# Information-Theoretic Approaches to Battleship and Hangman

Arya Maheshwari and Aditya Singhvi
Ms. Aiyer
Advanced Topics in Math: Information Theory 1
January 12th, 2021

## §1. Introduction

In this paper, we present information-theoretic approaches to the games of Battleship (Section 2) and Hangman (Section 3). Both problems are modeled on a system level using the following fundamental model of the flow of information.


As depicted, there are five main parts to the system: (1) source, (2) encoder, (3) channel, (4) decoder, (5) receiver. The broad goal is to transmit some information between the source and the receiver using the fewest number of bits possible, where each bit represents a unit of information. Each section of the above framework is delineated below.

1. The **source** is generally represented as a random variable **X**, with some corresponding probability distribution **p(X)** of its possible values, which describes the information we wish to convey.
2. The **encoder** here is a combination of the source and channel encoders. The purpose of the source encoder is to convert the values of **X** into bits in a manner that minimizes the expected length of the number of bits required to encode **X** based on its associated distribution. If the channel that the information is being transmitted over contains noise, a channel encoder could also be present to add redundant bits of information to the source encoder to create a tolerance for a specified number of errors that could occur during transmission.
3. The **channel** represents the piece of the system where bits are actually transmitted from the encoder to the decoder. Physically, this could be something like a wire, a wireless signal, or even simply a sound wave. The channel may contain noise, which could lead to the intended encoded bits being modified in some way before they reach the decoder.
4. The **decoder** receives the information sent through the channel, consisting first of the channel decoder and then of the source decoder. The channel decoder is the counterpart to the channel encoder, "undoing" the data buffer created by the encoder. Similarly, the source decoder acts as the counterpart to the source encoder, taking the compressed bit-based representation and reconstructing from it the value of an output random variable **Y**.

5. Finally, this output is passed to the **receiver**.

# §2. Battleship

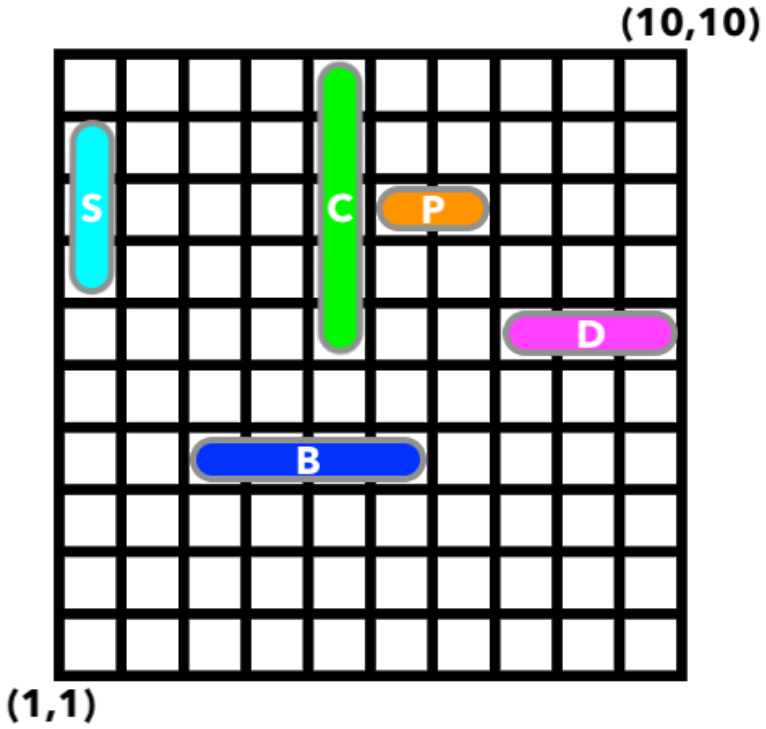
## §2.1 Game Objective

The goal of the game of Battleship, and specifically the problem statement presented in this project, is to determine the correct configuration of ships on a Battleship board using the fewest number of queries. The specifics of this terminology and details of the game are described in the next section.

## §2.2 Game Rules

In battleship, there are generally two players each with a board of ships (hidden to the other player) trying to fully determine the locations of their opponent's ships before their opponent succeeds in this task. For the purposes of this project, then, we focus on the guesses of one player (labeled **G**, the guesser) against the opponent **O**, in an attempt to find an optimal strategy. More specifically, our code acts as **G**, while Ms. Aiyer (whose board configuration **G** is trying to guess) is **O**.

