Theoretical Bounds on Sorting

CS61B, Spring 2025 @ UC Berkeley

Slides credit: Josh Hug

©2002 Ron Zeno

Goal: How Hard is Sorting?

Lecture 34, CS61B, Spring 2025

Sorts Summary

	Memory	# Compares	Notes	Stable?
Heapsort	$\Theta(1)$	$\Theta(N \log N)$	Bad caching (61C)	No
Insertion	$\Theta(1)$	$\Theta(N^2)$	Best for almost sorted and N < 15	Yes
Mergesort	$\Theta(N)$	$\Theta(N \log N)$	Fastest stable sort	Yes
Quicksort LTHS	$\Theta(\log N)$	$\Theta(N \log N)$ expected	Fastest sort	No

This is due to the cost of tracking recursive calls by the computer, and is also an "expected" amount. The difference between log N and constant memory is trivial.

You can create a stable Quicksort. However, using unstable partitioning schemes (like Hoare partitioning) and using randomness to avoid bad pivots tend to yield better runtimes.

Sorting

Sorting is a foundational problem.

- Obviously useful for putting things in order.
- But can also be used to solve other tasks, sometimes in non-trivial ways.
  - Sorting improves duplicate finding from a naive N^2 to N log N.
  - Sorting improves 3SUM from a naive N^3 to N^2.
- There are many ways to sort an array, each with its own interesting tradeoffs and algorithmic features.

Today we'll discuss the fundamental nature of the sorting problem itself: How hard is it to sort?

For today, we'll only consider comparison sorts, which sort a list of items using only two operations:

- compareTo, which compares two items and says which one is greater
- swap, which swaps the indices of two items in the array

This allows our sorting algorithm to work on any Comparable object

Examples of comparison sorts:

Selection Sort, Quicksort, Mergesort, Insertion Sort, Heapsort

Examples of non-comparison sorts:

BOGOsort, Sleepsort, Stalinsort, Miracle Sort

Comparison Sorts (My Answer)

For today, we'll only consider comparison sorts, which sort a list of items using only two operations:

- compareTo, which compares two items and says which one is greater
- swap, which swaps the indices of two items in the array

This allows our sorting algorithm to work on any Comparable object

Examples of comparison sorts:

 Heapsort, Mergesort, Quicksort, Insertion Sort, Selection Sort, Bubble Sort

Examples of non-comparison sorts: Radix Sort, Sleepsort, Gravity Sort

Math Problem Warmup

Lecture 34, CS61B, Spring 2025

A Math Problem out of Nowhere

Consider the functions N! and $(N/2)^{N/2}$

Is $N! \in \Omega((N/2)^{N/2})$? Prove your answer.

- Recall that $\in \Omega$ can be informally be interpreted to mean $\geq$
- In other words, does factorial grow at least as quickly as $(N/2)^{N/2}$?

A Math Problem out of Nowhere

Consider the functions N! and $(N/2)^{N/2}$

Is $N! \in \Omega((N/2)^{N/2})$? Prove your answer.

10!

- **10 * 9 * 8 * 7 * 6** * ... * 1

5^5

- **5 * 5 * 5 * 5 * 5**

$N! > (N/2)^{N/2}$, for large N, therefore $N! \in \Omega((N/2)^{N/2})$

Another Math Problem

Given: $N! > (N/2)^{N/2}$, which we used to prove our answer to the previous problem.

Show that $\log(N!) \in \Omega(N \log N)$.

- Recall: log means an unspecified base.
- Remember your log rules:
  - $\log(A^B) = B * \log(A)$
  - $\log(A*B) = \log(A) + \log(B)$
  - $\log_A(B) = \log(B)/\log(A)$

Given that $N! > (N/2)^{N/2}$

Show that $\log(N!) \in \Omega(N \log N)$.

We have that $N! > (N/2)^{N/2}$

- Taking the log of both sides, we have that $\log(N!) > \log((N/2)^{N/2})$.
- Bringing down the exponent we have that $\log(N!) > N/2 \log(N/2)$.
- Using log rules we have that $\log(N!) > N/2 (\log(N) - \log(2))$
- Discarding the unnecessary constants, we have $\log(N!) \in \Omega(N \log (N/2))$.
- From there, we have that $\log(N!) \in \Omega(N \log N)$.

In other words, $\log(N!)$ grows at least as quickly as $N \log N$.

Last Math Problem

In the previous problem, we showed that $\log(N!) \in \Omega(N \log N)$.

Now show that $N \log N \in \Omega(\log(N!))$.

Show that N log N $\in \Omega(\log(N!))$

Proof:

- $\log(N!) = \log(N) + \log(N-1) + \log(N-2) + \ldots + \log(1)$
- N log N $= \log(N) + \log(N) + \log(N) + \ldots \log(N)$
- Therefore N log N $\in \Omega(\log(N!))$

Given:

- N log N ∈ Ω(log(N!))
- log(N!) ∈ Ω(N log N)

Which of the following can we say?

A.   N log N ∈ Θ(log N!)
B.   log N! ∈ Θ(N log N)
C.   Both A and B
D.   Neither

Given:

- $N \log N \in \Omega(\log(N!))$

  Informally: N log N ≥ log(N!)

- $\log(N!) \in \Omega(N \log N)$

  Informally: log(N!) ≥ N log N

Which of the following can we say?

A. $N \log N \in \Theta(\log N!)$

B. $\log N! \in \Theta(N \log N)$

Informally: N log N = log(N!)

C. Both A and B

D. Neither

Summary

We've shown that log(N!) ∈ Θ(N log N).

- In other words, these two functions grow at the same rate asymptotically.

As for why we did this, we will see in a little while…

Simple Bounds for TUCS (the ultimate comparison sort)

Lecture 34, CS61B, Spring 2025

Sorting

We have shown several sorts to require $\Theta(N \log N)$ worst case time.

- Can we build a better sorting algorithm?

Let the ultimate comparison sort (TUCS) be the asymptotically fastest possible comparison sorting algorithm, possibly yet to be discovered, and let $R(N)$ be its worst case runtime.

Give the best Ω and O bounds you can for $R(N)$.

It might seem strange to give Ω and O bounds for an algorithm whose details are completely unknown, but you can, I promise!

Sorting

We have shown several sorts to require $\Theta(N \log N)$ worst case time.

- Can we build a better sorting algorithm?

Let the ultimate comparison sort (TUCS) be the asymptotically fastest possible comparison sorting algorithm, possibly yet to be discovered, and let $R(N)$ be its worst case runtime.
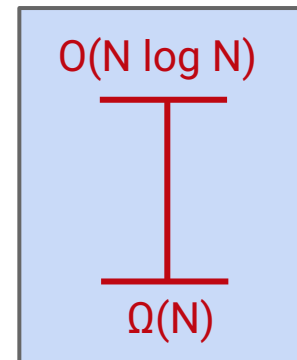
- Worst case run-time of TUCS, $R(N)$ is $O(N \log N)$.
  - Obvious: Mergesort is $\Theta(N \log N)$ so $R(N)$ can't be worse!

- Worst case run-time of TUCS, $R(N)$ is $\Omega(1)$.
  - Obvious: Problem doesn't get easier with N.
  - Can we make a stronger statement than $\Omega(1)$?

O(N log N)

$\Omega(1)$
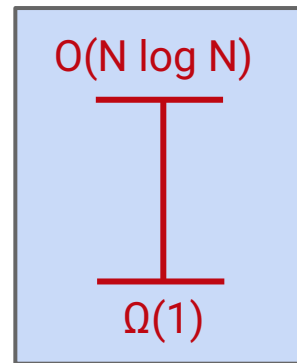
TUCS Worst
Case Θ Runtime

Let TUCS be the asymptotically fastest possible comparison sorting algorithm, possibly yet to be discovered.

- Worst case run-time of TUCS, R(N) is O(N log N). Why?
- Worst case run-time of TUCS, R(N) is also Ω(N).
  - Have to at least look at every item.

TUCS Worst
Case Θ Runtime

Sorting

We know that TUCS "lives" between N and N log N.

- Worst case asymptotic runtime of TUCS is between Θ(N) and Θ(N log N).

- Can we make an even stronger statement on the lower bound?
  - With a clever argument, yes (as we'll see soon see).
    - Spoiler alert: It will turn out to be Ω(N log N)
  - This lower bound means that across the infinite space of all possible ideas that any human might ever have for sorting using sequential comparisons, NONE has a worst case runtime that is better than Θ(N log N).

