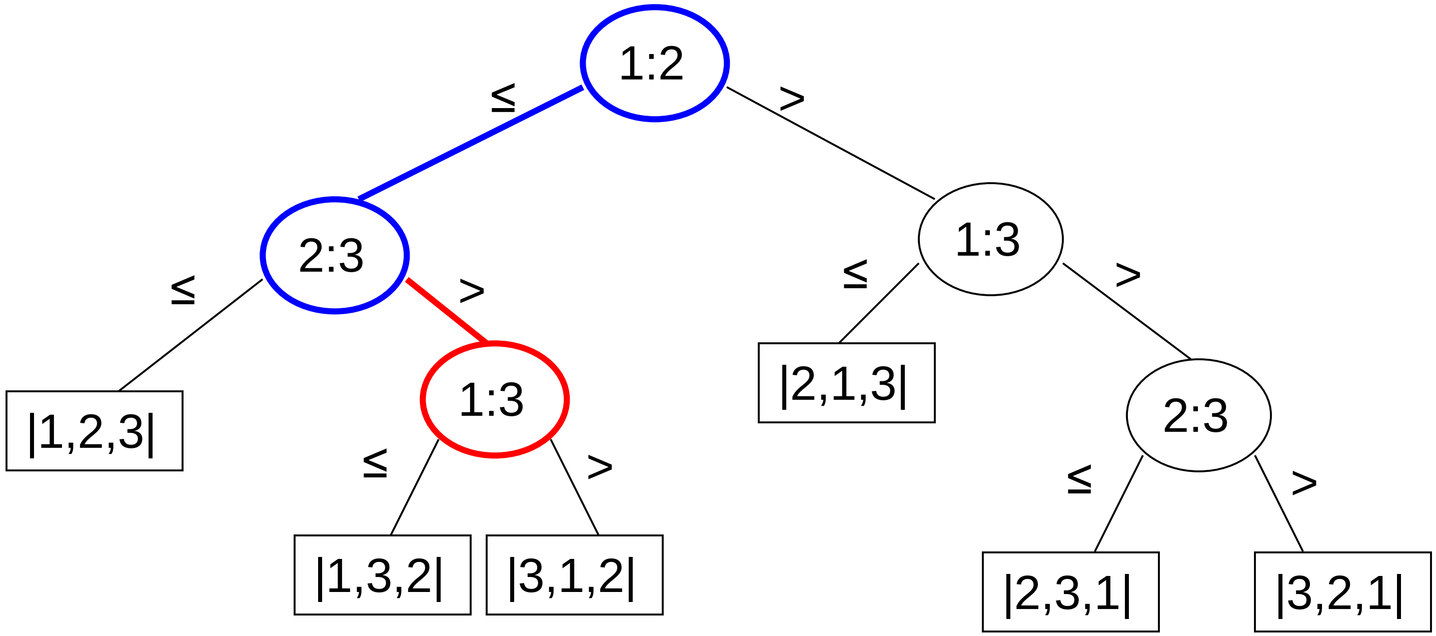
[7, 12, 3]

A decision tree model



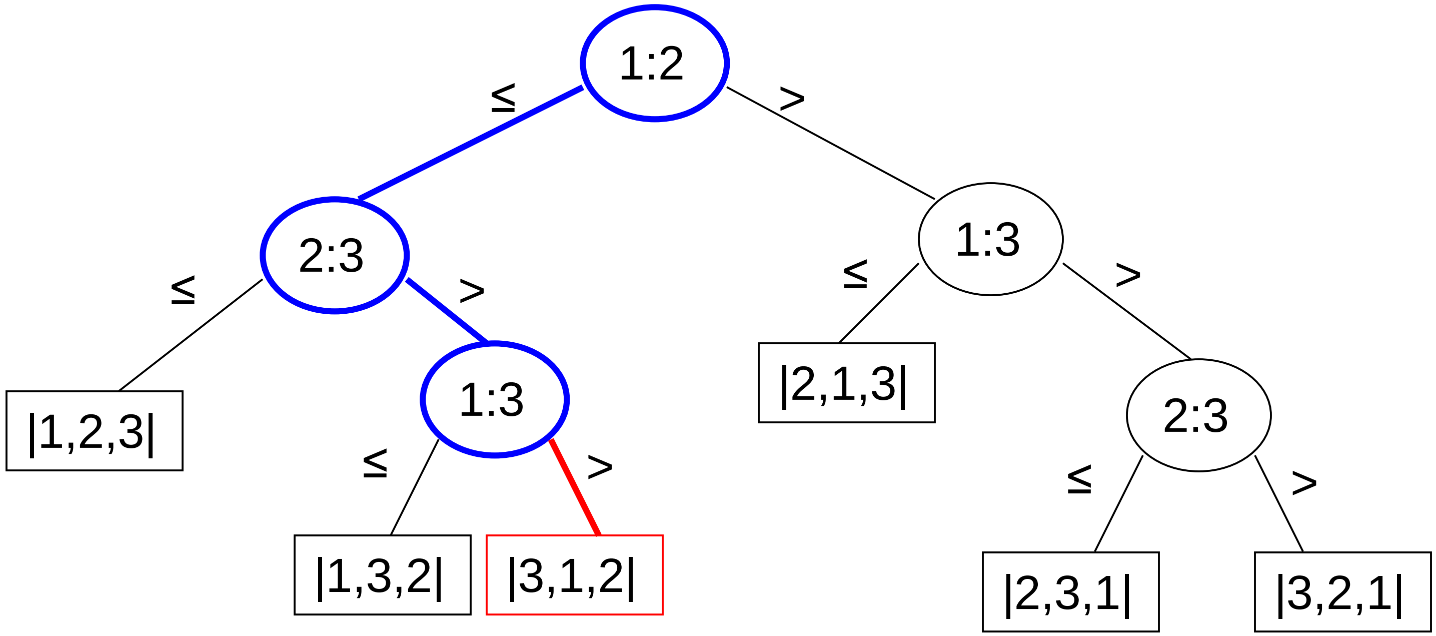
[7, 12, 3]