Each player has five ships: a cruiser, a battleship, a destroyer, a submarine, and a patrol boat. We reference these ships by their first character, and they have the following lengths: **c** = 5, **b** = 4, **d** = 3, **s** = 3, **p** = 2. These ships can be placed in any non-overlapping configurations (but immediate adjacent placement is allowed) on a $\mathbf{10 \times 10}$ grid. The conventions of this project stipulate that the lower-left corner be referred to as location $\mathbf{(1, 1)}$, while the upper-right corner is referred to as $\mathbf{(10,10)}$. One example configuration is shown below.




Guessing proceeds with **G** querying a location **L** on this $\mathbf{10 \times 10}$ grid (defined by a row and column). **O** will then respond in one of three ways:

1. A *miss*, indicating that there was no ship at L;
2. A *sink*, indicating that there was a ship at L and that with this guess, that ship has now been sunk;
3. A *hit*, indicating that there was a ship at L, but *it has not been sunk* (i.e. not fully discovered).

For the purposes of this project, these three cases will be indicated with the following phrases: **(1)** "Miss", **(2)** "Hit and Sunk", **(3)** "Hit." Notably, per the problem statement, **G** is ***not*** given the specific coordinates or length of a ship when **Case (2)**, or "Hit and Sink" occurs—it is just given that specific phrase, which simply conveys the information that *some* ship has been sunk.

Each of those guesses constitutes one query, and play continues until all ships have been sunk and **O**'s configuration has therefore been fully discovered. The total number of queries required to end the game is the quantity we aim to minimize.

## §2.3 Information-Theoretic Lens

We frame the Battleship problem using the information-theoretic model presented in the Introduction section using the following correspondences. Note again that player **O**, for our project, is Ms. Aiyer.

The first step to thinking in terms of information-theoretic systems is to specify the overall information flow. In our version of the Battleship problem, the guesser **G** is trying to ascertain the configuration of **O**'s ships: in other words, information about **O**'s ships is being conveyed to player **G**. This further aligns with the idea of encoding, as the goal of the problem is to minimize the number of queries required to fully convey this information, which is tantamount to minimizing bits required to convey the source information in the general information theory diagram. Specifically, we connect each building block in the information theory diagram to parts of the Battleship game as follows:

1. **Source:** Player **O**'s battleship board: it contains the information (the configuration of battleships) that we are iteratively trying to convey.
2. **Encoder:** A combination of player **O** and the queries being made by **G**: based on the queries, player **O** takes the source data and represents it in the format in which it will be transmitted (i.e., responses to the queries).
3. **Channel**: Player **O**, specifically as the actor who conveys the response to the query—for our project, transmission over the communication channel is effectively Ms. Aiyer responding to our query.
4. **Decoder:** The central algorithm in our code (which represents **G**), which takes the information transmitted over the channel and manipulates it to gain information about the source random variable/information (i.e., slowly determining the configuration of **O**'s battleships).
5. **Receiver:** The user-end battleship board, so **G**'s board, as the information from **O**'s board is ultimately being used to add information to **G**'s board until battleship configurations are completely determined (the message is completely transmitted) and the game ends.

## §2.4 Theory: Framework and Formalized Argument

Continuing with the information-theoretic lens, we suggest that in order to minimize the number of queries, we want to maximize information about the board (quantified later). However, because the probability distributions of the board (whether or not a ship is at a certain location) is dynamically evolving — updated every time a guess is marked as a hit or a miss — it becomes difficult to chart out a globally-optimal query path at the beginning. As a result, we opt for a **greedy** approach: pick the myopically/locally optimal query, where optimal means giving you the most information in one step.

We characterize the information we have as the net self-information of the remaining options. Quantitatively, this is $\mathbf{h} = -\log(\Sigma\, p_i)$, summed across all remaining states $\mathbf{i}$ and their associated probabilities $\mathbf{p_i}$. For example, with no guesses, all possible configurations are still possible—we have not eliminated any—so $\Sigma p_i = 1$, so information is $0$.

With each move, we want to gain information: which can be thought of as maximizing $\mathbf{h}$ (the self-information of remaining possible states, considered together). Based on our earlier argument for an entirely greedy approach, we can pick a certain stage of the game (say $\mathbf{x_k}$) with its corresponding state-space (say $\mathbf{s_k}$) of remaining possible configurations. We can then evaluate conditional probabilities to think about the idea of "iterative" self-information of a state: in other words, the original probability $\mathbf{p_j}$ of a given configuration $\mathbf{j}$, now *given* that $\mathbf{s_k}$ is our possible state-space:

$$\mathbf{p_j'} := p_j | s_k$$

Now, considering the upcoming stage of the game $\mathbf{x_{k+1}}$ after we make the next move (whatever that may be), we want to arrive at the stage with the most self-information, so we maximize $\mathbf{h' = -\log(\Sigma p_i')}$, for **all $\mathbf{p_i}$ in $\mathbf{s_{k+1}}$.** Since this approach considers all remaining states bundled together ( $\Sigma$ inside the log, not outside), we simply want to *minimize* $\mathbf{\Sigma\, p_i'}$, which means <u>we greedily pick which move would allow us to eliminate the greatest total probability</u>, or, in other words, rule out a group of configurations states that together cover that greatest total probability.