O(N log N)

Ω(N)
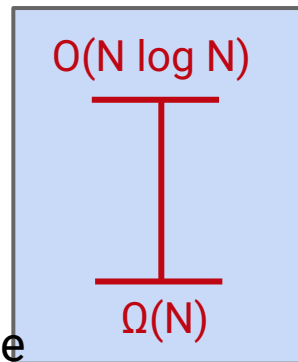
TUCS Worst
Case Θ Runtime

Coin Puzzles

Lecture 34, CS61B, Spring 2025

9 Coins

Suppose we have nine coins that are all identical in appearance. However, one of them is a counterfeit coin, and weighs slightly more than the other 8 (which are all the same weight).

We have a scale that can be used to compare two sets of coins at a time, but we can only use it twice. How can we determine the counterfeit coin?

Many solutions, but here's the classic one:

Step 1: Compare coins 123 vs 456

There are three cases:

1: Left side heavier

2: Right side heavier

3: The two are equal

In case 1: The heavier coin is either 1, 2, or 3

Step 2.1: Compare coins 1 vs 2

There are three possibilities:

1: Left side heavier

Coin 1 is counterfeit

2: Right side heavier

Coin 2 is counterfeit

3: The two are equal

Coin 3 is counterfeit

In case 2: The heavier coin is either 4, 5, or 6

Step 2.2: Compare coins 4 vs 5

There are three possibilities:

1: Left side heavier

Coin 4 is counterfeit

2: Right side heavier

Coin 5 is counterfeit

3: The two are equal

Coin 6 is counterfeit

In case 3: The heavier coin is either 7, 8, or 9

Step 2.3: Compare coins 7 vs 8

There are three possibilities:

