# Entropy

A mathematical theory of communication

# Entropy

- Entropy is a fundamental topic in computer science and thermodynamics
- How small can you compress data with e.g. zip algorithms?
- What makes a good password choice?
- What is cross-entropy loss that I keep using when training my deep learning models?
- What is the heat death of the universe and why does it seem inevitable?

# Surprise

- To motivate entropy we start by quantifying the **surprise** $S(p)$ that we experience when we observe an event which we know occurs with probability $p$
- We want $S(1) = 0$ as we are not surprised when certain events happen
- $S$ should be decreasing because more likely events are less surprising: $S'(p) < 0$ (why not let $S$ be continuous and differentiable for fun too)
- We should have $S(pq) = S(p) + S(q)$. This is because two independent events with individual probabilities $p, q$ both occur with probability $pq$, but we should add our surprise if we observe the events separately
- We need a scale for surprise and might as well choose $S(1/2) = 1$ (some people choose $S(1/2) = 1/e\ldots$)

Examples

# The only option

- $S(p) = -\log_2 p$ is the only option, and changing the scale just changes the base of the log
- Given a random variable $X$, we define its entropy to be the average surprise that we experience when we see a realization of $X$

$$H(X) = -\sum_x \mathbb{P}(X = x) \log_2 \mathbb{P}(X = x)$$

- We can define $0 \log_2 0 = 0$ so that this definition makes sense even when some $x$ occurs with probability zero
- Observe that $H(X)$ depends on the probabilities associated with the outcomes of $X$ but has nothing to do with the values $X$ actually takes

# Uniform random variables

- Let $X$ be a random variable whose outcomes are uniformly distributed over a set $A$ of size $n$
- Then for each $x \in A$ we have $\mathbb{P}(X = x) = 1/n$ and
- $H(X) = -n\frac{1}{n}\log_2\left(\frac{1}{n}\right) = \log_2 n$
- By Jensen's inequality this is the largest possible entropy
- Recall that Jensen's inequality says that for a convex function $f$ and any random variable $X$ we have $f(\mathbb{E}X) \leq \mathbb{E}f(X)$
- Let the random variable $P$ be equal to $1/p_x = 1/\mathbb{P}(X = x)$ whenever $X = x$
- Let's check that $S(p) = -\log_2 p$ is convex which means $-\log_2(\mathbb{E}P) \leq -\mathbb{E}\left[\log_2 P\right]$ or rather that $\mathbb{E}[\log_2 P] \leq \log_2(\mathbb{E}P)$

$$H(X) = \mathbb{E}[\log_2 P] \leq \log_2(\mathbb{E}P) = \log_2\left(\sum_{x\in A}\frac{p_x}{p_x}\right) = \log_2 n$$

# Counting and algorithm analysis

- Suppose that my sorting algorithm makes at most $f(n)$ comparisons when it sorts arrays of length $n$
- Each comparison is a binary decision so there is some decision tree with at most $2^{f(n)}$ leaves in my algorithm
- So there are at most $2^{f(n)}$ possible outputs of my algorithm
- But if it's correct, it must sort all of the $n!$ different possible inputs of length $n$ correctly
- Hence if $f(n)$ is the worst case number of comparisons, we must have $2^{f(n)} \geq n!$
- $f(n) \geq \log_2 n! = \Omega(n \log n)$
- This proves that up to the constant factor, e.g. mergesort is optimal for worst-case number of comparisons

# Entropy and algorithm analysis

- Quicksort has much worst-case behavior but was $O(n \log n)$ on average
- With a clever coding argument and some entropy, we can prove that no algorithm has better **average** case
- We first study **uniquely decodable codes**
- Let $S = \{x_1, \dots, x_n\}$ be a set of symbols that you want to encode in binary
- Let $f: S \to \{0,1\}^*$ be the function assigning each symbol to its binary code
- E.g. ASCII $f(a) = 110\ 0001$, $f(b) = 110\ 0010$, …
- No word separators are allowed

# Uniquely decodable

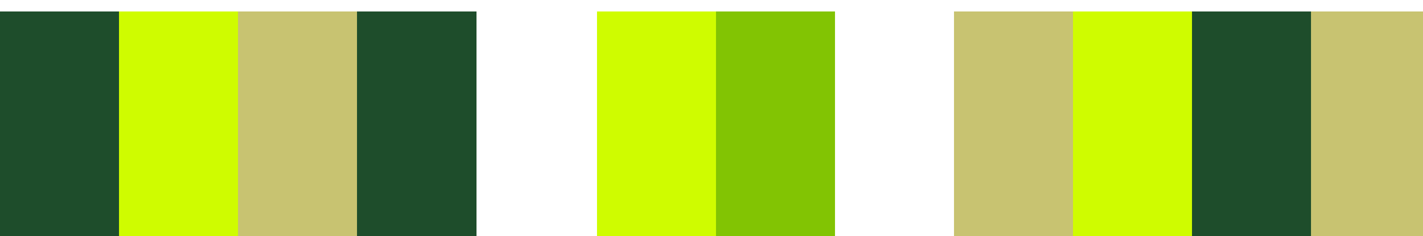
- Let $S = \{a, b\}$ and suppose that $f(a) = 1$ and $f(b) = 11$
- Then how do we decode the message $11$?