A decision tree model



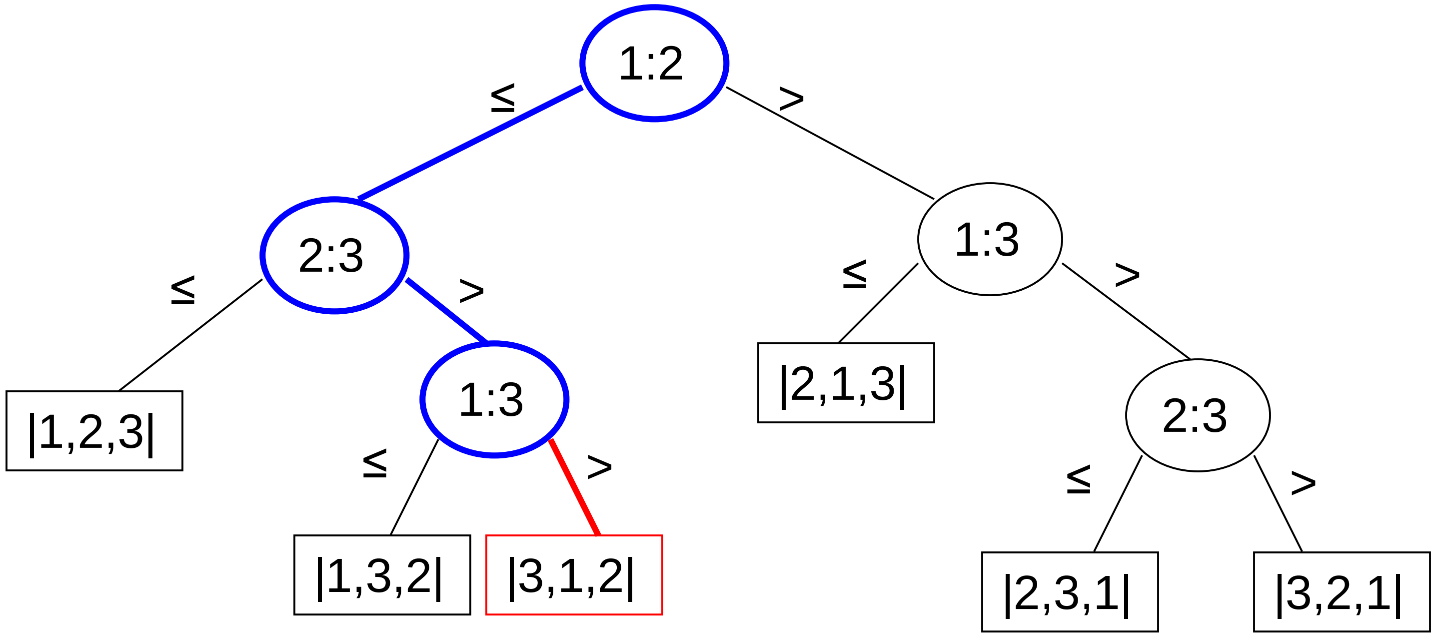
[7, 12, 3]

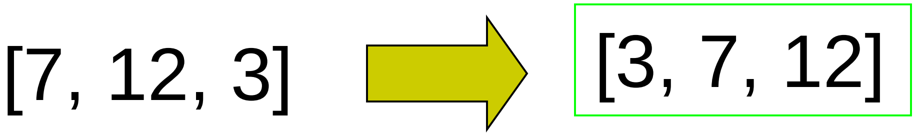
A decision tree model



[7, 12, 3]