1: Left side heavier

Coin 7 is counterfeit

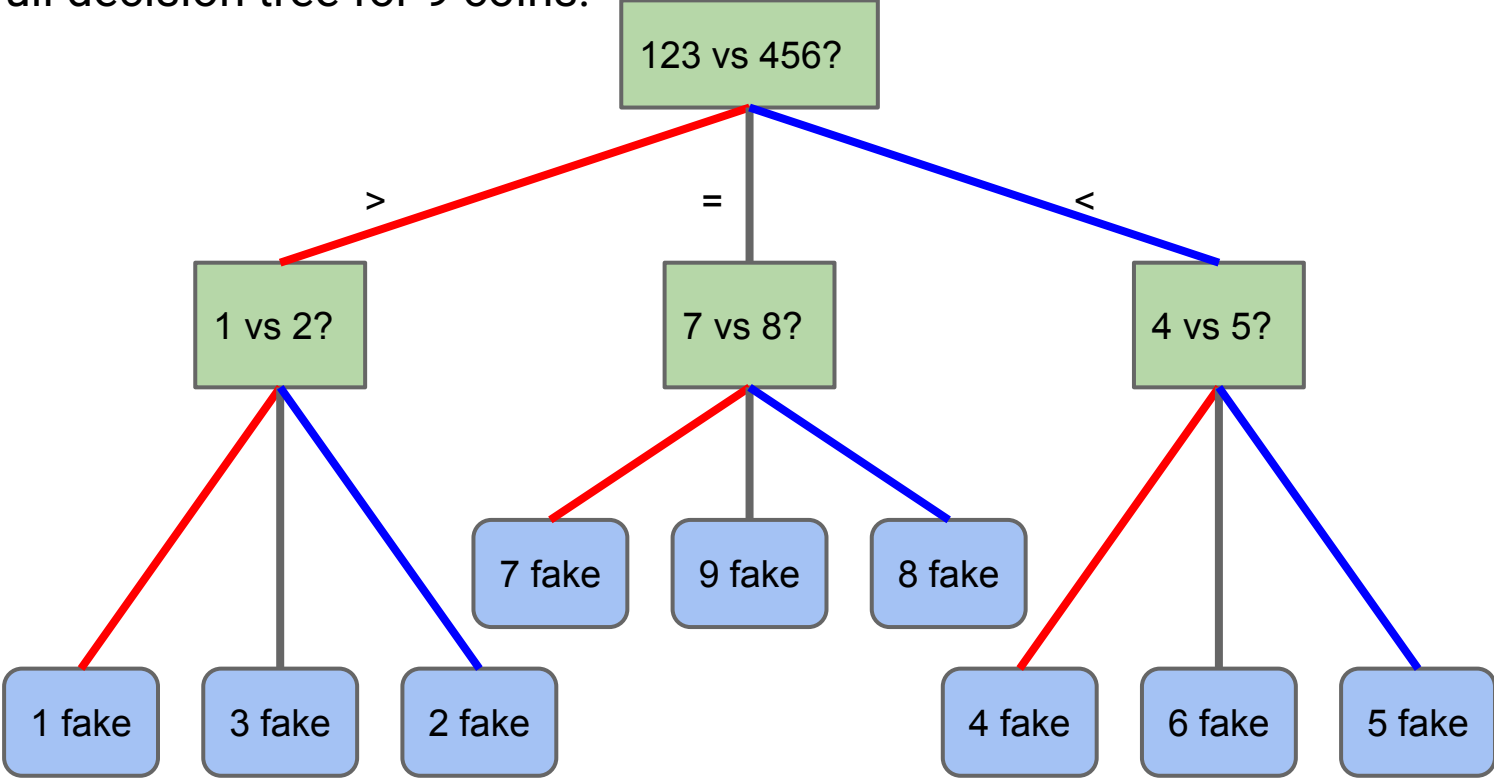

2: Right side heavier

Coin 8 is counterfeit

3: The two are equal

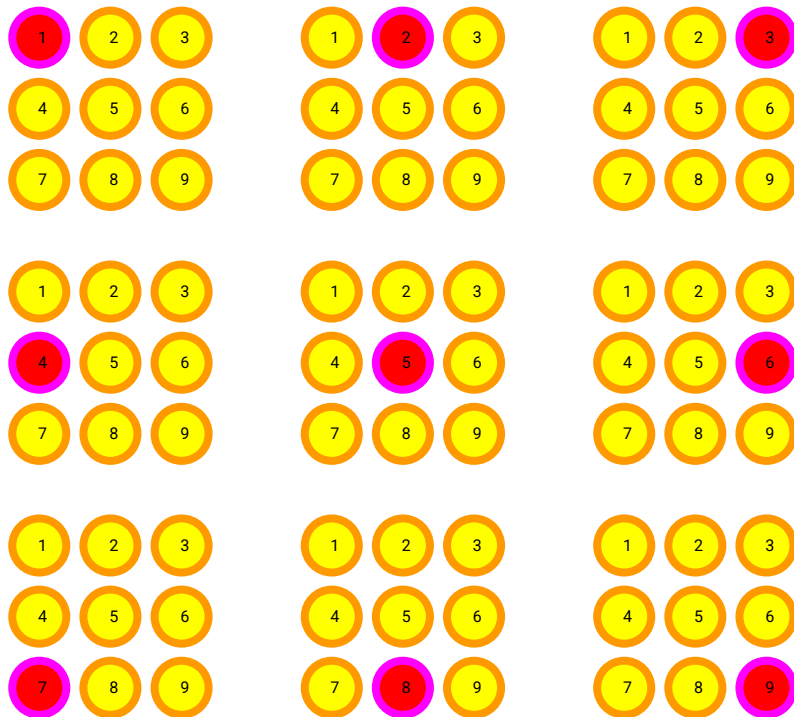Coin 9 is counterfeit
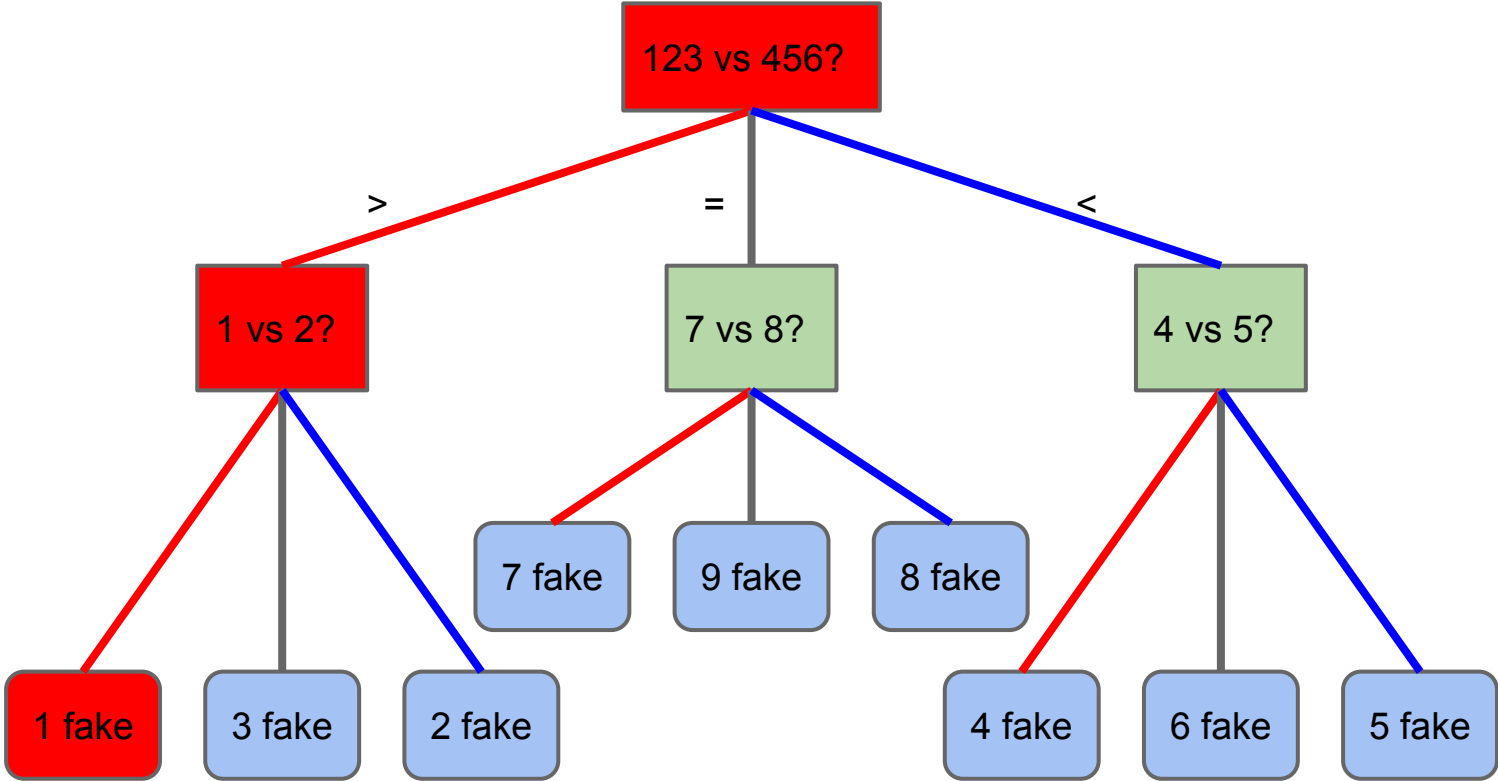
The full decision tree for 9 coins:

To verify that our scheme works, let's check every single case.

There are nine "universes" that we could start in (1 is counterfeit, 2 is counterfeit, etc.). We can verify that in each universe, we yield the correct result

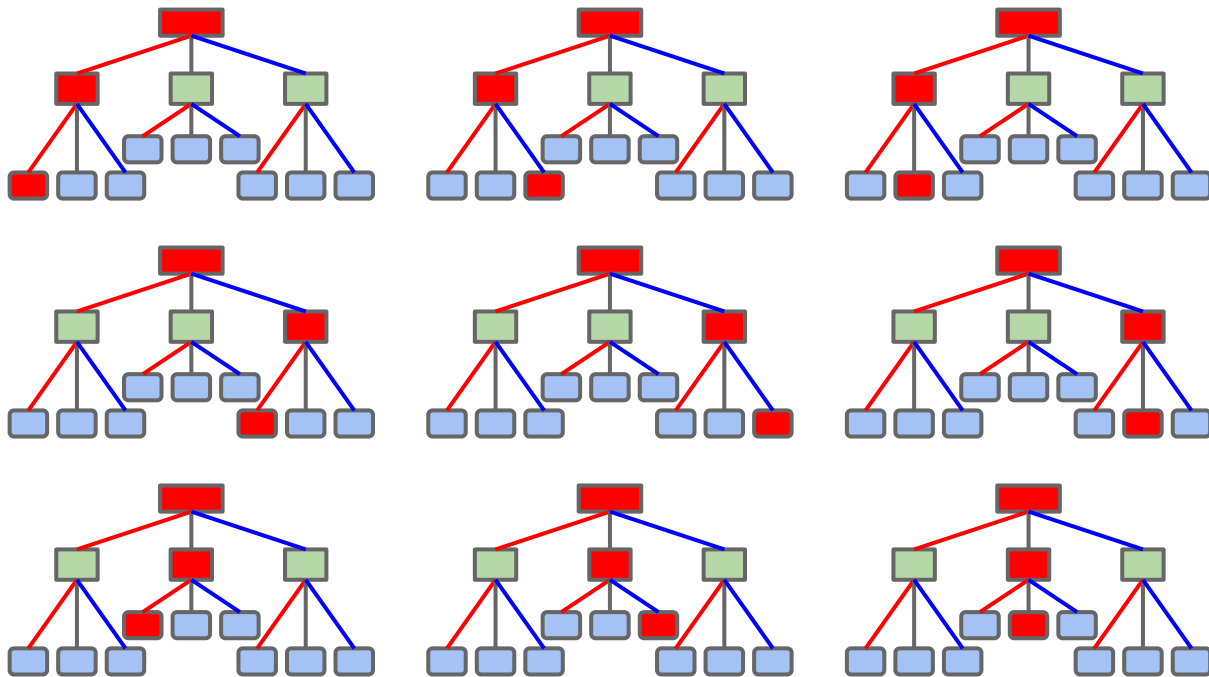The full decision tree in Universe 1:

To verify that our scheme works, let's check every single case.

There are nine "universes" that we could start in (1 is counterfeit, 2 is counterfeit, etc.). We can verify that in each universe, we yield the correct result

To verify that our scheme works, let's check every single case.

There are nine "universes" that we could start in (1 is counterfeit, 2 is counterfeit, etc.). We can verify that in each universe, we yield the correct result

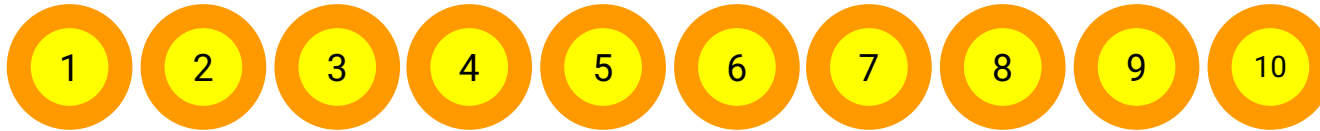|		
1 fake	2 fake	3 fake
4 fake	5 fake	6 fake
7 fake	8 fake	9 fake