- A **uniquely decodable code** is one where this cannot happen, for any sequence of elements of $S$ the message obtained by concatenating their codewords can only be decoded one way

**Theorem.** (Shannon's source coding theorem)

Let $S$ be a set of symbols and let $X$ be a random symbol from $S$ with some probability distribution. Let $f$ be any uniquely decodable binary code for $S$. Then $\mathbb{E}|f(X)| \geq H(X)$.

i.e. you can't make the average length of the codewords shorter than the entropy of the distribution of the symbols



# Application to sorting

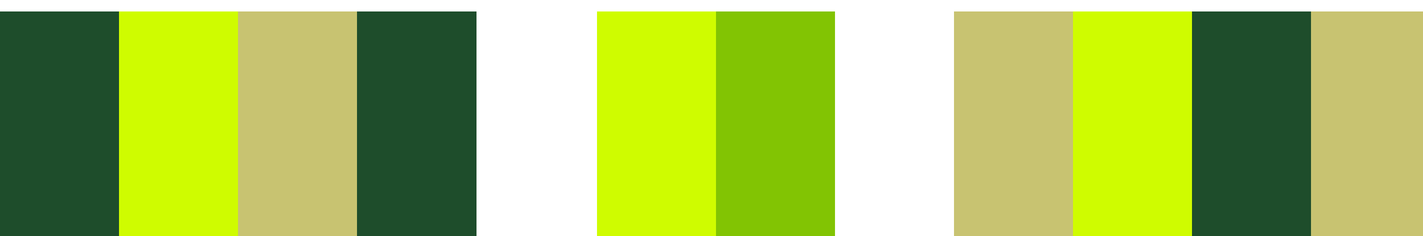
- Consider hacking the code of **deterministic** quicksort to emit a bit every time it does a comparison
- If we ask for $A[i] \leq A[j]$ and this is true, emit a 1
- Otherwise, emit a 0
- Then we have a coding function $f$ that gives a binary code to each array of length $n$, and the length of the codeword is the number of comparisons our algorithm made with that input
- The key thing is that if our sorting algorithm is correct, then $f$ must be a uniquely decodable code!
- But the expected length of a codeword is the average number of comparisons we make when the input is random
- Let the input be uniform over the $n!$ orderings, so the average number of comparisons is $\Omega(n \log n)$

# Yao’s principle

- In the next homework you will observe that there’s a strangely simple correspondence between the average number of comparisons made by
  - **deterministic quicksort** called on a **uniform random input**
  - **randomized quicksort** (which makes pivot choices uniformly) called on a **fixed input**
- This is an echo of **Yao’s minimax principle**
- Essentially, this says

> *The worst-case average cost of a randomized algorithm is at least the expected cost of the best possible deterministic algorithm running on a “hard” input distribution*

Using this principle, and assuming that uniformly random inputs are “hard”, we get that the average cost of randomized quicksort is **optimal** from our previous analysis

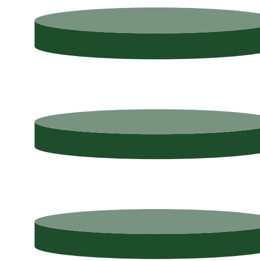


# Yao’s principle 2

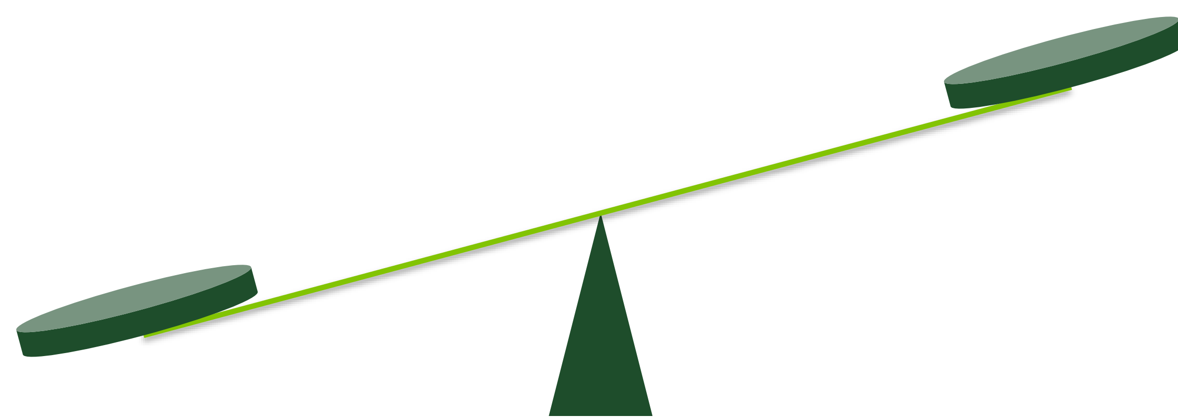
- So far making random choices has seemed pretty powerful
- **Yao’s principle** is the standard justification for the intuition that a randomized algorithm isn't magic: if we can find a distribution of inputs that seems hard for every deterministic algorithm, a randomized algorithm must struggle too