A decision tree model



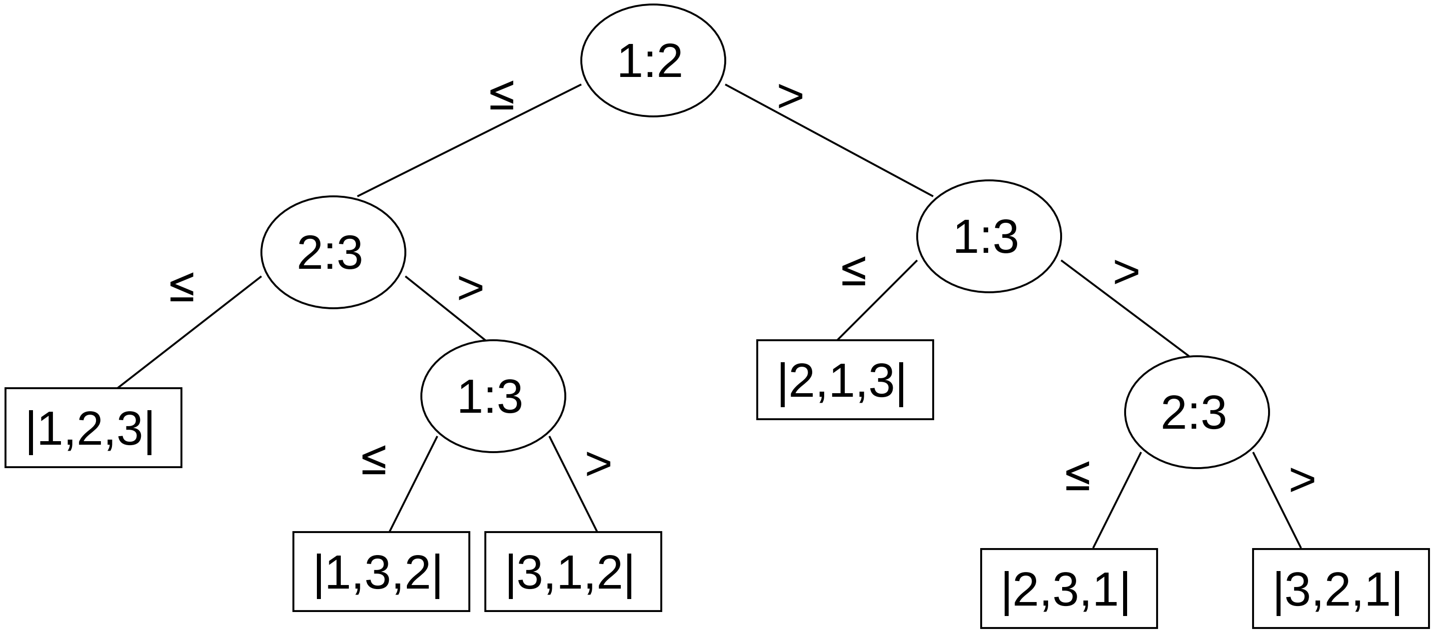


How many leaves are in a decision tree?

- Leaves **must** have all possible permutations of the input
- What if decision tree model didn't?
- Some input would exist that didn't have a correct reordering
- Input of size n, $n!$ leaves

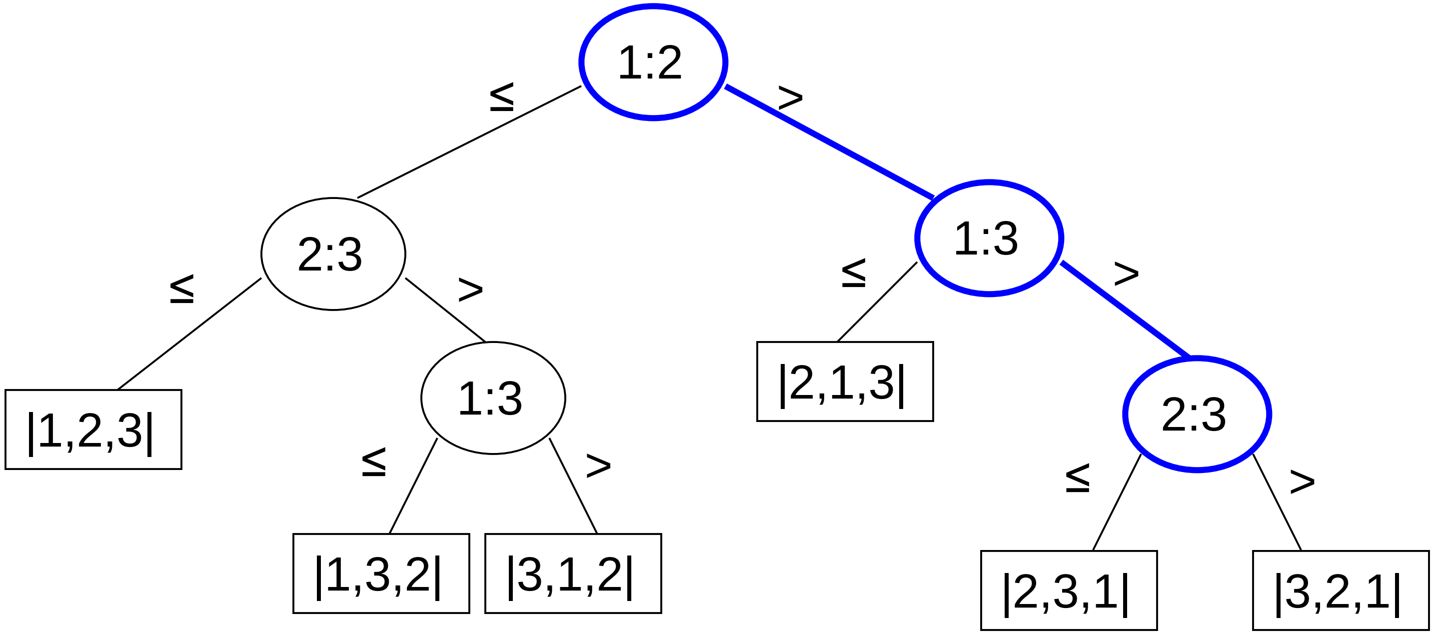
A lower bound

- What is the worst-case number of comparisons for a tree?



A lower bound

- The longest path in the tree, i.e. the height



A lower bound

- What is the maximum number of leaves a binary tree of height *h* can have?
- A complete binary tree has 2^h leaves

$$2^h \geq n!$$

$$h \geq \log n!$$

log is monotonically increasing

$$h = \Omega(n \log n)$$

Can we do better?