## §2.5 Implementation: Theory-based Procedure

### §2.5.1 Motivation and Background

The first key assumption in our argument is that all *valid* configurations of Battleship boards, on a macro-scale, are equally likely—in other words, player **O** has placed their ships randomly. This, of course, does not account for human psychology and tendencies, but that is beyond the scope of this project. However, the crucial distinction is that all macro-configurations being equally likely does ***not*** imply that all squares are equally likely to hold a ship. For a simple example, consider placing just the carrier (length 5) on the board. Only two possible placements would result in (1,1) being part of the carrier [(1,1) → (1,5), or (1,1) → (5,1)], while a position like (5,5) could result from 10 possible placements (starting at (5,1) ... (5,5) going right, or (1, 5) ... (5, 5) going down).

Now, connecting back to the theory presented in the previous section: if we are looking for a "hit," since that is a lot less likely than a miss, we want to query a square that would be most likely to be a hit (i.e. included in the most number of configurations, as per the argument above). In other words, we are optimizing for the query that would eliminate the most configurations in the case of a miss—which is, therefore, the move that would leave the greatest number of configurations remaining in the case of a hit. While our specific procedure changes when tracking down a hit or immediately after a ship is sunk (Sections 2.5.3, 2.5.4), this general idea is crucial throughout the game and especially in cases where G is querying without any active hits to track down.

Thus, to create an algorithm for G's guesses in each round of the game, we would ideally want to conduct a complete search, enumerating all possible configurations of ships at any state in the game and then determining which square most often contains a battleship to query. However, this is computationally unfeasible: at the start of the game, for example, there are roughly $3 \times 10^{10}$ possible configurations. Therefore, our procedure involves *random sampling* of this larger state-space, conducted via an algorithm described in the next section.

### §2.5.2 Sampling Approach

Our sampling approach proceeds by simulating a possible board configuration by randomly placing one ship at a time, given constraints based on the state of the board and ships previously placed in the

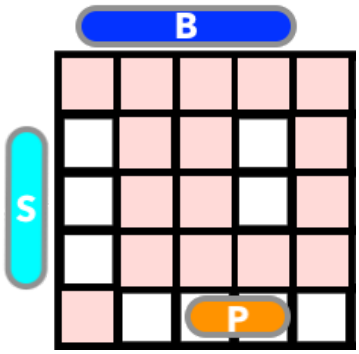
simulation. We place remaining (undiscovered) ships in increasing length order, which helps rule certain spaces out more efficiently: we can assert that if ship $i$ cannot be started (left end or top end) at location $(r,c)$, then neither can ship j for $j > i$ in our ordering, since `len(ship_j)` $\geq$ `len(ship_i)`.

Furthermore, the benefits of this one-at-a-time placement of ships to generate an individual configuration is that it allows us to avoid having to explicitly enumerate the full state-space, a computationally unfeasible task—in other words, we don't have to explicitly know the full population to still generate a strong sample. To very roughly quantify this, the runtime goes from $O(\prod n_i)$ to $O(\Sigma n_i)$, where $n_i$ is an approximate measure of the number of possible configurations of one ship. Note that this is by no means a rigorous or exact conclusion, as the various $n_i$'s affect each other differently in different cases.

While placing any given single ship, note that there are at most 200 possibilities we have to test as (1) the ship's orientation (horizontal or vertical) and (2) the location of a specified endpoint (say, W.L.O.G, the left or top endpoint, depending on orientation) together completely define a ship's placement. So, we can loosely upper-bound the number of possible placements to enumerate for one ship at $(2*10*10)=200$ in a $10x10$ board. Repeating this for up to 5 ships is much more manageable than $3*10^{10}$ operations.

Our sample is created by running $1000$ such simulations (the number can be changed based on preferences for query generation speed vs. sample size), and the square in the $10x10$ board that most often contains a ship across this sample is then queried. Initial runtime tests suggest that this process runs on the order of 1 second.

The only apparent flaw in this sampling method is that it is liable to run into an "impossibility" case while mid-simulation. (Note while this never occurs while simply trying to place five ships on the board by themselves, it can occur in later stages of the game, when we try to sample viable configurations of ships amid multiple constraints from misses scattered across the board.) For example, consider the section of the board below, assuming ships not shown have been placed in the rest of the board and there are no spaces for the depicted ships except the white spaces shown in this section of the board (red representing guesses that have resulted in a "Miss"). A human would quickly be able to evaluate that each ship fits exactly in the space of its length: B in the 4-length space, S in the 3-length space, and P in the 2-length space. However, the simulation might start by randomly placing P as depicted. Then, only one valid placement remains for S. But as it reaches B, the simulation would realize that no valid configurations remain, suggesting the impossibility of its earlier placements.




