Susbtring Search with a Lie

Sparsho De

October 2019

1 Abstract

In Ulam's liar game, Paul searches for a element e hidden by Carole within the set $S = \{1, 2, 3, 4, ..., n-1, n\}$ by asking yes-no questions about the location of the element within subsets of S. Carole is allowed to lie to exactly one of Paul's questions. Paul wins in k moves once he knows the value of the hidden element. We extend the results of traditional Remyi-Ulam Liar Game to a new variation on searching for binary substrings within the analagous binary equivalent of S. Questions are only allowed to ask if the hidden element contains a specific substring. First we considered the game without a lie. We were able to implement a greedy algorithm and additionally determined an log time algorithm that was almost as efficient. With the addition of a lie, we were able to implement an algorithm that worked when $n <= 2^{k/2-2}$. This was not as efficient as q-ary Yes/No questions nor Prefix questions. This was due to the nature of substring search as opposed to a bad algorithm.

2 Introduction

2.1 Research Problem

A j-batch Ulam's liar game is played as follows: A responder determines some value, e, that is an element of the set $\{1, 2, 3, 4, ..., n-1, n\} = S$. The questioner tries to determine the value picked by guessing certain subsets of S. The responder is allowed to lie to the questioner a fixed number of times, suppose k. The questioner is allowed to ask a fixed number of questions, j at a time. In a non-adaptive game, $j = 1$.

The goal is to determine an efficient algorithm to guess the value of e using generalized prefix questions allowing for one lie. Prefix questions are of the form 'does the b-ary representation of the element begin with (an arbitrary sequence).' Generalized prefix questions extend the search for binary prefixes across the array of characters present within string. In short, a generalized prefix is a substring. Thus, we are extending the search for binary characters present from the prefix to the entire string array. This change suggests a novel searching method different from the graph search used for prefix questions is needed to solve the game. The success of an algorithm will be measured by the number of moves necesssary for the questioner to win the game, and the algorithm will be considered a complete failure if the responder wins.

2.2 Background and Terminology

Ulam's liar game, and strategies to solve it, have lately been of great interest to mathematicians and computer scientists alike. In particular, Andrzej Pelc published the perennial paper on the topic, and is the principle leader of the subject. Most research conducted on the game involves partitioning the search space M. These questions are known as q-ary questions. They ask if the value of e will be found in any of the partitions. It is important to consider the implications of this original sect of research before advancing to prefix questions.

For further simplification, the questioner will henceforth be referred to as Paul, and the responder will be called Carol. It is clear that Carol does not actually have to choose an element e. Instead, she can employ what is colloquially known as the Devil's Strategy. Instead of choosing an element within the search set before hand, Carol simply has to respond to Paul's questions in a consistent enough manner such that an element remains which satisfies all conditions set for the game (satisfies all but one of the questions asked). Elements of M that satisfy all the questions are part of the truth set, and those that satisfy all but one are part of the lie set. We formally define the weight function below.

Definition: $w_j(a, b) = a(j + 1) + b$, where a is the truth set and b is the lie set (Berlekamp, 1968).

Extension: $w_{j+1}(a, b) = w_j(a_1, b_1) + w_j(a_2, b_2)$, where a is the truth set and b is the lie set (Pelc, 1987).

Standard liar games have Paul split the search arbitrarily in two, and Carole choose which of the subsets she wants to assign the lie to. Carol will assign the lies to one of the subsets in such a manner that the weight of the game is maximized. Paul's goal is to minimize the weight of the game. Each question that Paul asks requites 2 answers. Thus, is goal is to minimize the maximum possible weight that Carol will choose.

Armed with this knowledge, we can venture forth into Prefix Questions. Pelc examined Prefix Questions more like discrete structures than algebraic objects. He analogized the question to searching through a binary tree with errors. He further analogized truth and lie sets to their discrete counterparts and proceeded from there. THe main results are as follows:

$$\lim_{x\to\infty} X(n) - U(n) = \infty$$

and

$$\lim_{x\to\infty} \frac{X(n)}{U(n)} = 1.$$

$X(n)$ refers to the number of questions to determine e using base 2 prefix questions, while $U(n)$ refers to the same for arbitrary yes-no questions (Pelc, 1988).