Sorting in Linear Time

Counting sort

Radix sort

Bucket sort

Counting Sort – Sort small numbers

- Why it's not a comparison sort:
  - Assumption: input - integers in the range 0..k
  - No comparisons made!
- Basic idea:
  - determine for each input element *x* its *rank:* the *number of elements less* than *x.*
  - once we know the rank *r* of *x*, we can place it in position *r*+1

Counting Sort

The Algorithm

- **Counting-Sort(*A*)**
  - Initialize two arrays *B* and *C* of size *n and k* respectively, and set all entries to 0

- Count the number of occurrences of every A[i]
  - **for** *i* = 1..*n*
  - **do** C[*A*[*i*]] ← C[*A*[*i*]] + 1

- Count the number of occurrences of elements <= A[i]
  - **for** *i* = 2..*n*
  - **do** C[*i*] ← C[*i*] + C[*i* – 1]

- Move every element to its final position
  - **for** *i* = *n*..1
  - **do** B[C[*A*[*i*]] ← *A*[*i*]
  - C[*A*[*i*]] ← C[*A*[*i*]] – 1

Counting Sort Example

A =

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C =

0	1	2	3	4	5
2	0	2	3	0	1

C =

0	1	2	3	4	5
2	2	4	7	7	8

B =

1	2	3	4	5	6	7	8
						3	

C =

0	1	2	3	4	5
2	2	4	6	7	8

Counting Sort Example

	1	2	3	4	5	6	7	8
A =	**2**	**5**	**3**	**0**	**2**	**3**	**0**	**3**

	0	1	2	3	4	5
C =	**2**	**2**	**4**	**6**	**7**	**8**

	1	2	3	4	5	6	7	8
B =		**0**					**3**	

	0	1	2	3	4	5
C =	**1**	**2**	**4**	**6**	**7**	**8**

Counting Sort Example

A =

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C =

0	1	2	3	4	5
2	2	4	6	7	8

B =

1	2	3	4	5	6	7	8
	0				3	3	

C =

0	1	2	3	4	5
1	2	4	5	7	8

Counting Sort

```
CountingSort(A, B, k)
    for i=1 to k
        C[i]= 0;
    for j=1 to n
        C[A[j]] += 1;
    for i=2 to k
        C[i] = C[i] + C[i-1];
    for j=n downto 1
        B[C[A[j]]] = A[j];
        C[A[j]] -= 1;
```

Takes time O(k)

Takes time O(n)

What will be the running time?

Counting Sort

- Total time: $O(n + k)$
  - Usually, $k = O(n)$
  - Thus counting sort runs in $O(n)$ time
- But sorting is $\Omega(n \lg n)$
  - No contradiction--this is not a comparison sort (in fact, there are *no* comparisons at all.)
  - Notice that this algorithm is *stable*
    - If numbers have the same value, they keep their original order

Stable Sorting Algorithms

- A sorting algorithms is ***stable*** if for any two indices *i* and *j* with $i < j$ and $a_i = a_j$, element a_i precedes element a_j in the output sequence.

Input

2_1	7_1	4_1	4_2	2_2	5_1	2_3	6_1

Output

2_1	2_2	2_3	4_1	4_2	5_1	6_1	7_1

Observation: *Counting Sort is stable.*

Counting Sort

- Linear Sort! Cool! *Why don't we always use counting sort?*
- Because it depends on range *k* of elements
- *Could we use counting sort to sort 32 bit integers? Why or why not?*
- Answer: no, *k* too large (2^{32} = 4,294,967,296)