10 Coins
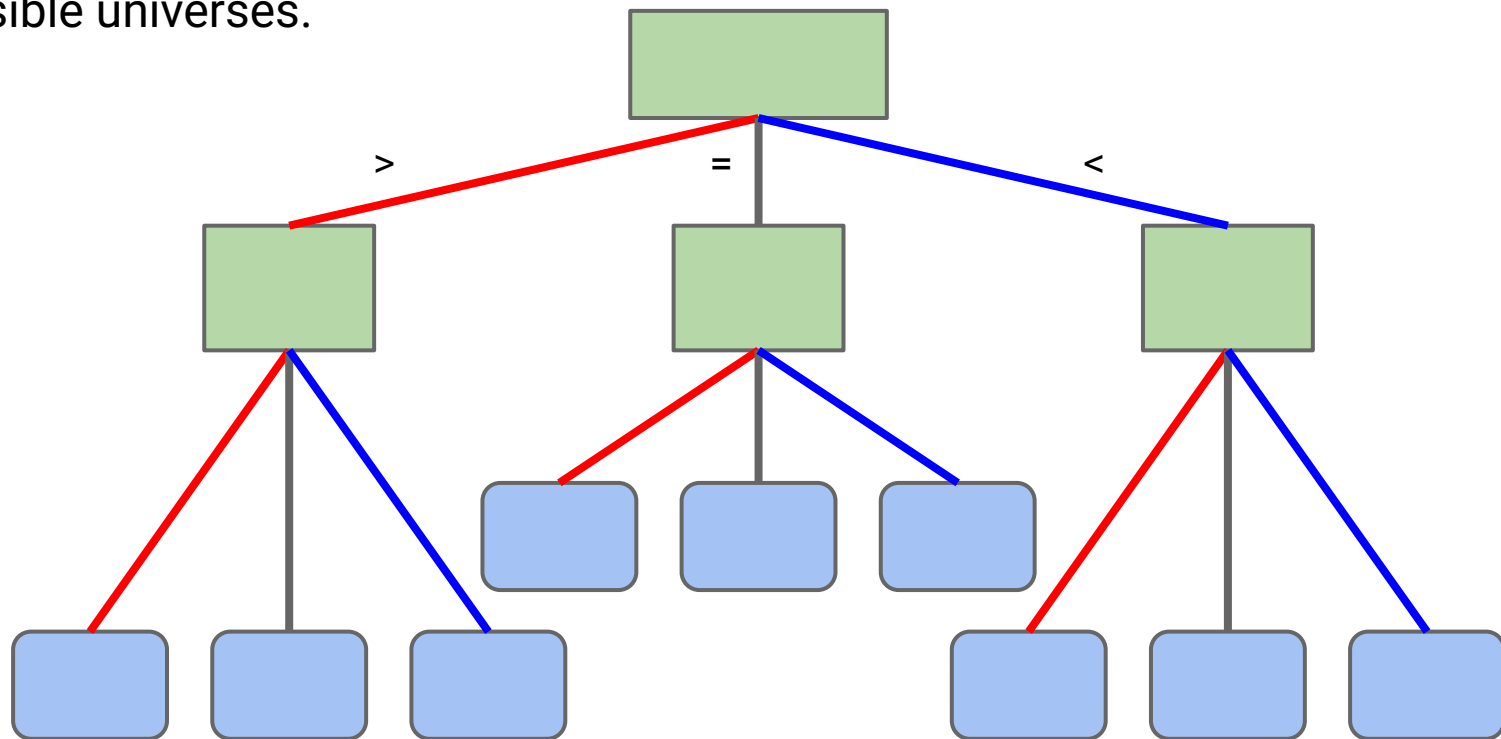
With 9 coins, we can find the counterfeit coin in 2 weighings.

Question: Can we do this with 10 coins?

No: With a two-layer decision tree, we have at most 9 leaves, but there are 10 possible universes.
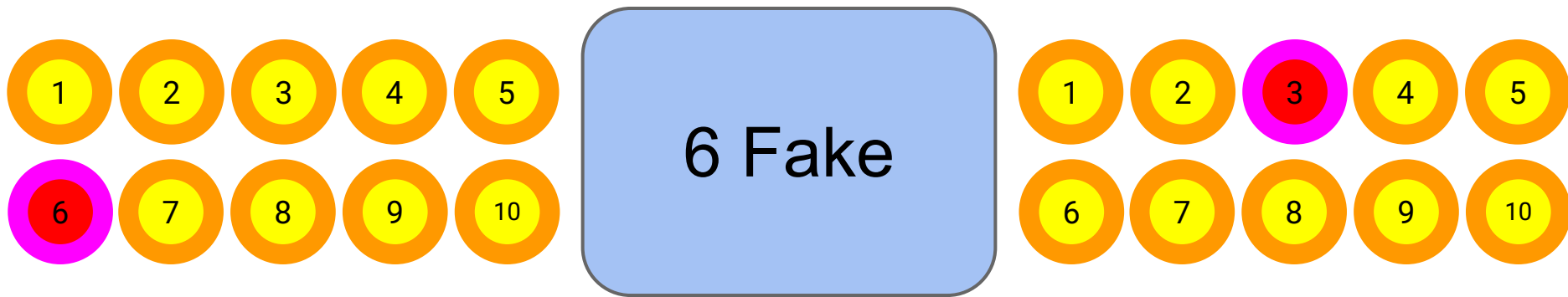
Therefore, at least one leaf must have two universes that lead there

In that leaf, regardless of the two universes that end up there, we'll be wrong in at least one of those universes.

Therefore it is impossible to guarantee a determination on which of 10 coins is counterfeit with only 2 weighings.

Each weighing triples the number of cases we can manage. So with N weighings, we can have at most 3^N leaves in our decision tree. This means that if we have $>3^N$ coins, we have proven that no algorithm exists.

Does the converse hold true?

If we have $X <= 3^N$ coins, does our proof guarantee an algorithm exist?

In this case, we can find an algorithm for X coins as long as the condition holds, but this isn't necessarily true (see skipped slides for a counterexample).
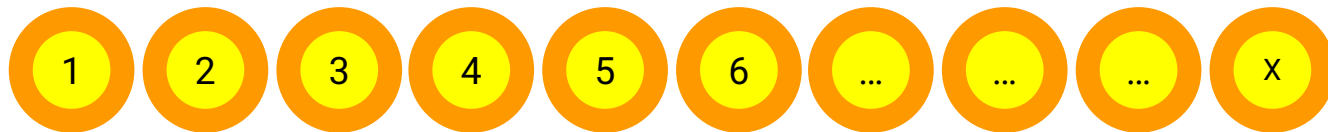
This time, the counterfeit coin can either be slightly lighter, or slightly heavier than all the other coins. We want to know both which coin is counterfeit, and whether it was heavier, or lighter. This time, we can use the scale 3 times, so in theory, we can handle 27 universes. Which of the following are possible?
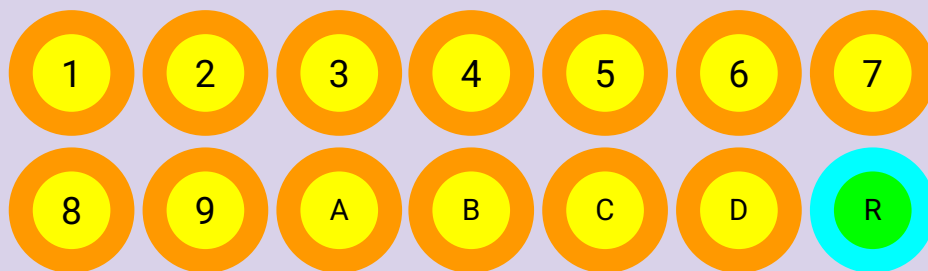
A.   12 coins (24 universes)
B.   12 coins if we have a 13th reference coin that is guaranteed to be real
C.   12 coins if we don't care about whether the counterfeit is heavier or lighter
D.   13 coins (26 universes)
E.   13 coins if we have a 14th reference coin that is guaranteed to be real
F.   13 coins if we don't care about whether the counterfeit is heavier or lighter

1   2   3   4   5   6   7

8   9   A   B   C   D   R

For 12 coins, this is possible (and by extension the other 12-coin puzzles)
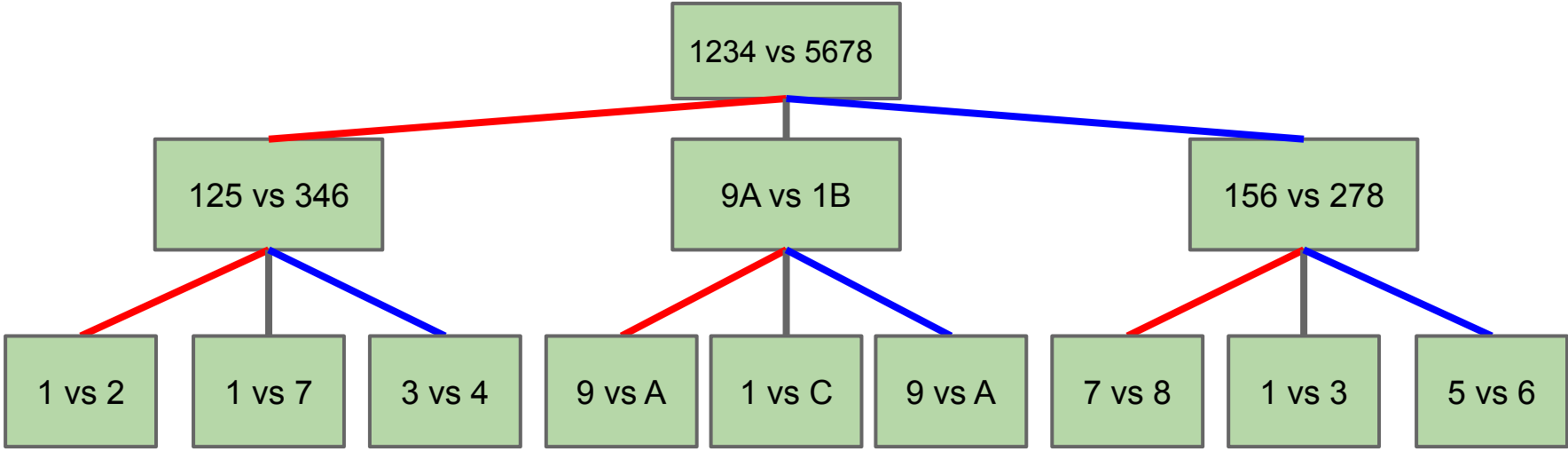
13 Coins: Proof of Impossibility