Red squares indicate misses (which ships cannot be placed on), while white squares are those that haven't been probed. If the sampling simulation starts by placing ship P as shown, then it will ultimately run into an impossibility when trying to place B (because P is occupying the only valid location for B.)

But encountering this rare sort of impossibility actually provides vital information: if B can't be placed in any remaining spots, it *must* occupy one or more of the squares where the simulation placed other ships. (In this case, that would be the squares where P was placed.) As such, our algorithm switches into an alternate procedure when it encounters such an impossibility, where it breaks out of its sampling procedure and instead randomly selects one of the squares allotted to an earlier ship during this simulated instance.

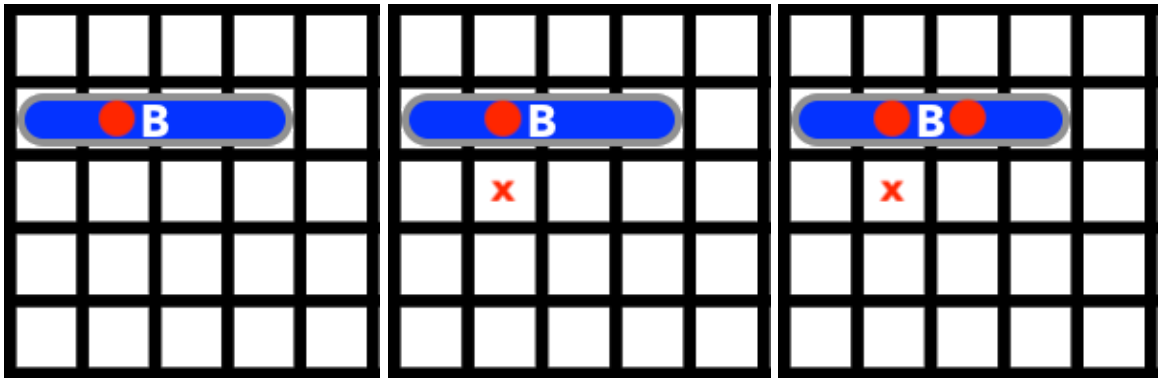
### §2.5.3 "Hit" Procedure

However, in the case of an **"active hit"** (when there is a hit on the board that does not correspond to a sunk ship), we toggle into a completely different procedure, one that more directly and explicitly focuses on chasing down the remainder of the unsunk ship. The general sampling method described in 2.5.2 cannot currently operate based on partial-hit data (i.e. in active hit situations), hence the necessity of this second procedural mode.

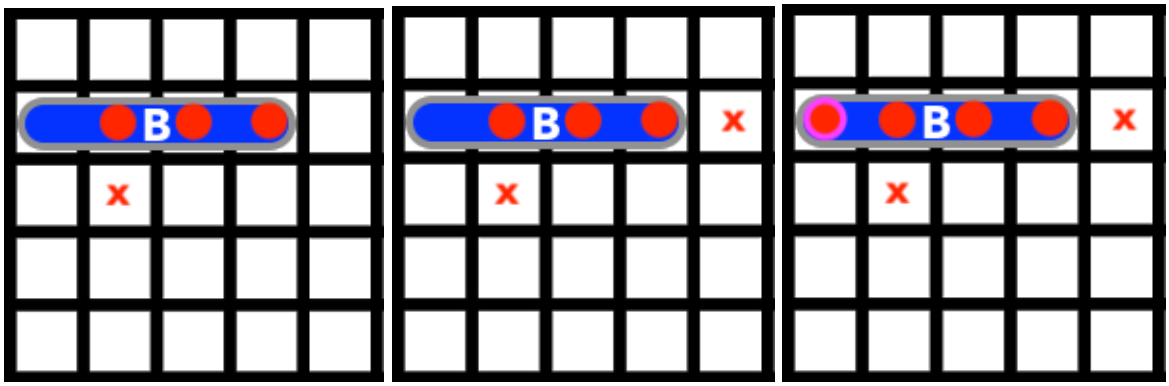
Once G gets a single hit—say at position $(r_H, c_H)$—we know that there must be a ship either along row $r_H$ (to the right of left of $c_H$ ) or along $c_H$ (up or down from $r_H$). In order to gauge which direction to begin chasing down, our algorithm begins by running a mini-simulation, enumerating all valid placements of each remaining/undiscovered ships such that it overlaps with the $(r_H, c_H)$ but does not run off the board or run into any other obstacle (like a previously-guessed square that resulted in a "Miss"). Then, whichever square immediately adjacent to the hit—$(r_H+1, c_H)$, $(r_H-1, c_H)$, $(r_H, c_H+1)$, or $(r_H, c_H-1)$—is included in the greatest number of the generated placements is the square that is queried by the algorithm.