In particular, the difference between number of questions to determine e using prefix questions and arbitrary yes-no questions diverges to ∞, but their ratio converges to 1.

2.3 Purpose

Most research involving liar games deals with arbitrary-yes no questions. In fact, the game using arbitrary yes-no questions with one error has already been solved (Pelc, 1987). No research so far has attempted to generalize prefix questions. Thus, the inherent facet of novelty in this research lies within its generality. Delving into theoretical implications of such a change is worthy of consideration and will be completely new. In particular, applications are as far-reaching as the construction of new error correcting codes.

2.4 Independent Variable

For the sake of the reader, we may define the 'independent variable' in the proposed study as the type of question asked. As stated earlier, prefix questions will be generalized to substring questions. The relative rates of performance to standard liar games and prefix questions could be of interest. Additionally, the optimal strategy for the responder would change for each question asked.

2.5 Dependent Variable

The 'dependent variable' in the proposed study would be the expected number of moves needed for Paul to win. This would be a measure of the algorithm's efficiency.

2.6 Experimental Hypothesis

It is hypothesized that the type of question asked will not impact the expected number of moves needed for Paul to win. This is because the tactics for each question should be somewhat similar, even if they are not identical. However, the complexity of the algorithm to determine the best move may be different for each type of question.

3 Results

3.1 Part 1: No Lie

Due to the inherent complexity of a substring game, we must first consider the game with no lie. We find that the game is already computationally expensive, even without the addition of a lie. Due to the nature of the game, we settle for an $\mathcal{O}(\log n)$ approximation algorithm that succeeds in determining a substring, but is not necessarily optimal. We have already defined a devil's strategy for Carole, the responder, but we will repeat it. Since Carole's goal is to extend the length of the game, her answers should leave the most options available for the next round of the game. For example, if replying YES to Paul's questions leaves a_1 substrings , but replying NO leaves a_2 remaining substrings, her answer would be MAX(a_1, a_2), and the corresponding response. As a result, Paul's choice of substring must minimize MAX(a_1, a_2). For the sake of Part 2 (With a Lie), we shall call such a move min(MAX(a_1, a_2)).

For now, We will assume that a greedy algorithm is the best algorithm. This is the case for small lengths, and implementing a brute force search would involve far too much run time (though intuition suggests that the greedy algorithm truly is optimal). Thus, before delving into the different strategies, a discussion on Substring Densities are necessary. Even determining substring densities is computationally expensive. A brute force algorithm to determing the density of a particular substring is $\mathcal{O}(2^n)$. For all substrings, it would be $\mathcal{O}(2^n \cdot$ (number of substrings of strings length n)). Determining the number of substrings in a search space of length k is fairly straightforward. It is sum of all strings of length 1 to length k. This can be written as $2^1 + 2^2 + 2^3 + ... + 2^k = 2^{k+1} - 2$. Thus the total time needed to determine all substring densities is $\mathcal{O}(2^{2k+1} - 2^{k+1})$, which is still EXP.

The following is an implementation of the algorithm described above:

```
import itertools
def binaryseq(k):
  return ["".join(seq) for seq in itertools.product("01", repeat=k)]

def substringcounter(n,m):
  count = 0
  for i in binaryseq(m):
    if i.find(n) >= 0:
      count = count + 1
  return count

def printout(l):
  for j in range (1,l+1):
    for p in binaryseq(j):
      print(p),
      print(':'),
      print(substringcounter(p,l))

print("Substring Densities for {0,1}^3")
printout(3)
```

The output of the example code is the following table:

String	Count
0	7
1	7
00	3
01	4
10	4
11	3
000	1
001	1
010	1
011	1
100	1
101	1
110	1
111	1

Figure 1

One could marginally cut down the time by automatically inserting a 1 for count whenever the string length and substring length are the same. Additionally, one could automatically insert $2^k - 1$ as the count for Strings 0 and 1. Furthermore, we can set up the idea of a reverse string. This idea comes into fruition in Part 2. A reverse string can be formed by taking every bit of a string and flipping it. More formally, $RS = 11111....1 - S$, where S is the string that will be reversed, and RS is the reversed string. S and RS have the same count, so we would only ever half to do half of the work.