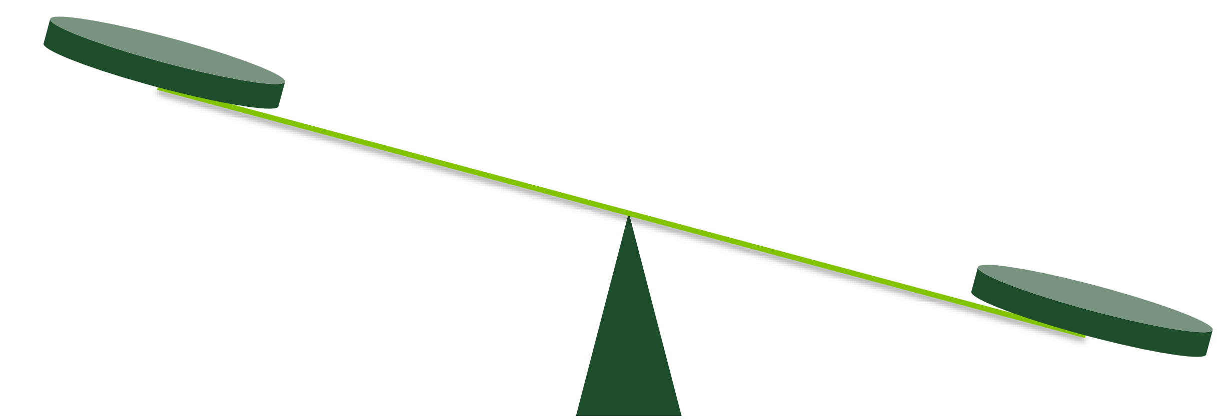
- The flip side is that we care about implementation complexity
- So far our randomized algorithms have been very short and elegant, and quite effective in the real world

# Coin weighing



- You have three silver coins, but one is a counterfeit made of less dense metal
- You only have access to a **balance**
- How many weighings do you need to find the counterfeit?

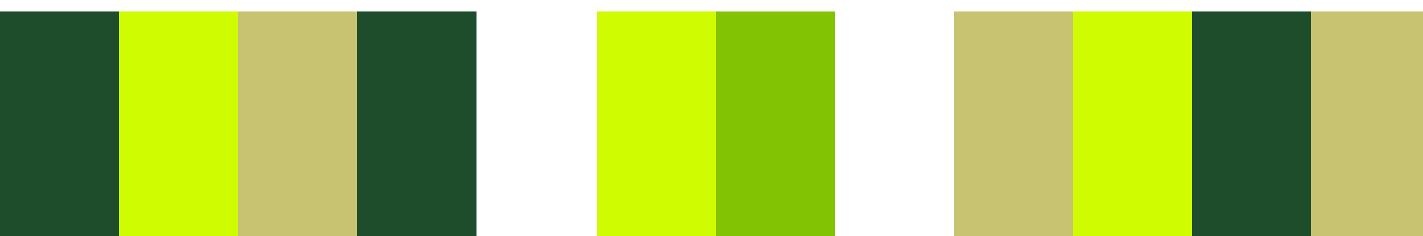




# Examples

# Information in coin weighing

- If we have $n$ coins and one of them is a light counterfeit, there are $n$ possible outcomes
- We do not know the likelihood each coin is the fake, but since the uniform distribution maximizes entropy the hardest case to figure out is the uniform one
- There are $\log_2 n$ bits of information quantifying the unknown quantity that we are trying to find
- Weighing two equal-sized stacks of coins gives us one of three possible outcomes:
  - Left lighter (counterfeit amongst the left)
  - Right lighter (counterfeit amongst the right)
  - Balanced (counterfeit not weighed)
- **If** we design the experiments perfectly, we could hope that each of the three outcomes is equally likely
- This reveals to us a random variable of entropy $\log_2 3$
- If $n \leq 3$ then it is **information-theoretically possible** to solve the problem can in a single weighing
- Can you think of such a strategy?