In the case that the queried square is a **Miss**, this procedure is repeated with the remaining adjacent squares until the second hit is found. Now, the algorithm has effectively found the "line" of the ship and can go track it down, continuing in the same direction until reaching either a **"Hit and Sunk"** result or a **"Miss."** If a Miss is received, the algorithm goes back to the original hit square and continues in the opposite direction from the line it was pursuing, as this case suggests that its original hit began in the

middle of the ship and it now needs to continue in the opposite direction. One example of this entire hit-pursuit procedure is visualized in the diagram below.




**1:** Initial hit triggers "active hit" **2:** Adjacent guess → Miss **3:** Re-evaluate, next adj. guess → Hit

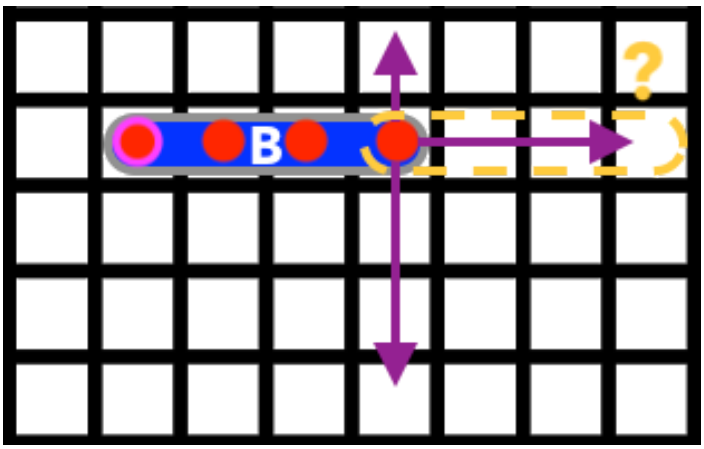



**4:** Once line found, continue chasing **5:** Next guess: Miss, so retrace… **6:** Move in opposite direction→ sink

## §2.5.4 Post-Sink Procedure

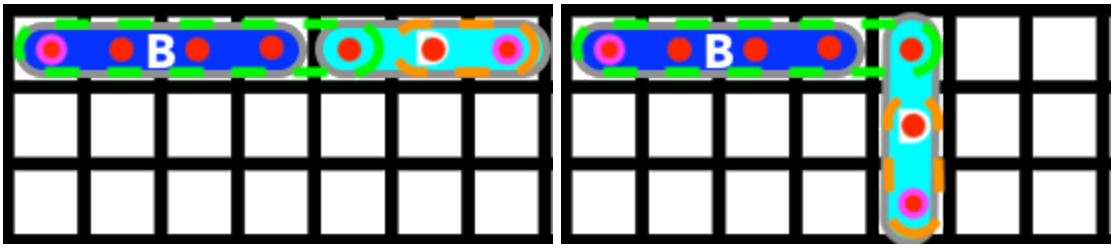
One crucial difficulty in this problem is that when a ship is sunk, **G** only receives the message "Hit and Sunk" rather than also being given the coordinates of the sunk ship: in other words, **G** does not necessarily know which ship was sunk. Broadly, in qualitative information-theoretic terms, this reduction in information must be countered with additional guesses to be gained back. More specifically, **G** must ask additional questions in order to try to constrain the specific ship identity better.

Roughly speaking, our procedure thus traces down lines in a "+" pattern from the end of a line of hits *opposite* to the "sink" end (the hit that resulted in a **"Hit and Sunk"** response), as shown below. This helps determine whether there are other ships along the same line, or perhaps perpendicularly adjacent to the ship that we have actually sunk. To reiterate, this is important because hits that may appear to be part of the sunk ship might actually be from other, partially-discovered ships.




As shown, the rightmost hit could belong to an undiscovered ship,
an uncertainty that motivates the "+" pattern check depicted.
Leftmost red circle with pink outline indicates hit that resulted in "Hit and Sunk".

Even with these probes, it is *impossible* in some cases to completely determine the identity of a ship, with some examples presented in the diagrams that follow. As such, our strategy is to make an assumption about the ship length that is found—which we need in order to place specific ships at specific positions to run our sampling simulation—while also storing data about all other possible configurations of ship identities that may have resulted in the observed hit pattern. Then, if presented later in the game with evidence that suggests our assumption was impossible (e.g. being unable to even place the first ship during the sampling simulation), we can switch into a stored alternate possibility that would be valid given the current game state.