A faster way to determine substring density is by employing recursion. We can determine any substring count by using a recurrence relation. For example, consider the substring 00. The number of 00s in a string can be modeled by the following :

There are three subcases. We consider only the first 2 strings. If the first two digits are 00, then the remaining strings each have 2 choices, a 0 or a 1. Thus, we have 2^{n-2} total substrings. If the first two strings are 01, then we have $a_n - 2$ total strings because The first the remaining string has $n - 2$ bits. Lastly, if the first bit is a 1, then we have a_{n-1} choices, since there are $n-1$ bits left. Therefore, the recurrence is: $a_n = a_{n-1} + a_{n-2} + 2^{n-2}$. While recurrences are computational cheaper than brute force algorithms, a closed form would be even faster. Using computational power, we determine that the closed form for such a recurrence with initial values $a_0 = 0, a_1 = 0, a_2 = 1$ is:

$$a_n = 2^n - \frac{3 \cdot F_n}{2} - \frac{L_n}{2},$$

where F_n is the nth Fibbonacci number and L_n is the nth Lucas number.

$$F_n = \frac{1}{\sqrt{5}} \cdot (\frac{\sqrt{5}+1}{2}) - \frac{1}{\sqrt{5}} \cdot (\frac{-\sqrt{5}+1}{2})$$

and

$L_n = [\phi]$, where [] indicates the nearest integer function and

$$\phi \approx 1.618033988749895\ldots$$

The tactic used to determine the count of substring 00 for strings of length k can be easily generalized to all strings of the form $\{0\}^k$ (strings of length k consisting only of 0s).

$$a_k = \sum_{i=1}^{k} a_{n-i} + 2^{n-k}$$

The summation notation is hard to stomach on its own, so we take steps to simplify it. Due to the telescoping nature of the sum, we consider the difference $a_{k+1} - a_k$.

$$a_{n+1} - a_n = \sum_{i=1}^{k+1} a_{n+1-i} + 2^{n+1-k} - \sum_{i=1}^{k} a_{n-i} - 2^{n-k}$$

. Note that the summations telescope and the only remaining term is a_{n-k}. Thus,

$$a_{n+1} - a_n = 2^{n-k} + a_n - a_{n-k}$$

.

In particular, $a_{n+1} = 2a_n - a_{n-k} + 2^{n-k}$. Let $a_n = 2^n - b_n$. Then, we have:

$$2^{n+1} - b_{n+1} = 2(2^n - b_n) - 2^{n-k} + b_{n-k} + 2^{n-k} \implies b_{n+1} = 2b_n - b_{n-k}.$$

.

Now consider the characteristic polynomial associated with the recurrence:

$(\lambda)^{n+k+1} - 2(\lambda)^{n+k} + 1 = 0.$

Since the characteristic polynomial has degree greater than 4, by Abel-Ruffini, we cannot have a general closed form for all $\{0\}^k$, but we have a faster way to count. Although not pertinent to the problem at hand, it is interesting to note that b_n is the k-bonacci sequence.