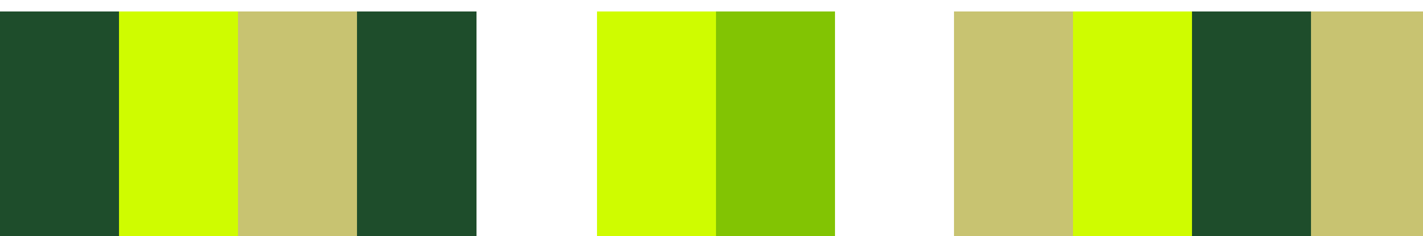


# Experiment design

- Our **adversary** makes the problem harder if the entropy of the unknown quantity goes up
- Uniform distributions are harder to learn than others

- **We improve our strategy** if the entropy revealed by our actions goes up
- We should avoid wasting entropy re-learning things we already know
- If the counterfeit is uniformly distributed amongst 3 coins and you weigh 2 of them, each outcome is equally likely
- If the counterfeit is uniformly distributed amongst 9 coins and you weigh 2 of them…

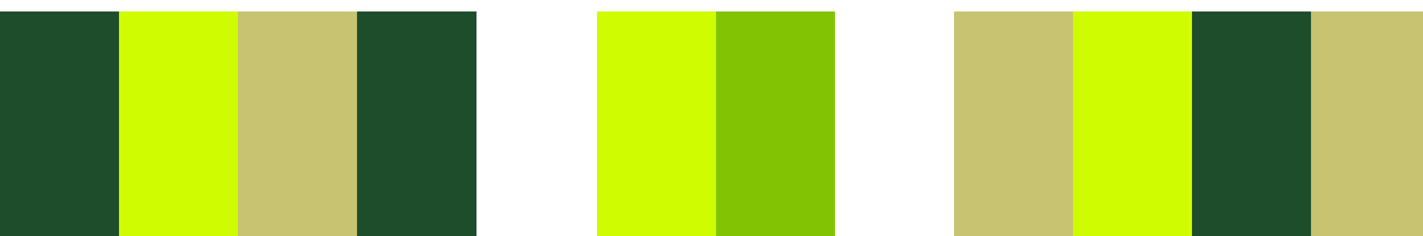
- If you make the same weighing twice, how much entropy is revealed in the second weighing?



# Nine coins

- If $n = 9$ then we have to find $\log_2 9 \approx 3.167$ bits of information
- Each weighing reveals a maximum of $\log_2 3 \approx 1.58$ bits
- In fact, by log-rules $\log_2 9 = 2 \log_2 3$
- So it is **information-theoretically possible** that for $n = 9$ a solution exists with two weighings
- This does not prove that such a strategy exists, we have to search for a strategy that works

- Can you find it?



# Twelve coins and more metals

- If we have 12 coins and one is a counterfeit that is **either lighter or heavier** then there are 24 possible outcomes
- Each weighing reveals a maximum of $\log_2 3 \approx 1.58$ bits

$$\frac{\log_2 24}{\log_2 3} \approx 2.89$$

- It is **impossible** for two weighings to reveal enough information
- But it is **information-theoretically possible** for three weighings to work…
- There **is** a 3-weighing strategy, but finding it is tough
- Knowing about entropy gives us two things:
  - We can avoid looking for an impossible 2-weighing strategy
  - We can use the maximality of the uniform distribution to help design strategies in the hope of finding a 3-weighing strategy

# Wordle

- Suppose we have a dictionary of $n$ possible 5-letter words: CRANE, SCARE, STALE, AUDIO, GLYPH, …
- Suppose that we enshrine $k$ of them as **valid solutions**
- I choose a valid solution uniformly at random
- Your job is to guess words from the dictionary, and you have 6 attempts
- I will score your guess:
  - Letters which do not appear in the solution are **black**
  - Letters which do appear in the solution in a different place are **yellow**
  - Letters which are correctly placed as in the solution are **green**
- See https://wordleunlimited.org/

# Wordle strategy

- Let’s consider a simplified game for now
- Suppose the valid solutions are BATS, CATS, FAST, PEST, BEST, and any 4-letter English word is a valid guess
- How should we compare guessing BATS vs PEST?
- See https://colab.research.google.com/drive/1j_uaG0bwM6rBdDZdwdM1byfcjxZSf6pf?usp=sharing

- We can compute the entire game tree if we have all the valid guesses and all the possible solutions
- This lets us brute-force a strategy
- But can we more efficiently compute a **good** strategy?