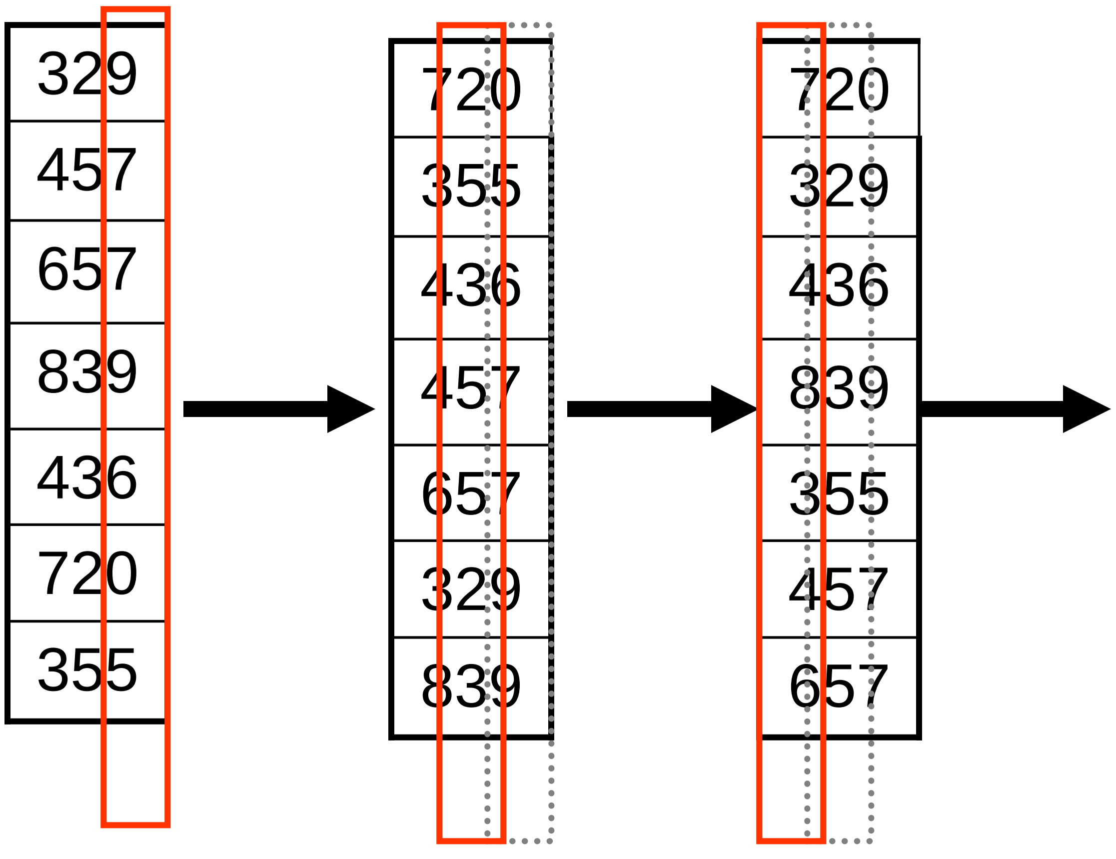
Radix Sort

- Why it's not a comparison sort:
  - Assumption: input has *d* digits each ranging from 0 to *k*
  - *Example: Sort a bunch of 4-digit numbers, where each digit is 0-9*
- Basic idea:
  - Sort elements by digit starting with *least* significant
  - Use a stable sort (like counting sort) for each stage

Radix Sort
The Algorithm

- Radix Sort takes parameters: the array and the number of digits in each array element
- **Radix-Sort(*A*, *d*)**
- 1 **for** *i* = 1..d
- 2 **do** sort the numbers in arrays *A* by their *i*-th digit from the right, using a stable sorting algorithm

Radix Sort Example



Radix Sort

Correctness and Running Time

- What is the running time of radix sort?
  - Each pass over the d digits takes time $O(n+k)$, so total time $O(dn+dk)$
    - When d is constant and $k=O(n)$, takes $O(n)$ time
- Stable, Fast
- Doesn't sort in place (because counting sort is used)

Bucket Sort

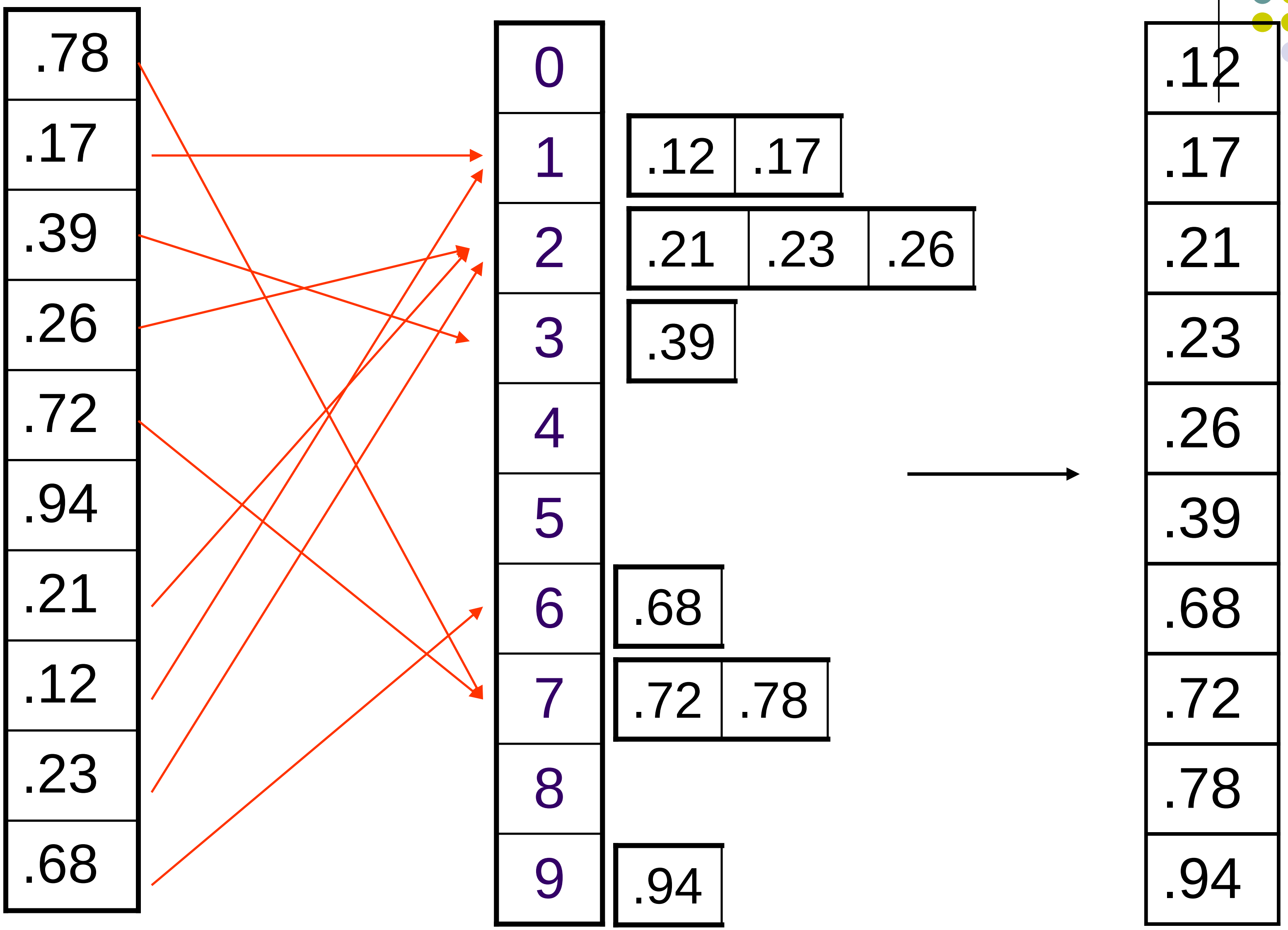
- Assumption: input - *n* real numbers from [0, 1]
- Basic idea:
  - Create *n* linked lists (*buckets*) to divide interval [0,1] into subintervals of size 1/*n*
  - Add each input element to appropriate bucket and sort buckets with insertion sort
- Uniform input distribution → O(1) bucket size
  - Therefore the expected total time is O(n)