True configuration could be **4** + **3** but could also be interpreted as **5** + **2**

Once this "post-sink" procedure is wrapped up, we loop back into our standard sampling method of guessing squares until another active hit is triggered, repeating this whole process until **5 "Hit and Sink"** messages are received and the game ends.

# §3. Hangman

## §3.1 Game Objective & Rules

In Hangman, there are two players — a “Solver” and a “Host.” The Host chooses a phrase consisting of common English words, initially providing the Solver with the number of words and the length of each word.

The Solver then seeks to guess the phrase in the fewest number of queries; a query takes the form of guessing a single letter, with the Host either filling in every position where that letter appears in the phrase or declaring that the letter does not appear.

In this version of Hangman, the Host is allowed to lie at most once, falsely declaring that a letter does not appear when it actually does. However, the lie must be consistent throughout the whole clue; for instance, if the phrase is “APPLE PIE” and the computer guesses the letter “E”, the user must either choose to lie throughout the phrase and claim no E’s exist at all or provide a complete listing of all Es in the phrase. Essentially, the user cannot enter “****E  ***” as their lie.

The game ends when the Solver guesses the phrase correctly. Scoring is contingent on the total number of questions asked — this is the quantity that we seek to minimize.

## §3.2 Information-Theoretic Framework

### §3.2.1 Background & Motivation

We frame the Hangman problem using the information-theoretic model presented in the Introduction section using correspondences presented in 3.2.2. Note that the Host **H** in this specific game would be Ms. Aiyer, while the Solver **S** is represented by our program/algorithm. The first step to thinking in terms of information-theoretic systems is to specify the overall information flow. For the purposes of this formalization, we divide each query—asking the Host to fill in the specified letter in the phrase—into two parts.

The first part is a simple Yes/No question about whether the letter exists in the phrase. If this answer is yes, the second part of the query—the actual positions of the letter in the phrase—becomes relevant. However, the winning conditions and gameplay can be thought of as contingent only on the first part. In other words, a “win” for the Solver can be seen as ascertaining the set of letters that exist in the Host’s phrase because the Host must divulge the specific positions once a letter has been deemed to exist—a determination that could be based solely on these binary responses. The second part of the query exists to speed up this guessing process by indicating which letter to ask about next. Thus, we model the game as two separate information flows that are, in practice, conducted simultaneously.

### §3.2.2 First Information Flow

The first information flow corresponds to the initial binary questions that determine whether the queried letter exists in the phrase:

1. **Source**: The sets of letters that exist and don't exist within the original phrase. Essentially just a random variable X composed of 26 Bernoulli RVs (either 0 for missing or 1 for existent) for each letter.
2. **Encoder:** The queries from the Solver, specifically the part that asks if a particular letter exists in a phrase. The specific letter that the Solver asks about is tied into the estimated probability distribution for random variable X and therefore its entropy. Each query ascertains a single bit of information.
    a. Although the queries also ask about the specific positions of the letter in each phrase, that is irrelevant to our primary game objective of ascertaining whether or not a letter appears in the phrase. The secondary goal of minimizing the number of queries will be affected by the specific positions of the letters and is addressed in the second information flow.
    b. **Channel Encoder:** Furthermore, to account for the noise in the channel (as described below), redundant questions may be asked — this depends on the work of the channel decoder and, as always, the second information flow.
3. **Channel:** The Host's answers to the binary query of whether the letter exists. This channel could contain noise, represented by the potential lies specified in Section 3.1.
    a. **Noise:** The channel will, over the course of a single game, make at most one mistake. Furthermore, a 0-bit (representing that the queried letter does not exist) will never be flipped to a 1-bit. However, a 1-bit could potentially be flipped to a 0-bit.
4. **Decoder:** The algorithm that takes back the binary answers and uses those to eliminate certain words, bringing the Solver closer to ascertaining the full set of existing letters in the Host's phrase.
    a. **Channel Decoder:** the number of possible words remaining. Accounts for the lie by maintaining several potential word lists.
5. **Receiver:** The Solver's account of which letters exist in the phrase and which do not. The game ends once all letters that exist in the actual phrase are also contained within the Solver's list of existing letters.

### §3.2.3 Second Information Flow

The second information flow, representing the specific positions of the letters in each phrase, becomes relevant only when the first part of the query to determine existence returns a 1-bit (i.e., the letter exists). This information can be thought of as not central to the game, but rather secondary information that is used to refine the Solver's estimated probability distribution for the random variable X (defined above) and better inform the next query. In practice, once the Solver knows that a certain letter exists in a phrase, the specific positions are "inevitable" as the Host must divulge them. Furthermore, this second channel contains no noise, differentiating it from the first.