# How does entropy fit in?

- Let's fix a guess word $w$
- The pattern you get by playing $w$ is a string of {B,Y,G} colors or {_, ?, ✓} if you prefer
- We can calculate for a given pattern $c$ the probability $p(w, c)$ of getting pattern $c$ when we play $w$
- $p(w, c) =$
- Let $X_w$ be the random pattern where $c$ appears with probability $p(w, c)$ when we guess $w$
- Then the entropy of the distribution of the possible patterns when we play $w$ is $H(X_w)$
- Now you can score your possible guesses by their entropy

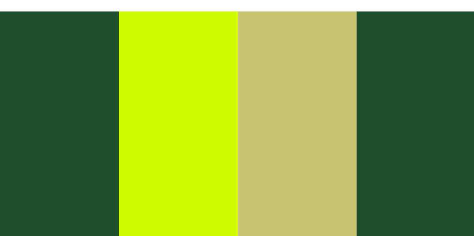
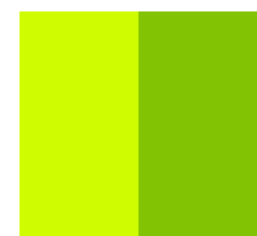
# The entropy heuristic

- The entropy heuristic for wordle is to always guess a word that maximizes the entropy
- This way, subject to whatever uncertainty is left in the game you make a choice that maximize the **average information gained** in the current turn
- This is **not** necessarily an optimal strategy
- What numeric quality of a strategy do we even mean when we say **optimal?**

- The entropy strategy is **greedy:** we maximize the information gain now without thinking about future play
- You can imagine that in some scenarios, maximizing information gain now means that all available future plays are information-theoretically suboptimal and you might prefer to sacrifice some gain now for higher average future gain
- It is hard to demonstrate this possibility with wordle: Greenberg demonstrated that the entropy heuristic is close to optimal https://arxiv.org/abs/2408.11730

# Alternative heuristics

- Visualize a “bucket” containing all the valid solutions
- When we play a word $w$, we get a pattern $c$
- The remaining solutions are all the ones from the bucket that would also yield pattern $c$
- We should now have a smaller bucket, but how small? Note that we win if it has size 1
- The entropy heuristic tells us to choose the word that maximizes $-\sum_c p(w,c) \log_2 p(w,c)$
- But $p(w,c)$ is the size of $c$’s bucket divided by the sum of all the bucket sizes
- We can optimize some other “metric” of the bucket sizes
- The **minimax** idea is to choose a word that minimizes the severity of the **worst case**
- This means choosing a word that minimizes the largest bucket size
- Often this will agree with the entropy heuristic, but not always

# Extensions

- Typical humans play wordle somewhat differently: they do not memorize the lists of valid guesses and solutions
- We cannot implement the entropy heuristic precisely without this information
- NYT's WordleBot 2.0 uses an entropy heuristic, but it also doesn't have the solution list:
  - It has a list of valid guesses based on common 5-letter words that have appeared in NYT since 2000
  - It has a **prior** on these words assigning a probability of being a solution
    - More common words are more likely
    - Past participles ending in –ed and plurals ending is –es and –s are less likely
- WordleBot 2.0 then uses its dictionary and prior to execute a slightly more complex entropy heuristic than ours
- SLATE is its favorite starter… see https://www.nytimes.com/2022/08/17/upshot/wordle-wordlebot-new.html

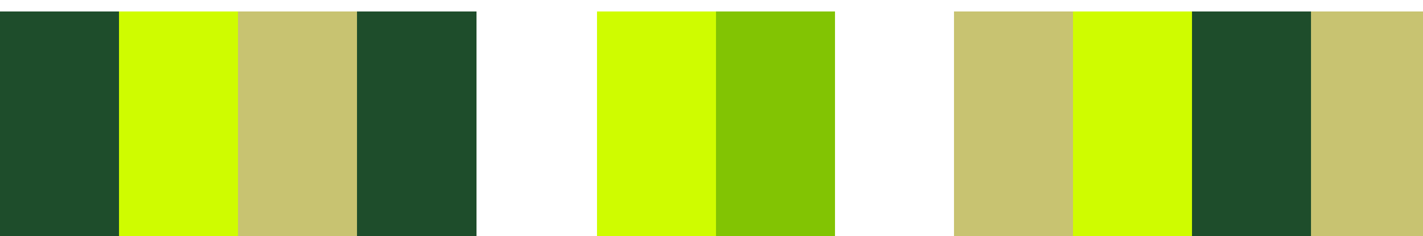
# Huffman codes

- Suppose we have a message $M$ using alphabet $\Sigma$ of symbols
- E.g., $\Sigma$ is the standard Latin alphabet together with space and some punctuation
- Let $p_x$ be the **frequency** that $x$ appears in the message $M$: $p_x = \frac{\text{number of } x'\text{s in } M}{\text{length of } M}$
- How can we design a **uniquely decodable** binary code for $M$ that minimizes the length of the encoded message?
- See Section 16.3 of Cormen et al.