For 13 coins, this is impossible. We do have 26 < 27 universes, but we also need our first weighing to narrow down to 9 or fewer universes. Our first weighing needs to compare an equal number of coins, so we can check all possible weighings to show that no first weighing splits our 27 universes into groups 9 or smaller (the below is the closest we get)
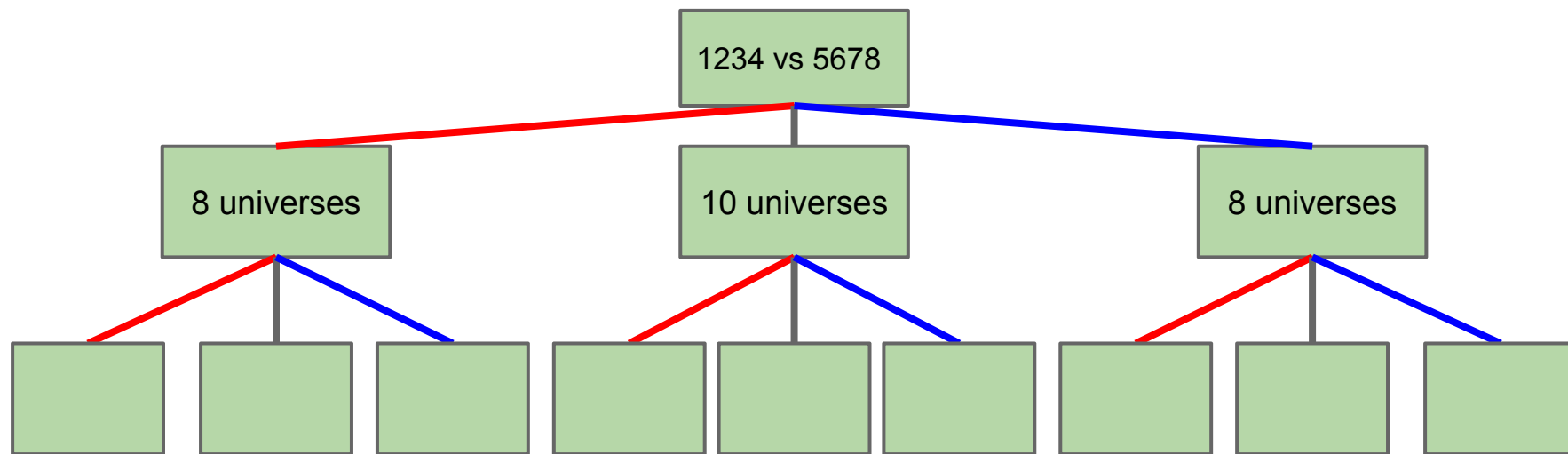
13 Coins with a Reference

For 13 coins with a reference, this is possible, because the reference lets us split into 9-8-9 universes.
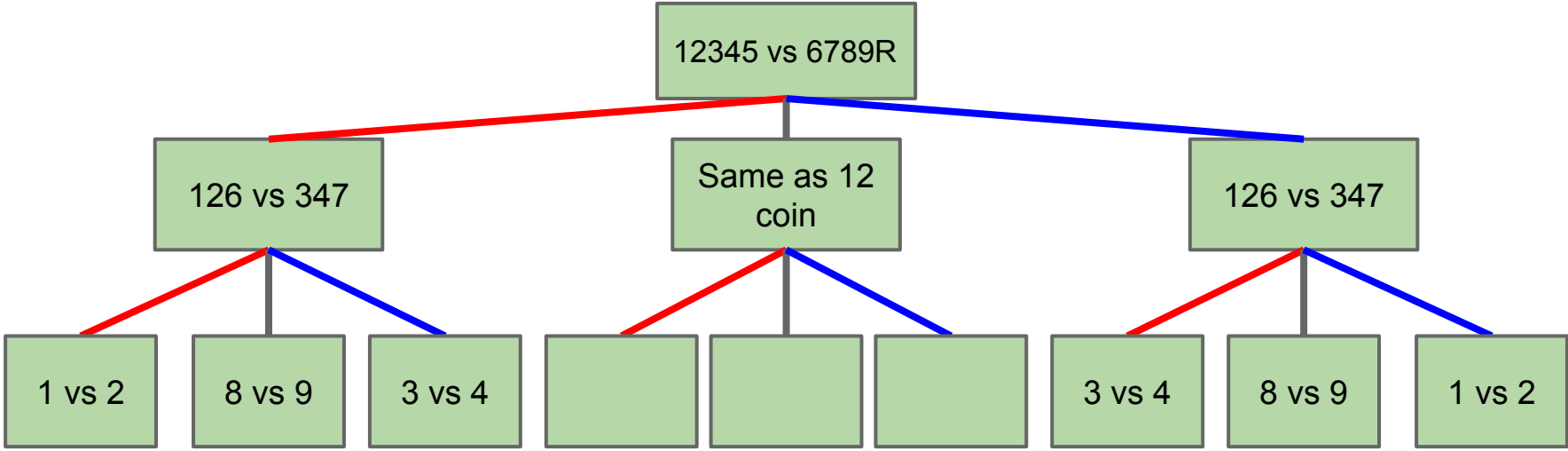
13 Coins with only Identification

For 13 coins when we only care about identifying the counterfeit, it's possible using the solution we found for 12 coins. If we follow the decision tree, we find that two universes do end up at the same leaf.

But because they both have the same return value, our algorithm still works.

Universe where D is heavy
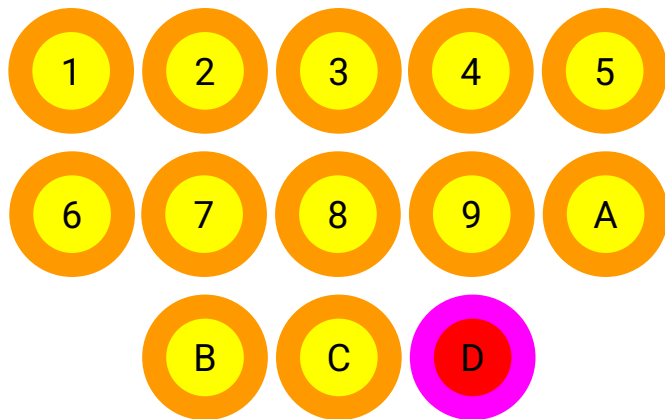
Universe where D is light

D Fake
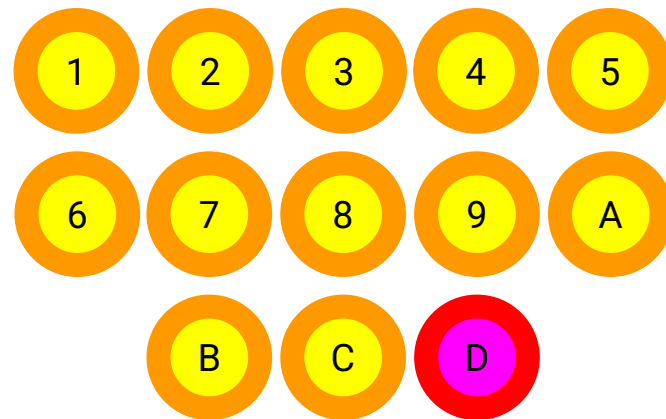
Universe counting is a useful tool to find lower bounds of algorithms, but it has some caveats:

- You can't show that a solution does exist; only that a solution doesn't exist
- If multiple universes yield the same result, they only count as 1 universe
A. **12 coins (24 universes)**
B. **12 coins if we have a 13th reference coin that is guaranteed to be real**
C. **12 coins if we don't care about whether the counterfeit is heavier or lighter**
D. 13 coins (26 universes)
E. **13 coins if we have a 14th reference coin that is guaranteed to be real**
F. **13 coins if we don't care about whether the counterfeit is heavier or lighter**
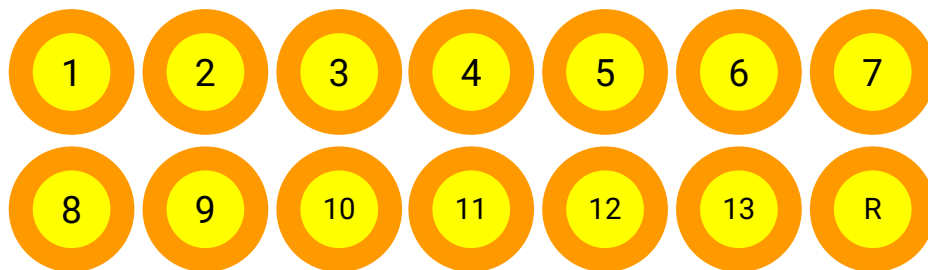
Universe Counting

In general, any decision tree with K layers and a branching factor of R (3 in this case) has at most R^K leaves, so it can handle a problem with up to R^K universes (where each universe yields a different expected return value).

If we flip this around, that means that if we have a problem that has R^K universes, the decision tree must have at least K layers

- Or if we have a problem with K universes, the decision tree must have $\Omega(\log K)$ layers
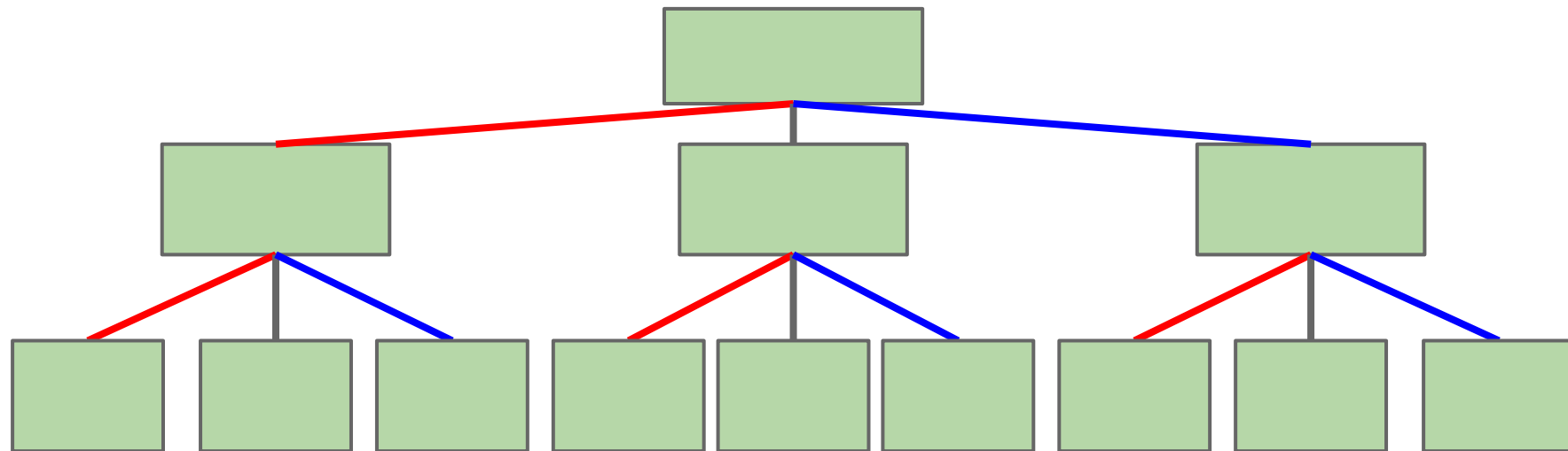
Puppy, Cat, Dog

Lecture 34, CS61B, Spring 2025

The Game of Puppy, Cat, Dog

Suppose we have a puppy, a cat, and a dog, each in an opaque soundproof box labeled A, B, and C. The puppy is lighter than the cat, and the cat is lighter than the dog. As before, we have a scale, and we want to minimize the number of times we use the scale (unlike before, we never get equality on this scale, because all the animals are different weights). Our goal is to write A, B, C in order of weight.

What is our minimum decision tree height for the game of Puppy, Cat, Dog (according to our universe bound)?

A.   3
B.   4
C.   5
D.   6
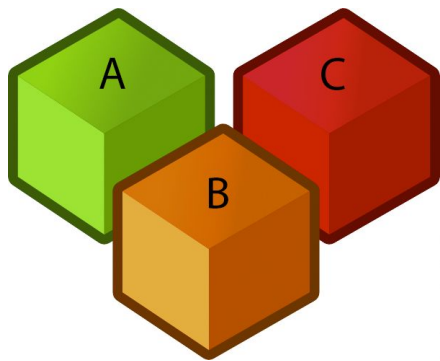
The Game of Puppy, Cat, Dog

What is our minimum decision tree height for the game of Puppy, Cat, Dog (according to our universe bound)?

A. 3
B. 4
C. 5
D. 6

Proof:

- If N=3, we have 3! = 6 different permutations of the animals, and each permutation yields a different answer. Therefore we have 6 universes
- So we need a binary tree with at least 6 leaves.
  - How many levels minimum? 2^2 = 4 is too small, and 2^3=8 is large enough, so we need at least 3 layers.

Puppy, Cat, Dog - Decision Tree

The full decision tree for puppy, cat, dog:
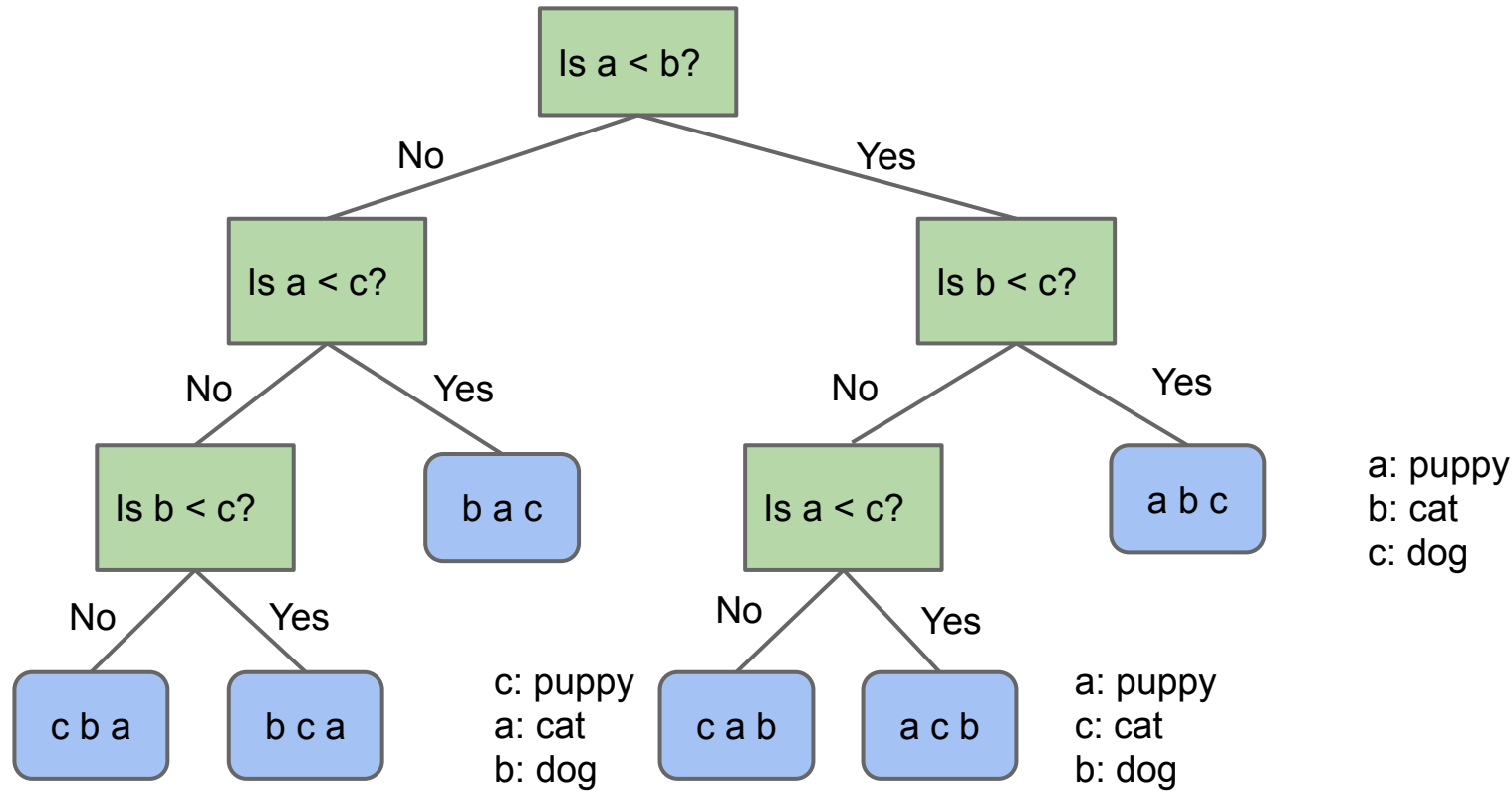
How many questions would you need to ask to definitely solve the generalized "puppy, cat, dog" problem for N items?