Bucket Sort

Bucket-Sort(A)

1. $n \leftarrow$ length(A)
2. **for** $i \leftarrow 1$ to n *Distribute elements over buckets*
3. **do** insert $A[i]$ into list $B[\text{floor}(n*A[i])]$
4. **for** $i \leftarrow 0$ to $n - 1$ *Sort each bucket*
5. **do** Insertion-Sort($B[i]$)
6. Concatenate lists $B[0]$, $B[1]$, … $B[n - 1]$ in order

Bucket Sort Example



Bucket Sort – Running Time

- All lines except line 5 (Insertion-Sort) take $O(n)$ in the worst case.
- In the worst case, $O(n)$ numbers will end up in the same bucket, so in the worst case, it will take $O(n^2)$ time.
- **Lemma:** *Given that the input sequence is drawn uniformly at random from* $[0,1]$*, the expected size of a bucket is* $O(1)$.
- So, in the *average case*, only a constant number of elements will fall in each bucket, so it will take $O(n)$ (see proof in book).

- Use a different indexing scheme (hashing) to distribute the numbers uniformly.

Summary

- Every ***comparison-based sorting*** algorithm has to take **$\Omega(n \lg n)$ *time***.
- ***Merge Sort***, ***Heap Sort***, and ***Quick Sort*** are comparison-based and take $O(n \lg n)$ time. Hence, they ***are optimal***.
- Other ***sorting algorithms can be faster*** by exploiting assumptions made about the input
- ***Counting Sort*** and ***Radix Sort*** take ***linear time*** for ***integers in a bounded range***.
- ***Bucket Sort*** takes ***linear average-case time*** for ***uniformly distributed*** real numbers.

Review of Existing Linear Sorting

Non-Comparison Based Sorting Algorithms

- **Counting sort** assumes input elements are in range [0,1,2,..,k] and uses array indexing to count the number of occurrences of each value.

- **Radix sort** assumes each integer consists of d digits, and each digit is in range [1,2,..,k'].

- **Bucket sort** requires advance knowledge of input distribution (sorts n numbers uniformly distributed in range in O(n) time).

[B1] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson, Introduction to Algorithms (2nd ed.). McGraw-Hill Higher Education, 2001.

Non-comparison Sort: A new approach (DR)

DR Features

Salient Features of DR (Dividend-Remainder) are:

- ✓ First algorithm which works only on arithmetic operators.
- ✓ Easy to understand and implement.
- ✓ Worst case as well as expected running time complexity is $O\left(n\frac{log\ k}{\log n}\right)$ where k is the largest number in the input sequence, n is the total number of inputs.
- ✓ Does not require the uniform distribution of numbers as bucket sort does.
- ✓ Works with same time complexity no matter what is the range of input integers as required by count sort which runs in time $O(n^2)$if the range of integers K $= O(n^2)$
- ✓ **Stable** sorting algorithm.
- ✓ **Incremental sort**: can produce outputs in the each subsequent passes. On the other hand, Radix sort, Bucket sort, Count sort produce results at the end of the entire pass.
- ✓ It does not extend the memory usage as Radix sort does when increase its based to n.

➢ ***Special Mention Award** in Poster presentation at Research Colloquia in XRCI Open 2016.*

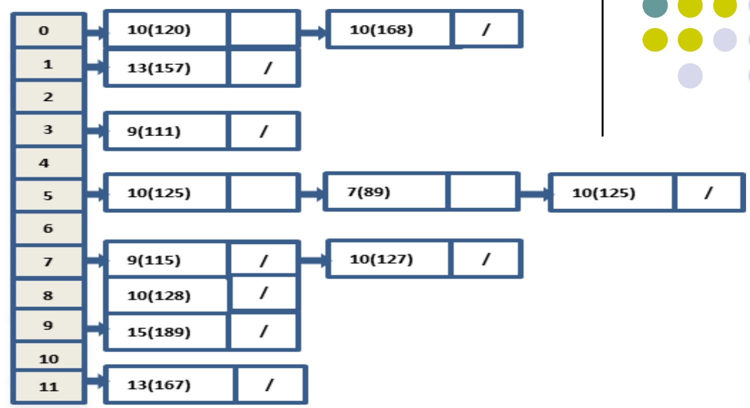
DR Algorithm

1. Count =0;
2. **While** (Count <n)
3.   **For** all elements
4.   Calculate **Remainder and Dividend** (= element/n)
5.   Store the **Dividend at the remainder** position
6.   **End for**
7.   **For** all elements
8.   **If** (Dividend of element==0)
9.   Store it in Output list
10.   Count++;
11.   **End if**
12.   **End for**
13. **End while**

DR Demonstration

n=12, k=189

Elements (Initial Input List)	Index	Pass 1: Dividend(Element)
125	0	**10(120),14(168)**
167	1	**13(157)**
120	2	
115	3	**9(111)**
189	4	
111	5	**10(125),7(89),10(125)**
89	6	
127	7	**9(115),10(127)**
168	8	**10(128)**
128	9	**15(189)**
157	10	
125	11	**13(167)**