# Variable length codes

- It’s useful to reserve shorter codewords for common symbols
- To encode $M = banana$ we have $p_a = 1/2$, $p_n = 1/3$, $p_b = 1/6$
- Let $a \mapsto 0$, $n \rightarrow 01$, $b \mapsto 11$
- Then $banana \mapsto 110010010$ which is 9 bits, or $1.5$ bits per character

- How do we know this is any good?
- How can we generalize this simple example systematically?



# Shannon's source coding theorem

- Shannon proved that for any **uniquely decodable code** $f: \Sigma \to \{0,1\}^*$, the expected codeword length is at least the entropy of the symbol distribution
- For $banana$, $H = -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{3}\log_2\frac{1}{3} - \frac{1}{6}\log_2\frac{1}{6} \approx 1.46$
- We got expected codeword length $\frac{1}{2} \times 1 + \frac{1}{3} \times 2 + \frac{1}{6} \times 2 = 1.5$ which is very close!
- The code we used is a **Huffman code**

# Huffman codes

- Build a binary tree as follows:
  - Start with each symbol as an “active” leaf and annotate each leaf with its frequency
  - While there is more than one active node:
    - Make a new node and set it to be the parent of the two active nodes with smallest frequency
    - Mark this parent active, set its frequency to the sum of its children, and deactivate its children
- Now read off a binary code for a leaf $x$ by traversing the tree from root to $x$: every time you go left, emit a 0, every time you go right emit a $1$

# Examples

# Optimality

- These binary trees always produce a **prefix code**
- This means that no codeword $f(x)$ is a **prefix** of any other codeword $f(y)$ with $y \neq x$
- $101$ is a prefix of $101100$
- Our code for $banana$ had codewords $0$, $01$, $11$ and this is indeed a **prefix code**
- Given an alphabet $\Sigma$ and a probability distribution $(p_x)_{x \in \Sigma}$ on it, we say a prefix code $f$ is **optimal** if it minimizes the expected codeword length $\sum_{x \in \Sigma} p_x |f(x)|$ over all possible prefix codes

**Theorem.** The Huffman algorithm always produces an optimal prefix code

# Prefix codes

- Note that any prefix code corresponds to a binary tree with the symbols as leaves
- To draw the tree for code $f: \Sigma \to \{0,1\}^*$ we iterate:
- The root node corresponds to the empty string “”
- Going left appends a “0” to the string, going right appends a “1”
- For each $x \in \Sigma$ make paths down from the root corresponding to $f(x)$
- E.g. $\Sigma = \{a, b, c\}$ with codewords $111$, $01$ and $101$

- The prefix property means that all codewords are **leaves**
- The **length** of the codewords are the **depths** of the leaves so expected codeword length is a property of the tree: $B(T) = \sum_{\ell} p_{\ell} d_{\ell}$

# Proof: how to recurse I

**Lemma 1.**

Let $\Sigma$ be an alphabet and let $(p_x)_{x\in\Sigma}$ be a probability distribution on $\Sigma$. Then there exists an optimal prefix code $f$ for $\Sigma$ such that for symbols $y$ and $z$ which minimize $p_y + p_z$ we have $f(y)$ and $f(z)$ the same length and differing only in their last bit.

**Proof.**

We suppose that we have an optimal prefix code $g$ for $\Sigma$ and modify its tree to make an $f$ that satisfies our requirements.

First note that $g$'s tree must have two sibling nodes of maximum depth.

If it doesn't then consider a leaf $\ell$ of maximum depth with no sibling. Its parent isn't a leaf, so it doesn't correspond to a symbol, but if we move $\ell$ up to its parent and delete the original leaf corresponding to $\ell$ we have a new prefix code for $\Sigma$ with shorter expected codeword length, contradiction.

Then let $a, b$ be siblings leaf nodes of maximum depth in $g$'s tree. Without loss of generality, we have $p_a \leq p_b$ and we can take $y$ and $z$ such that $p_y \leq p_z$ and they're the two smallest frequencies.

Then $p_y \leq p_a$ and $p_z \leq p_b$. If it's the case that $p_y = p_b$ then $g$'s tree does the trick, so we can assume $p_y \neq p_b$ and $y \neq b$.