1. **Source:** The actual phrase itself, which is a list of positions with a specific letter in each one.
2. **Encoder:** The program query asking for the specific positions of the letters in the phrase, triggered by the first part of the question determining that the letter does indeed exist within the phrase.

   a. The specific letter that is queried is based on the probability distribution of random variable X, which is re-computed with each iteration of this secondary information flow.
3. **Channel:** The Host's responses to these secondary queries. This is a theoretically-noiseless channel, as the Host will never lie about the position of a letter. The lies are contained within the first information flow, as this flow is only triggered if the first answer determined that the letter exists (which can never be a lie, as specified above).
4. **Decoder:** The algorithm that creates the relevant possible word lists from this information, eliminating words that don't fit the pattern, to create the ultimate probability distribution to better inform the guesses.
5. **Receiver:** The resulting computed probability distribution for random variable X that informs the next query.

Again, it is important to note that although we separate these two interconnected information flows for a theoretical framework, in practice they are concurrent and part of the same query.

## §3.3 Theory: Framework & Formalized Argument

### §3.3.1 Introduction

To minimize the total number of guesses, it is sufficient to minimize the number of incorrect guesses as queries asking about existing letters will always be required. Thus, it follows that with each query, we seek to ask about the letter that has the highest likelihood of appearing in the phrase with the information we have at the time, as this minimizes the probability of an incorrect guess.

The game of Hangman is limited to valid words in the English dictionary, and phrases are simply lists of valid English words. First, we approach the high-level problem of ascertaining which letter to guess after the individual probability distributions of each word are determined. In order to maximize the information we gain from each guess, we must seek to guess the letter that has the highest probability of appearing at least once in the phrase.

### §3.3.2 Combining word-probabilities to form phrases

Given a phrase with $\mathbf{n}$ words, $\mathbf{n}$ probability vectors $\mathbf{w} = \langle \mathbf{w_1}, \mathbf{w_2}, \ldots, \mathbf{w_{26}} \rangle$ exist, with each vector representing the likelihood that each letter ($A \rightarrow p_1$, $B \rightarrow p_2$, ... $Z \rightarrow p_{26}$) appears at least once in the corresponding word. To obtain the cumulative probability vector $\mathbf{q} = \langle \mathbf{q_1}, \mathbf{q_2}, \ldots \mathbf{q_{26}} \rangle$, we define

$$\mathbf{q_k} := 1 - \prod (1 - \mathbf{w_k}) \qquad \mathbf{k = 1, 2, \ldots 26}$$

over all **w-vectors** in the set. The maximum value within $\mathbf{q}$ is then found and the corresponding letter is guessed; as the probability of not guessing wrong is maximized, the complementary probability of guessing wrong is minimized, thus minimizing the total number of guesses.

### §3.3.3 Determining probability distributions for each word

Before the high-level task of combining the probability vectors for each word can occur, we must actually determine the expected probability vector $\mathbf{w}$ for each individual word. To do so, we must take into account the noise in the first channel. We keep track of every potential lied-about letter, defined as any letter for which Part 1 of our query (as defined in section 3.2) ascertained that it does not exist within the phrase, but queried it exactly one time. This is because of the restrictions on the first channel's noise, limited to at most one lie and only shifting a yes-response to a no-response.

If $\mathbf{L}$ such potentially lied-about letters exist, there are $\mathbf{L+1}$ possible "true" probability vectors — exactly one of the $\mathbf{L}$ letters may actually appear in the word, or none of them could (i.e. there were actually no lies). Thus, the probability vector $\mathbf{w}$ of the word is defined as $\mathbf{E[S]}$, where $\mathbf{S}$ is a random variable that could equal any one of the $\mathbf{L+1}$ possible true probability vectors. Therefore,

$$\mathbf{w := E[S] = s_1 p(s_1) + s_2 p(s_2) + \ldots + s_{L+1} p(s_{L+1}).}$$

To determine the values contained in each of the 26-dimensional $\mathbf{s_1, s_2, \ldots s_{L+1}}$ vectors, the dictionary is first used to create $\mathbf{L+1}$ potential word lists, with each list eliminating all words that do not match its assumptions about the word being guessed. There are three criteria used for this elimination:

1. The set of letters that could potentially have been lied about but are assumed not to exist in the word for that specific $\mathbf{s}$-vector
2. The letters that are known not to exist in the word, either through redundant guesses or because the initial query determined that they do exist in the phrase, just not in the specific word (this assumption is common across all $\mathbf{s}$-vectors for a word)
3. the known positions of letters that do exist in the word (this is also common across all s-vectors in a word)

For each word list, the probability that a given letter exists in a randomly-chosen word from that list is computed for each of the 26 letters, yielding the $\mathbf{s}$-vector.