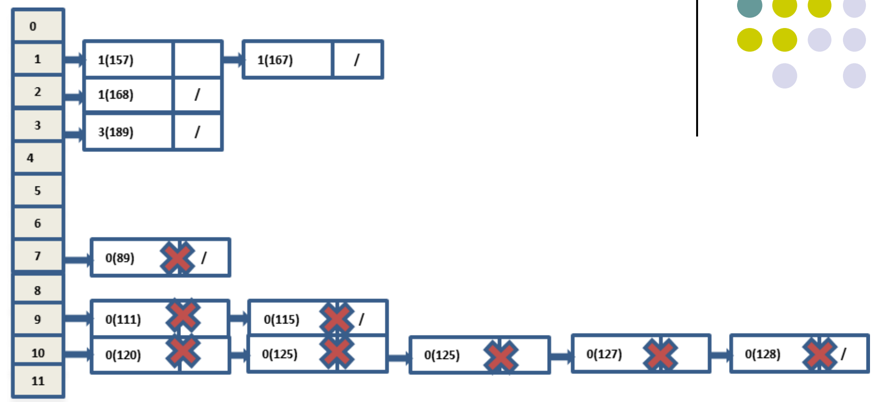



```
Count =0;
While (Count <n)
    For all elements
        Calculate Remainder and Dividend
        Store the Dividend at the remainder position
    End for
    For all elements
        If (Dividend of element ==0)
            Store it in Output list
            Count++;
        End if
    End for
End while
```

Final output:

DR Demonstration

Elements	Index	Pass 1: Dividend(Element)	Pass 2
125	0	10(120),14(168)	
167	1	13(157)	1(157),1(167)
120	2		1(168)
115	3	9(111)	1(189)
189	4		
111	5	10(125),7(89),10(125)	
89	6		
127	7	9(115),10(127)	**0(89)**
168	8	10(128)	
128	9	15(189)	**0(111),0(115)**
157	10		**0(120),0(125),0(125),0(127)0(128)**
125	11	13(167)	



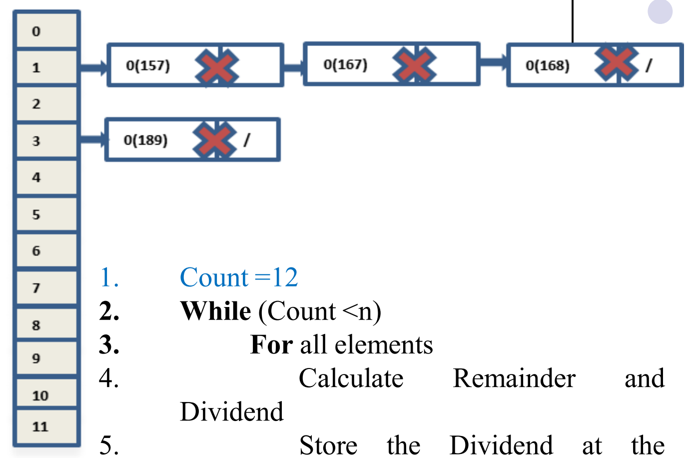

```
Count =8;
While (Count <n)
        For all elements
                Calculate Remainder and Dividend
                Store the Dividend at the remainder position
        End for
        For all elements
                If (Dividend of element ==0)
                    Store it in Output list
                    Count++;
                End if
        End for
End while
```

Final output:

89	111	115	120	125	125	127	128				

DR Demonstration

Elements	Index	Pass 1: Dividend(Element)	Pass 2	Pass 3:
125	0	10(120),14(168)		
167	1	13(157)	1(157),1(167)	**0(157),0(167),0(168),0(189)**
120	2		1(168)	
115	3	9(111)	1(189)	
189	4			
111	5	10(125),7(89),10(125)		
89	6			
127	7	9(115),10(127)	0(89)	
168	8	10(128)		
128	9	15(189)	0(111),0(115)	
157	10		0(120),0(125),0(125),0(127)0(128)	
125	11	13(167)		



1. Count =12
2. **While** (Count <n)
3. **For** all elements
4. Calculate Remainder and Dividend
5. Store the Dividend at the remainder position
6. **End for**
7. **For** all elements
8. **If** (Dividend of element ==0)
9. Store it in Output list
10. Count++;
11. **End if**
12. **End for**
13. **End while**

Final output:

89	111	115	120	125	125	127	128	157	167	168	189

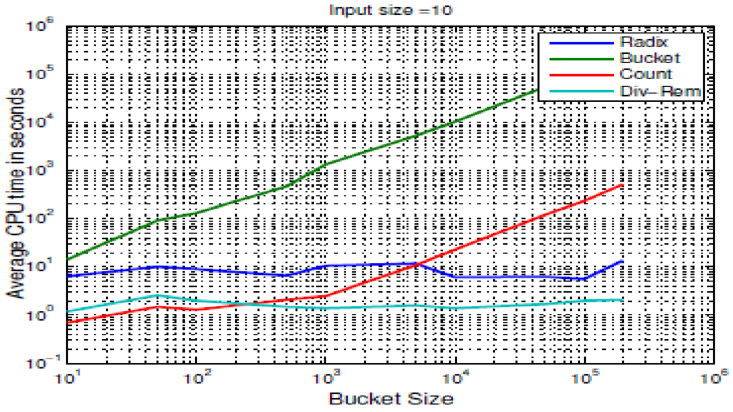
Implementation

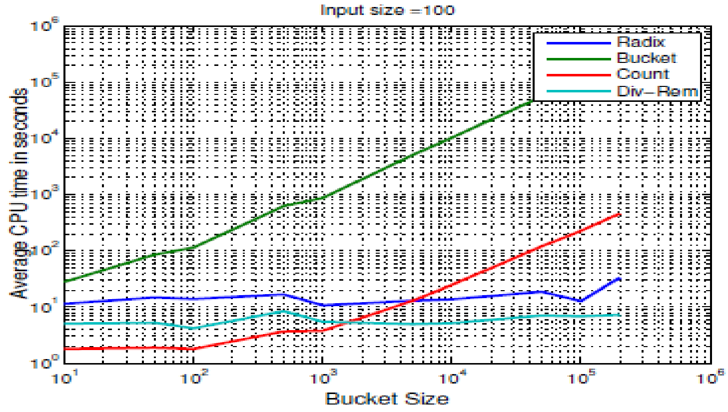
Algorithm 1 DR: Dividend-Remainder Algorithm using array of Linked Lists