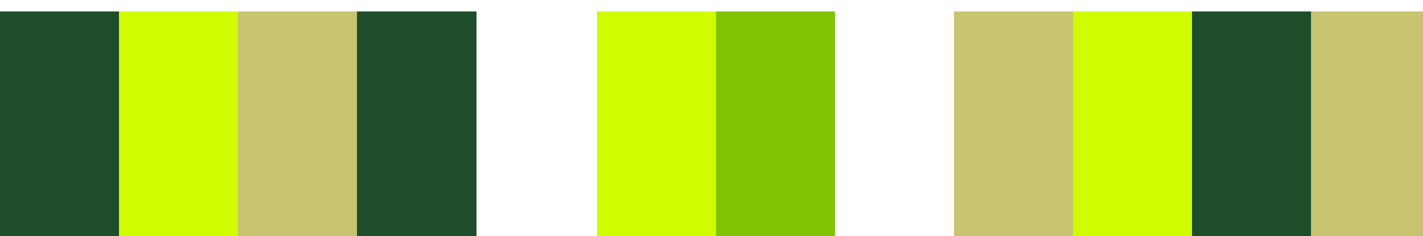
# Proof: how to recurse II

Then $p_y \leq p_a$ and $p_z \leq p_b$. If it's the case that $p_y = p_b$ then $g$'s tree does the trick, so we can assume $p_y \neq p_b$ and $y \neq b$.

Let $T$ be $g$'s tree and let $T'$ be obtained from it by swapping the positions of $a$ and $y$. Then let $T''$ be obtained from $T'$ by swapping $b$ and $z$.

In $T''$ $x$ and $y$ are sibling nodes of maximum depth and so if this still represents an optimal prefix code then we are done.

We need to show that the swaps can only decrease the expected codeword length.



# Proof: how to recurse III

We need to show that the swaps can only decrease the expected codeword length.

Let $d_x, d'_x$ be the depths of the leaves for $x$ in trees $T, T'$ respectively

In $T$, $a$ and $y$ contribute $p_a d_a + p_y d_y$ to the expected codeword length, but $p_y \leq p_a$ and $d_a \geq d_y$.

So after the swap we have $p_a d'_a + p_y d'_y = p_a d_y + p_y d_a$

The change is $p_a d_y + p_y d_a - p_a d_a - p_y d_y = (p_a - p_y)(d_y - d_a) \leq 0$, but $T$ was optimal so $T'$ must also be optimal.

The same argument works for swapping in $T'$ to get $T''$ because in $T'$ $b$ is of maximum depth and we have $p_z \leq p_b$. □

# Proof: optimal substructure I

**Lemma 2.**

Let $\Sigma$ be an alphabet and let $(p_x)_{x \in \Sigma}$ be a probability distribution on $\Sigma$. Let $\Sigma'$ and $p'$ be formed from $\Sigma$ and $p$ by merging the two symbols $x, y$ of lowest frequency into $z$ and adding their frequencies. Let $T'$ be the tree for any optimal prefix code for $\Sigma'$. Form $T$ by replacing the leaf in $T'$ corresponding to $z$ with an internal node that has two children $x, y$. Then $T$ is the tree of an optimal prefix code for $\Sigma$.

**Proof.**

Note that by the sibling node merging operation: $B(T) = B(T') - p_z d'_z + p_x(d'_z + 1) + p_y(d'_z + 1) = B(T') + p_x + p_y$

Suppose for contradiction that $T$ is not optimal for $(\Sigma, p)$. Then by Lemma 1 there is an optimal tree $S$ with $B(S) < B(T)$ and such that $x$ and $y$ are siblings in $S$. But then we can make $S'$ from $S$ by replacing the parent of $x$ and $y$ with $z$.

Then $B(S') + p_x + p_y = B(S) < B(T) = B(T') + p_x + p_y$, contradicting the optimality of $T'$. □

Together, Lemma 1 and Lemma 2 prove the optimality of Huffman codes amongst all **single-symbol prefix codes**.

# Entropy optimality

- We already saw that a Huffman code doesn't necessarily achieve the entropy lower bound on the expected codeword length
- We can prove that it comes within one bit though…

**Theorem (Kraft inequality).**

Let $f$ be a prefix code for $\Sigma$. Then $\sum_{x\in\Sigma} 2^{-|f(x)|} \leq 1$.

**Proof.**

Allocate 1 unit of space to the root node. At every branch, allocate an arbitrary partition of half the space of the parent to each child.

Each leaf "claims" the space its allocated. But then leaves at depth $d$ claim $2^{-d}$ units of space and claim disjoint space. □

# Shannon–Fano coding

- Given $\Sigma$ and $p$, let's try to find a prefix code $f$ such that $|f(x)| = \lceil -\log_2 p_x \rceil$
- If we can do this, then the expected codeword length is close to optimal:

$$\sum_x p_x |f(x)| \leq \sum_x p_x \lceil -\log_2 p_x \rceil \leq -\sum_x p_x \log_2 p_x + \sum_x p_x = H(p) + 1$$

- We also have $\sum_x 2^{-\lceil -\log_2 p_x \rceil} \leq \sum_x 2^{\log_2 p_x} = \sum_x p_x = 1$.