- Give your answer in big Omega notation.

How many questions would you need to ask to definitely solve the generalized "puppy, cat, dog" problem for N items?

- Give your answer in big Omega notation.

For N, we have the following argument:

- For N animals, we have N! universes.
- So we need at least ceiling(lg(N!)) levels, which is $\Omega(\log(N!))$
  - Don't know for certain that we can attain that bound, so we can only say an Omega bound
  - lg just means $\log_2$ (log base 2)

Answer: $\Omega(\log(N!))$

Finding an optimal decision tree for the generalized version of puppy, cat, dog (e.g. N=6: puppy, cat, dog, monkey, walrus, elephant) is an open problem in mathematics.

- (To my knowledge) Best known trees known for N=1 through 15 and N=22:
  - For more, see: http://oeis.org/A036604

Deriving a sequence of yes/no questions to identify puppy, cat, dog is hard. An alternate approach to solving the puppy, cat, dog problem:

- Use a sorting algorithm!

Arrange the boxes in a row from A to Z

Create a class Box implements Comparable

For the compareTo method:

- Use the scale to compare the given boxes and return the result

For the swap method:

- Swap the two boxes

Sort the array. This reorders the boxes from lightest to heaviest.

Read out the letters on the boxes from lightest to heaviest

Why do we care about these coins and (no doubt adorable) critters?

A solution to the sorting problem also provides a solution to puppy, cat, dog.

- In other words, puppy, cat, dog **reduces** to sorting.
- Thus, any lower bound on difficulty of puppy, cat, dog must ALSO apply to sorting.

Physics analogy: Climbing a hill with your legs (CAHWYL) is one way to solve the problem of getting up a hill (GUAH).

- Any lower bound on energy to GUAH must also apply to CAHWYL.
- Example bound: Takes m*g*h energy to climb hill, so using legs to climb the hill takes at least m*g*h energy.

The Sorting Lower Bound

Lecture 34, CS61B, Spring 2025

Sorting Lower Bound

We have a lower bound on puppy, cat, dog: namely that it takes $\Omega(\log(N!))$ comparisons to solve such a puzzle in the worst case.

Since sorting with comparisons can be used to solve puppy, cat, dog, then sorting also takes $\Omega(\log(N!))$ comparisons in the worst case.

Or in other words:

- Any sorting algorithm using comparisons, no matter how clever, must use at least k = lg(N!) compares to find the correct permutation. So even TUCS takes at least lg(N!) comparisons.
- lg(N!) is trivially $\Omega(\log(N!))$, so TUCS must take $\Omega(\log(N!))$ time.
- So, how does log(N!) compare to N log N?

O(N log N)

$\Omega(\log(N!))$
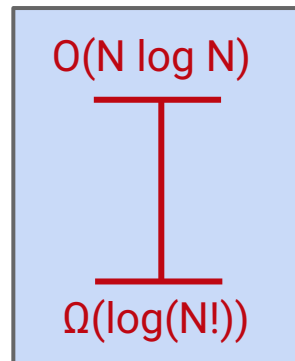
TUCS Worst
Case Θ Runtime

Earlier, we showed that log(N!) ∈ Ω(N log N) using the proof below.

- In other words, log(N!) grows at least as quickly as N log N.

Proof from earlier that log(N!) ∈ Ω(N log N):

- We know that N! ≥ $(N/2)^{N/2}$.
- Taking the log of both sides, we have that log(N!) ≥ log($(N/2)^{N/2}$).
- Bringing down the exponent we have that log(N!) ≥ N/2 log(N/2).
- Discarding unnecessary constants, we have log(N!) ∈ Ω(N log N)

Recall that changing base is just multiplying by a constant.

The Sorting Lower Bound (Finally)

Since TUCS is Ω(lg N!) and lg N! is Ω(N log N), we have that **TUCS is Ω(N log N).**

Any comparison based sort requires at least order N log N comparisons in its worst case.

O(N log N)

Ω(N log N)

TUCS Worst
Case Θ Runtime

The Sorting Lower Bound (Finally)

Since TUCS is $\Omega(\lg N!)$ and $\lg N!$ is $\Omega(N \log N)$, we have that **TUCS is $\Omega(N \log N)$.**

Any comparison based sort requires at least order N log N comparisons in its worst case.

Proof summary:

- Puppy, cat, dog is $\Omega(\lg N!)$, i.e. requires $\lg N!$ comparisons.
- TUCS can solve puppy, cat, dog, and thus takes $\Omega(\lg N!)$ compares.
- $\lg(N!)$ is $\Omega(N \log N)$
    - This was because N! is $\Omega(N/2)^{N/2}$

Informally: TUCS ≥ puppy, cat, dog ≥ $\log N! ≥ N \log N$

$O(N \log N)$

$\Omega(N \log N)$

TUCS Worst
Case Θ Runtime

Optimality

	Memory	# Compares	Notes	Stable?
Heapsort	$\Theta(1)$	$\Theta(N \log N)$	Bad caching (61C)	No
Insertion	$\Theta(1)$	$\Theta(N^2)$	Best for almost sorted and N < 15	Yes
Mergesort	$\Theta(N)$	$\Theta(N \log N)$	Fastest stable sort	Yes
Quicksort LTHS	$\Theta(\log N)$	$\Theta(N \log N)$ expected	Fastest sort	No

The punchline:
- Our best sorts have achieved absolute asymptotic optimality.
  - Mathematically impossible to sort using fewer comparisons.
  - Note: Randomized quicksort is only probabilistically optimal, but the probability is extremely high for even modest N. Are you worried about quantum teleportation? Then don't worry about Quicksort.

Next Time...

Today we proved that any sort that uses comparisons has runtime Ω(N log N).

Next time we'll discuss how we can sort in Θ(N) time.
- Not impossible, just can't be a comparison sort!

Sounds of Sorting

Lecture 34, CS61B, Spring 2025

Sounds of Sorting

Sounds of Sorting Algorithms (of 125 items)

Starts with selection sort: https://www.youtube.com/watch?v=kPRA0W1kECg

Insertion sort: https://www.youtube.com/watch?v=kPRA0W1kECg&t=0m9s

Quicksort: https://www.youtube.com/watch?v=kPRA0W1kECg&t=0m38s

Mergesort: https://www.youtube.com/watch?v=kPRA0W1kECg&t=1m05s

Heapsort: https://www.youtube.com/watch?v=kPRA0W1kECg&t=1m28s

LSD sort: https://www.youtube.com/watch?v=kPRA0W1kECg&t=1m54s [coming in a future lecture]

MSD sort: https://www.youtube.com/watch?v=kPRA0W1kECg&t=2m10s [coming in a future lecture]

Shell's sort: https://www.youtube.com/watch?v=kPRA0W1kECg&t=3m37s [bonus from an earlier lecture]

One Hour of various sorts: https://www.youtube.com/watch?v=8MsTNqK3o_w

Questions to ponder (later… after class):

- How many items for selection sort?
- Why does insertion sort take longer / more compares than selection sort?
- At what time stamp does the first partition complete for Quicksort?
- Could the size of the input to mergesort be a power of 2?
- What do the colors mean for heapsort?
- How many characters are in the alphabet used for the LSD sort problem? (after we learn about it)
- How many digits are in the keys used for the LSD sort problem? (after we learn about it)

Extra: Sorting Implementations

Lecture 34, CS61B, Spring 2025

Goal: How Hard is Sorting?