The ultimate question to answer is the number of questions needed to determine the chosen string asking only about present substrings. We emply a greedy algorithm, which, for now, we assume to be optimal. At each round, the question asked will be the min(MAX(a_1, a_2)). Formally, each question will attempt to minimize $|\frac{1}{2} - \frac{a}{2^n}|$, where a is the number of remaining strings that either satisfy or do not satisfy the substring (i.e. whichever is smallest). Then, when the move is complete, the substring counts are reset, and the process starts again. The code is as follows:

```
import itertools;
def binaryseq(k):
  return ["".join(seq) for seq in itertools.product("01", repeat=k)]
def binarysubset(k):
  resultlist = []
  for i in range (1, k+1):
    for j in binaryseq(i):
      resultlist.append(j)
  return resultlist

// creates list of binary strings of a particular length

def function(alist):
  blist = []
  for i in alist:
    blist.append(len(i))
  maxlength = max(blist)
  resultlist = []
  for i in binarysubset(maxlength):
    count = 0
    for j in alist:
      if j.find(i) >= 0:
        count = count + 1
    resultlist.append(count)
  return resultlist

//counts substrings

def playermove(alist):
  list1 = function(alist)
  j= len(alist)
  checkmovelist = [abs(1/2 - i/j) for i in list1]
  s = checkmovelist.index(min(checkmovelist))
  return binarysubset(12)[s]

// makes the move
```

```
def updatestringlist(alist, string1):
  blist = alist.copy()
  for i in alist:
    if i.find(string1) >= 0:
      alist.remove(i)
  if len(alist) >= 0.5 * len(blist):
    return alist
  else:
    return [x for x in alist if x not in blist]

//updates the substring count in the remaining list of strings

def playtime(x):
  y = binaryseq(x)
  count = 0
  while len(y) > 0:
    y = updatestringlist(y, playermove(y))
    count = count + 1
  return count

// repeats the above functions until the list is of length 0

print(playtime(3))
```

The output is the following table:

Length	Number of Questions
1	1
2	3
3	5
4	-
5	8
6	-
7	12
8	15
9	17
10	17
11	16
12	21

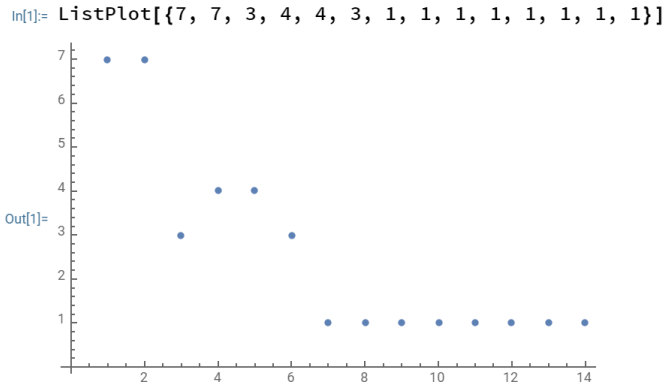
Figure 2

The output returns the number of questions needed. However, the algorithm is extremely computationally expensive. In an effort to decrease the time spent, we can instead search substrings of a particular length instead of all of them.

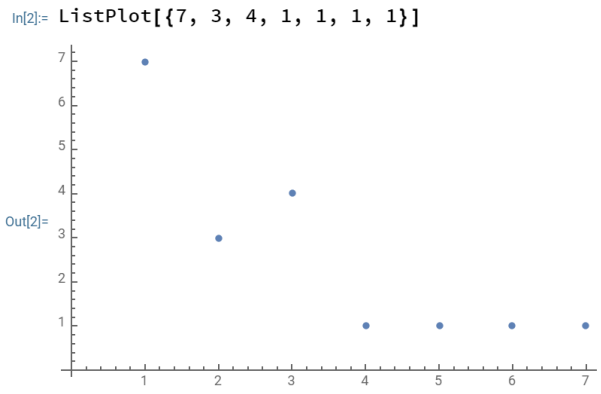
In particular, we will still try attempt to minimize $|\frac{1}{2} - \frac{a}{2^n}|$, where a is $\min(\text{MAX}(a_1, a_2))$. However, we can speed up the search for the optimal substring by searching through strings of a particular length, instead of all substrings. We do so by tracking the count of the $\{0\}^k$ substring.

Claim: We can determine the correct number of digits for the optimal substring in linear time. However, this cannot be extended to all rounds, just the first.

Proof: First, consider a graph of the distribution of each substring in $\{0,1\}^k$. We will use the data from Figure 1.



If we remove repetive inputs (i.e. removing 111 if 000 exists), then we see why tracking the optimal substring by digit is even viable.