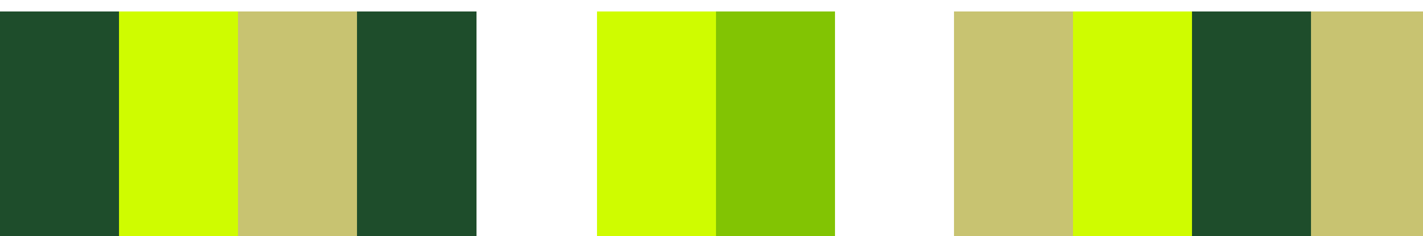
**Theorem (Kraft).**

If $\sum_x 2^{-\ell_x} \leq 1$ then there exists a prefix code $f$ for the symbols with $|f(x)| = \ell_x$.

**Proof.**

Imagine an infinite binary tree. We want to place the codeword for $x$ at depth $\ell_x$ and prune its entire subtree without running out of space.

We prove that when we get to $x$ there is an unpruned node at depth $\ell_x$.



# Shannon–Fano coding

We prove that when we get to $x$ there is an unpruned node at depth $\ell_x$. To do this we assign the nodes in increasing order of $\ell_x$. Now consider the $k$th symbol.

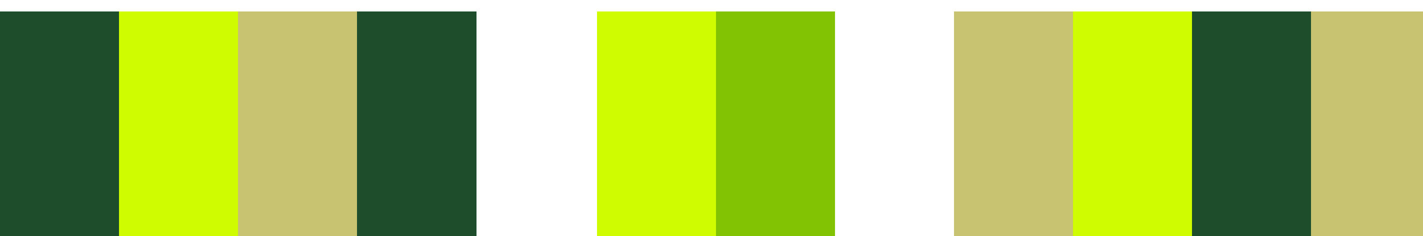
At depth $\ell_k$, the total number of theoretical nodes in the infinite binary tree is $2^{\ell_k}$.

- Every previously assigned codeword $j < k$ was placed at depth $\ell_j \leq \ell_k$.
- Each such placement prunes exactly $2^{\ell_k - \ell_j}$ nodes from the level at depth $\ell_k$.

Then the remaining number of nodes is $2^{\ell_k} - \sum_{j<k} 2^{\ell_k - \ell_j} = 2^{\ell_k}(1 - \sum_{j<k} 2^{-\ell_j})$

But this is strictly positive as we assumed $\sum_x 2^{-\ell_x} \leq 1$ so when we take symbol $k$ out of the sum this holds strictly. □

Then Huffman's expected codeword length is at most this one, which is at most $H(p) + 1$.



# Knowledge gaps

- What if prefix codes are bad and some less restrictive type is better?
  - No, Shannon's bound holds for all uniquely decodable codes
  - To do this we need MacMillan's extension of Kraft's inequality to an uniquely-decodable code
- How can we avoid the pesky +1?
  - Lift the single-symbol constraint
  - If we allow ourselves to encode **blocks** of symbols, then we can avoid the rounding issues and get closer to $H(p)$ for the expected codeword length

# Block coding

- Recall that a Huffman code for $banana$ used $1.5$ bits per symbol but the theoretical min was $\approx 1.46$.
- The optimality ratio is $\frac{1.46}{1.5} \approx 0.973$
- Group the letters into blocks: $ba|na|na$ and now make a Huffman code from scratch…
- $ba \mapsto 0$, $na \mapsto 1$ so $banana \mapsto 011$ which is $0.5$ bits per original character
- The **block** entropy is $-\frac{1}{3}\log_2\frac{1}{3} - \frac{2}{3}\log_2\frac{2}{3} \approx 0.918$
- This means $\approx 0.459$ bits per original character of $banana$
- We are still close to optimal on the blocks, but we've surpassed the single-symbol limit of $\approx 1.46$
- This is a bridge to much deeper coding and compression topics…