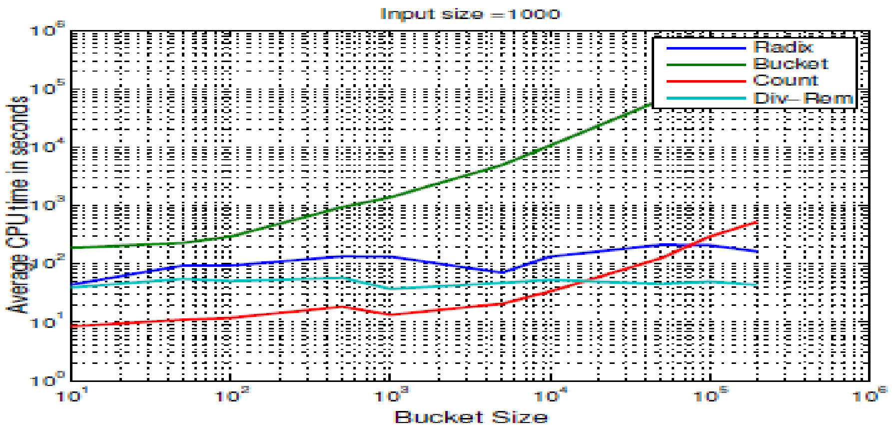
Require: Array $A[n]$ of input integers, where n is the size of input
Ensure: Array $B[n]$ containing sorted sequence of input numbers

```
set flag = 0, out = 0, index[n], idx[n], list[n]   ▷ list[n] is an array of pointers of node type
while out < n do
    if flag = 0 then
        for i = 0 to n − 1 do
            dividend = A[i]/n
            remainder = A[i]%n
            INSERTNODE( dividend, remainder)
            index[i] = index[i] + 1
        end for
    else
        for i = 0 to n − 1 do
            set pointer p = list[i]
            while index[i] > 0 do
                if p− > dividend == 0 then
                    B[out + +] = p− > key
                else
                    dividend = p− > dividend/n
                    remainder = p− > dividend%n
                    INSERTNODE( p− > dividend, remainder)
                    idx[remainder] = idx[remainder] + 1
                end if
                DELETENODE(p, i)
                index[i] = index[i] − 1
            end while
        end for
        copy array idx to index
    end if
    flag = 1
end while
```

Experimental Results







Comparative Study

Table I: List of Noncomparison Based Practical Sorting Algorithm

Name	Time (Average Case)	Time (worst Case)	Memory	Stability	$n << 2^k$	Remarks
LSD Radix Sort [Cormen et al. 2001]	nk/d	nk/d	$n + 2^d$	*yes*	*no*	Depends on other sorting like count, insertion sort and less space efficient
MSD Radix Sort	nk/d	nk/d	$n + 2^d$	*yes*	*no*	Depends on other sorting like count, insertion sort and less space efficient
MSD Radix Sort (in place)	nk/d	nk/d	2^d	*yes*	*no*	Depends on other sorting like count, insertion sort.
Counting Sort [Cormen et al. 2001]	$n + r$	$n + r$	$n + r$	*yes*	*yes*	Limited to range of value i.e. $k << O(n^2)$
Bucket Sort [Cormen et al. 2001]	$n + r$	$n + r$	$n + r$	*yes*	*yes*	Limited to uniform distribution, also depends on other sorting like count, insertion sort.
Spread Sort [Ross 2002]	nk/d	$n(k/s + d)$	$2^d(k/d)$	*no*	*no*	Limited to uniform distribution, more programmer effort is required in implementation .
Burst Sort [Sinha and Zobel 2004]	nk/d	nk/d	nk/d	*no*	*no*	Uses trie (standard prefix tree) for storage efficiency but in time complexity as similar as MSD radix sort.
Flash Sort [Neubert 1998]	$n + r$	n^2	n	Can be with additional $O(n)$ space	*no*	Requires uniform distribution to run in $O(n)$ otherwise it drives in $O(n^2)$ as insertion sort.
Postman Sort	nk/d	nk/d	$n + 2^d$	$-$	*no*	A variation of bucket sort, very specific to MSD radix sort.
DR	$n\lvert\log k/\log n\rvert$	$n\lvert\log k/\log n\rvert$	n	*yes*	*no*	

DR Algorithm: Summary

- ✓ First novel sorting algorithm using **Division & Modulus Operators.**
- ✓ It **overcomes the drawbacks** of the count, bucket and radix sort with improved performance.
- ✓ The novelty of the DR algorithm in terms of time its complexity: upper bounded by O(n) (for $k < n$).
- ✓ DR algorithm is also **tested through real system implementation**.
- ✓ It **beats three algorithms** (count, bucket and radix sort) for large data values, using array implementation.
- ✓ **Constraint free**.
- ✓ **Incremental sort.**
- ✓ Can be **used for parallel applications**.