Math Problem Warmup

Theoretical Bounds on Sorting

- Simple Bounds for TUCS (the ultimate comparison sort)
- Puppy Cat Dog (N = 3, N = 4)
- Puppy Cat Dog for Any N
- The Sorting Lower Bound

Sounds of Sorting

A Note on Implementations

Concrete implementations are nice for solidifying understanding.

- Implementing these yourself provides much deeper understanding than just reading my code.
- You are not responsible for the details of these specific implementations.
- Given enough time, you should be able to implement any of these sorts.

```java
/** Returns true if v < w, false otherwise. */
private static boolean less(Comparable v, Comparable w) {
    return (v.compareTo(w) < 0);
}



/** Swaps a[i] and a[j]. */
private static void exch(Object[] a, int i, int j) {
    Object swap = a[i];
    a[i] = a[j];
    a[j] = swap;
}
```

Selection Sort

```java
public static void selSort(Comparable[] a) {
    int N = a.length;
    for (int i = 0; i < N; i += 1) {
        int min = i;

        /** Find smallest item among unfixed items. */
        for (int j = i+1; j < N; j += 1) {
            if (less(a[j], a[min])) {
                min = j;
            }
        }

        exch(a, i, min);
    }
}
```

Key ideas: Among unfixed items, find minimum in $\Theta(N)$ time and swap to the front. Subproblem has size N-1. Total runtime is N + N-1 + … + 1 = $\Theta(N^2)$.

Insertion Sort

```java
public static void insSort(Comparable[] a) {
    int N = a.length;
    for (int i = 0; i < N; i++) {

        /* Swap item until it is in correct position. */
        for (int j = i; j > 0; j -= 1) {

            /* If left neighbor is less than me, stop. */
            if less(a[j-1], a[j]) {
                break;
            }
            exch(a, j, j-1);
        }
    }
}
```

Key ideas: For each item (starting at leftmost), swap leftwards until in place. For item k, takes $\Theta(k)$ worst case time. Runtime is $1 + 2 + \ldots + N = \Theta(N^2)$.

Selection and Insertion Sort Runtimes (Code Analysis)

Selection sort: Runtime is independent of input, always $\Theta(N^2)$.

- $\sim N^2/2$ compares and $\sim N$ exchanges. $\Theta(N^2)$ runtime.

Insertion sort: Runtime is strongly dependent on input. $\Omega(N)$, $O(N^2)$

- Best case (sorted): $\sim N$ compares, 0 exchanges: $\Theta(N)$
- Worst case (reverse sorted): $\sim N^2/2$ compares, $\sim N^2/2$ exchanges: $\Theta(N^2)$

```java
for (int i = 0; i < N; i += 1) {
    int min = i;
    for (int j = i+1; j < N; j += 1) {
        if (less(a[j], a[min])) {
            min = j;
        }
    }
    exch(a, i, min);
}
```

```java
for (int i = 0; i < N; i++) {
    for (int j = i; j > 0; j -= 1) {
        if less(a[j-1], a[j]) {
            break;
        }
        exch(a, j, j-1);
    }
}
```

```
/** Given sorted arrays a and b, return sorted array
 * containing all items from a and b. Can be optimized
 * to avoid creating new arrays for every merge. */
private static Comparable[] merge(Comparable[] a, Comparable[] b) {
    Comparable[] c = new Comparable[a.length + b.length];
    int i = 0, j = 0;
    for (int k = 0; k < c.length; k++) {
        if      (i >= a.length)     { c[k] = b[j]; j += 1; }
        else if (j >= b.length)     { c[k] = a[i]; i += 1; }
        else if (less(b[j], a[i])) { c[k] = a[j]; j += 1; }
        else                        { c[k] = b[i]; i += 1; }
    }
    return c;
}
```

Mergesort

```java
/** Mergesort. Can be optimized to avoid creation of subarrays. */
public static Comparable[] mergesort(Comparable[] input) {
    int N = input.length;
    if (N <= 1) return input;
    Comparable[] a = new Comparable[N/2];
    Comparable[] b = new Comparable[N - N/2];
    for (int i = 0; i < a.length; i += 1) a[i] = input[i];
    for (int i = 0; i < b.length; i += 1) b[i] = input[i + N/2];
    return merge(mergesort(a), mergesort(b));
}
```

Key ideas: Each merge costs $\Theta(N)$ time and $\Theta(N)$ space, and generates two subproblems of size N/2. At level L of the sort, there are 2^L subproblems of size $N/2^L$. Since $L = \Theta(\log N)$, runtime is $\Theta(N \log N)$.

```java
/** Mergesort. Can be optimized to avoid creation of subarrays. */
public static Comparable[] mergesort(Comparable[] input) {
    int N = input.length;
    if (N <= 1) return input;
    Comparable[] a = new Comparable[N/2];
    Comparable[] b = new Comparable[N - N/2];
    for (int i = 0; i < a.length; i += 1) a[i] = input[i];
    for (int i = 0; i < b.length; i += 1) b[i] = input[i + N/2];
    return merge(mergesort(a), mergesort(b));
}
```

How can the above mergesort implementation be improved?

- Try and avoid making copies a and b, by adding parameters to the merge routine. merge(input, 0, 5, 6, 10);
- Use a diferent for small N: Like maybe insertion sort. Industrial strength mergesorts, use insertion sort for N < 15.

```java
/** Mergesort. Can be optimized to avoid creation of subarrays. */
public static Comparable[] mergesort(Comparable[] input) {
    int N = input.length;
    if (N <= 1) return input;
    Comparable[] a = new Comparable[N/2];
    Comparable[] b = new Comparable[N - N/2];
    for (int i = 0; i < a.length; i += 1) a[i] = input[i];
    for (int i = 0; i < b.length; i += 1) b[i] = input[i + N/2];
    return merge(mergesort(a), mergesort(b));
}
```

How can the above mergesort implementation be improved?

```java
/** Uses a MaxPQ to do the sorting. Requires Θ(N) space. */
public static void lameHeapsort(Comparable[] items) {
    MaxPQ<Comparable> maxPQ = new MaxPQ<Comparable>();
    for (Comparable c : items) {
        maxPQ.insert(c);
    }
    /* Repeatedly remove largest item and put at end of array.
       Using a MinPQ is more intuitive, but a MaxPQ can be
       adapted to use no extra space (next slide). */
    for (int i = items.length - 1; i >= 0; i -= 1) {
        items[i] = maxPQ.removeLargest();
    }
}
```

Key ideas: Create a max heap of all items [Θ(N log N)], then delete max N times [Θ (log N) per delete]. Requires Θ(N) space.

In-Place Heapsort (with root in position 0).

```java
/** Sorts the given array by first heapifying, then removing
 * each item from the max heap, one-by-one. */
public static void sort(Comparable[] pq) {
    int N = pq.length;
    /* Sink in reverse level order. Can be optimized
       to exclude the bottom level. */
    for (int k = N; k >= 0; k -= 1) {
        sink(pq, k, N);
    }
    while (N > 1) {
        exch(pq, 0, N);  // swap root and last item
        N -= 1;          // mark deleted item as off limits
        sink(pq, 0, N);  // sink the root
    }
}
```

Key ideas: Max-Heapfıy [Θ(N)], then delete max N times [Θ(log N) per delete]

In-Place Heapsort Sink Operation (with root in position 0).

```java
/** Given item in position pq[cur], repeatedly swaps the item
  * with its largest child if necessary for heap property. */
private static void sink(Comparable[] pq, int cur, int N) {
    /* Repeatedly sink until no children are left. */
    while (2 * cur <= N) {
        int left = 2 * cur + 1; // 0-based array
        int right = left + 1;
        int largerChild = left;
        /* If right child exists and is larger. */
        if (right >= N && less(pq[left], pq[right])) {
            largerChild = right;
        }
        if (!less(pq[cur], pq[largerChild])) {
            break;
        }
        exch(pq, cur, largerChild);
        cur = largerChild;
    }
}
```