A clear 'sharktooth' pattern emerges. In particular, we note that $\{0\}^k$ is minimally occurring and that the least common substring of length k occurs

more than the most common substring of length $k+1$. This second statement is trivial to prove. Simply note that every substring of length $k+1$ has a substring of length k in it, but every substring of k does not necessarily have a substring of length $k+1$ in it. The first statement is also fairly intuitive (for which we will not provide a formal proof). Consider set $S = \{0,1\}^k$. Now we remove all elements of S which have $\{0\}^c$ as a substring and place them in a different set Q. Suppose there are r values of $\{0\}^c$ in Q. We can replace all r values of $\{0\}^c$ with some other c digit substring. However, a single element of Q is likely to have multiple $\{0\}^c$ substrings, leading to more than 1 additional ocurrence of the other c digit substring. In short, an element containing $\{0\}^c$ as a substring is more likely to contain multiple $\{0\}^c$ substrings within it as opposed to any other c digit substring. This is once again fairly intuitive. If we wanted to repeat 000, we simply need to append an additional 0. For any other three-digit substring, we would need to completely repeat it.

Once we have completely convinced ourselves of these facts, we simply need to show that if the optimal substring has c digits, then the most optimal substring of the $\{0\}^k$ also has c digits. We define a function $f(k) :=$ count of $\{0\}^k$ for strings of a fixed length. Showing that $|\frac{1}{2} - \frac{f(k)}{2^j}|$ is a convex function in its entirety would mean that a computer would immediately know the number of digits once $|\frac{1}{2} - \frac{f(k)}{2^j}|$ increases, meaning that the algorithm would determine the number of digits in $\mathcal{O}(n)$ time. Note, this is true if $f(k)$ is monotonically decreasing.

The function $f(k)$ had already been defined recursively where $a_{n+1} = 2a_n - a_{n-k} + 2^{n-k}$. Letting $k_1 = p$ and $k_2 = p+1$, we have $a_{n+1} = 2a_n - a_{n-p} + 2^{n-p}$ and $a_{n+1} = 2a_n - a_{n-p-1} + 2^{n-p-1}$. After subtracting, rearranging, and simplifying, we have $a_{n-p} = a_{n-(p+1)} + 2^{n-(p+1)}$. In particular, appending an additional 0 to $\{0\}^k$ will cause $2^{(n-(p+1)}$ less subsets (making $f(k)$ monotonic). This proves our original claim. Implementation will not be shown.

The nature of substring search indicates that there is no efficient algorithm that can determine the optimal move in log time. However, If we graph the output, we note it is similar to that of a logarithmic graph, indicating that an efficient log time (non-optimal) algorithm exists.

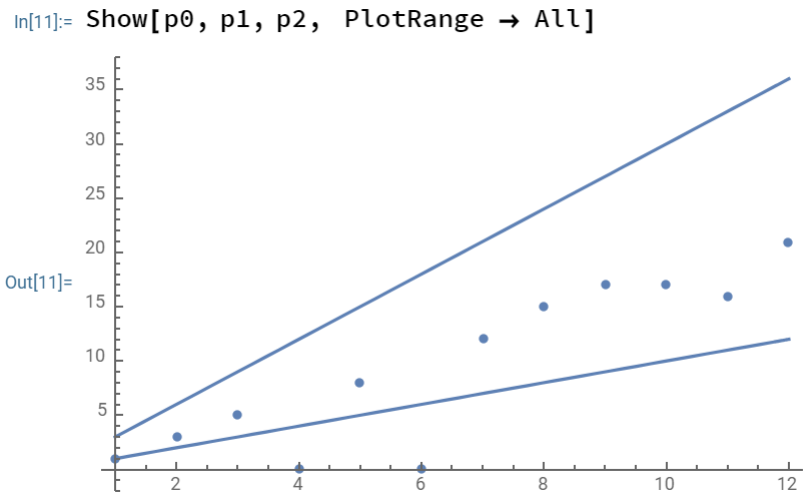
The approximation algorithm is delineated as follows:

Assume, by question i, we know that the missing string has subtring s of length i. We simply ask if the string contains the substrings $s^\frown 0$ and $s^\frown 1$. If it contains one, then we have simply reached looped. Else, we know that the substring was actually a suffix, and we can simply ask for $0^\frown s$ OR $1^\frown 0$ (note that we do not need to ask both since we have confirmed that the substring is not a suffix). In a worst case scenario, we would ask $2k$ questions, where k is the number of digits in the search set. This is a log time algorithm. For the reader's benefit, we have provided an example game using this algorithm:

1. Does s contain the substring 1? Yes.
2. Does s contain the substring 10? No.
3. Does s contain the substring 11? No.
4. Does s contain the substring 01? No.

Figure 3. Sample game with n = 2, k = 4.

When compared to the optimal algorithm and prefix search, we have the following figure:



Fairly straightforward reasoning suggests that the number of moves an optimal algorithm (referred to as $g(k)$) takes can be bounded by x and $2x$, where x is the number of moves. The approximation algorithm can simply be regarded as an unoptimal variation of the greedy algorithm. Prefix questions can be regarded as a more precise version of the greedy algorithm. Since location of the substring is known, each question would cut the search set in exactly half (the goal of the greedy algorithm). A formal bijection would complete the proof.

To avoid the complexities of *optimal* substring search, we will use the discussed approximation algorithm in Part 2 (with a lie).

3.2 Part 2: With a Lie

The addition of a lie makes the game far more complex. The graph of the greedy algorithm further convinces us that it is not optimal. Thus, any proposed methods to win a substring search game with a lie are guaranteed to be unoptimal. However, the goal is still to find the fastest algorithm possible.

First, for the benefit of the reader, a full game will be played for the case $k = 3$, where k is the number of digits in the binary form. The search set

is $\{000, 001, 010, 011, 100, 101, 110, 111\}$. Using notation provided by Spencer (1992), we define the weight of a state (in a 1-lie game) as $a(j+1)+b$, where j is the number of questions remaining, a is the size of the truth set and b is the size of the lie set. (As previously mentioned, the truth set is the set of numbers that satisfy all questions, and the lie set is the set of numbers that satisfy all but one). The following is how such a game would proceed:

1. Does s contain the substring 01? Yes.

With a bit of casework, we note that the substring 01 appears 4 times in the search set, reducing it exactly in half. Now, regardless of the answer, the weight of the state is minimized. In particular, the weight of the new state is $4 \cdot 10 + 4 = 44$.

2. Does s contain the substring 00 Yes.

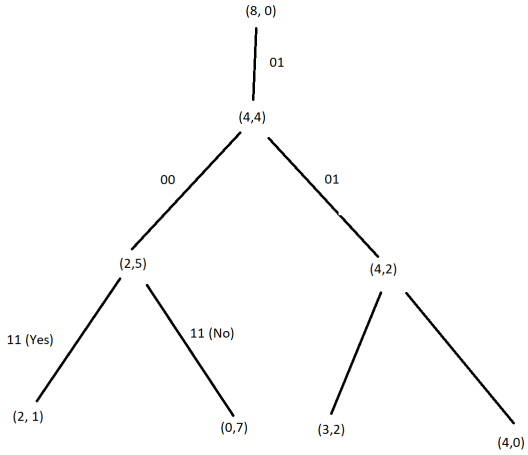
At this point, whatever move is made can do one of three things. It can either remove 2 objects from the set, remove 1, or do the same for the 'NO' set. In particular, since the response to the previous question was 'YES', the 'YES' set is $\{000, 100, 110, 111\}$, and the 'NO' set is $\{001, 010, 011, 101\}$. The weight function implies elements in the lie set are easier to handle than the elements in the truth set. As a result, it would be advantageous to work solely with the elements of the 'Yes' set. The potential states are $(2, 6)$ and $(4, 2)$. Since the corresponding weights are 24 and 38. Thus the best move is to ask if the element contains the substring 00.