The associated probabilities $\mathbf{p(s)}$ for each $\mathbf{s}$-vector for the $\mathbf{E[S]}$ calculation are determined by the magnitude of the vector; we assume that the more words that are possible under that assumption, the more likely it is to be true. Formally,

$$\mathbf{p(s_k) = |s_k| \,/\, \sum |s|.}$$

Combining these two definitions yields the $\mathbf{w}$-vectors for each word, which are then combined into the overall $\mathbf{q}$-vector used to determine the next optimal query.

## §3.4 Practical Coded Implementation

### §3.4.1 Logistical Considerations and Structural Overview

The code was written entirely in Python 3.8, with the English word dictionary used being pulled from https://raw.githubusercontent.com/raypereda/hangman/master/words.txt. The code consists of a

number of helper methods to create word lists and probability distributions, a Word class that enumerates useful methods and member variables, and the main play_game() function that contains the actual game loop.

### §3.4.2 Game Initialization

The game begins by asking the Host to input a comma-separated list of $\mathbf{n}$ word lengths for their phrase with $\mathbf{n}$ words. Each word must be a valid dictionary word (see Section 3.4.1) for the code to work optimally. Based on these values, a list of $\mathbf{n}$ Word objects (see Section 3.4.1) is created, with each Word being initialized with a potential word list based solely on its length.

Furthermore, an empty list is created to keep track of letters that have been queried exactly once and could represent potential lies (Section 3.3) as well as a simple counter to keep track of the number of wrong guesses.

### §3.4.3 Game Loop: Determining Distributions, Formulating the Query, and Retrieving Host Responses

Following the initialization process, the game loop begins and continues until all words in the phrase are fully known, determined by a flag updated in the body of the loop that retrieves each word's status. Each loop begins by printing the known letters in the phrase so far, using asterisk (*) characters for any unknown letters. With each iteration, the probability distributions for each of the $\mathbf{n}$ words ($\mathbf{w_1}$, $\mathbf{w_2}$, … $\mathbf{w_n}$) are retrieved using the relevant method in the Word class, which calls on several of the helper methods mentioned earlier and uses the procedure outlined in Section 3.3.3. These probability distributions are combined as they are retrieved to create the cumulative probability distribution $\mathbf{q}$ using the procedure outlined in Section 3.3.2. Note that this procedure uses information determined both from the primary and secondary flow of information from previous loop iterations.

After these computationally-intensive steps of the game loop end, the maximum value contained in $\mathbf{q}$ and the letter that it corresponds to (based on its position) are determined, and that letter is queried. Word-by-word, the Host is asked to input the specified word, using asterisks for irrelevant letters and the actual queried letter wherever it appears. This process is repeated for all Words in the phrase. Note that this information retrieval procedure combines the two flows of information detailed in Section 3.2; this is simply because it eases the Host's user experience. As the Host enters this information, the known positions for each Word object are updated using the relevant method in the Word class. Furthermore, on the last iteration, this is where the loop termination flag is set if all letters in each word have been determined.

### §3.4.4 Game Loop: Branching to Account for Channel Noise (Lies)

Once the Host enters information about all words in the phrase, the code updates certain fields to account for the channel noise — i.e., that the Host is lying — using the information conveyed through the primary information channel, defined in Section 3.2 as the one-bit response to whether the letter exists in the phrase (Bit = 1) or not (Bit = 0).

If the letter is said to not exist in the phrase at all (Bit = 0), then the number of wrong guesses is incremented. Furthermore, if the letter already existed in the list of potentially-lied-about letters, it is removed from that list as the Host can lie only once over the course a game. If the letter did not previously exist in this list, it is added (assuming no lie has been detected yet).

If the letter does exist in the phrase and it previously existed in the list of potentially-lied-about letters, that indicates that the code has discovered a lie. The number of wrong guesses is decremented, and the list of potentially-lied about letters is now irrelevant. Furthermore, a class method is called for each Word object setting a flag that ultimately modifies the way in which the probability distributions $\mathbf{w}$ for each word are calculated, now only taking into account a single $\mathbf{s}$-vector (see Section 3.3 for details).

After this set of conditionals, the game loop returns to the top, continuing if any letters remain unguessed in any of the words.

### §3.4.5 Robustness: Deviating from the Dictionary

Although the rules of the game specifically prevent any non-dictionary words from being used within the phrase, the code remains robust enough to handle such a complication. In this case, Word objects whose potential word list(s) collectively do not contain any dictionary words matching the known information return a probability distribution based on the initial distribution of all dictionary words with the specified length, ignoring the known letter positions. This distribution accounts for letters known to already have been queried, setting those to probability values to zero. Although certainly not nearly as optimal as the original code when used according to the specified rules, this provision provides a failsafe in case the dictionary fails.