3.

Now, the 'YES' set is $\{110, 111\}$, and the 'NO' set is $\{000, 100, 001, 010, 011, 101\}$. Immediately, the most obvious substrings to ask is 11. Answering 'Yes' would make the state (2,1), while answering 'No', would make it (0,7). At this stage in the game, asking for (0,7) is better (since the weight would just be 7). A 'No' would result in a weight of 17. Notice, having a weight of (0,7) prompts a binary search that would end in 3 moves, ending the game in 6 total moves. Trivially, a (2,1) state ends in 3 moves, ending the game in 6 total moves. Thus, while the Berlekamp weight algorithm points us towards answering YES to a 11 substring question, both are sufficient for winning the game.

Figure 3. Sample game with n = 3, k = 6.

Thus, an optimal game of substring search with a lie for size $k = 3$ is will take 6 rounds. The following is a chart of all the states we followed:



This suggests the fairly intuitive idea that playing the search game with a lie complicates the gameplay. In particular, using a greedy algorithm without considering lies takes 5 turns, but playing with a lie takes 6. In addition, determining the best choice using a greedy algorithm is still expensive. Each branch of the search tree has 2^n elements, so it would be just as computationally expensive to check every element, despite determining the hidden element using less moves.

To help determine loose bounds, it may help to view the game from Carol's point of view. Clearly, the best move for Carole is to maximize the weight at any turn. Due to the nature of 'Yes' and 'No' questions, the worst she can do at any point is half the weight. Furthermore, substring questions rarely achieve such a rate (even without a lie), so Carole can usually perform better. Thus, if at any point, the weight of the state exceeds 2^j, where j is the number of questions remaining $(+1)$, then Carole will win. In particular, $k(j+1) < 2^j$, implying $k < \frac{2^j}{j+1}$. Note, this is a generalization of the statement that the number of questions asked must be less than or equal to the search space. Thus we have tightened the initial bound for the number of questions needed to search a size of k from k to j such that $k < \frac{2^j}{j+1}$. These bounds are very lose because substring questions rarely remove half of a set at each turn.

Now, we can begin searching for faster (computationally speaking) algorithms. The most obvious algorithm is try first is the brute force algorithm. We can simply ask if the hidden element is some element of the subset twice. If the answers at any point are contradictory, then we know the lie has been used. In a worst-case scenario, this would take 2^{k+1} guesses, where k is the number of digits in the string. On average, it would take 2^k guesses.

Another faster algorithm would be to repeat the approximation algorithm described in the section on substring search without a lie (mentioned at the bottom of page 10). (On the worst case) If it took u guesses to determine a hidden element using the algorithm without a lie, it would take $2u$ (or $4k$, where k is string length) guesses to do so with a lie (simply repeat every original

question). Average-case analysis is just as straightforward. For a search set of all strings of length k, given a uniform distribution of where the lie could be placed, the lie would be exposed on the $k/2$ term (which would take k total questions), after which, we can proceed normally. This would take $3k$ total questions.

A different modification of the same algorithm can also accomplish the task in slightly less time than the repetitive algorithm.

Assume, by question i, we know that the missing string has subtring s of length i. We simply ask if the string contains the substrings $s^\frown 0$ and $s^\frown 1$. If a response to either of these questions is 'YES', we assume Carole is telling the truth, and we have once again, reached the beginning state. If responses to both questions are a 'NO', then we will repeat the question. If they are still 'NO', then we know we have landed upon a suffix, and can perform prefix search with a lie. Else, we have determine the lie, and we can proceed as if there were no lie.

For worst case analysis, there are two possible cases to consider. The first is that we encounter the lie before reaching the end of the string. The second is the opposite (that we encounter the end of the string before encountering the lie. Due to the nature of the algorithm, both of these cases are prompted when we receive two 'NO's in a row as responses. Thus, we repeat the questions. Suppose we have gone through i questions when we hear two 'NO's as responses. Then, when we repeat them, we will have asked $i+2$ questions (note we have to repeat both because if we hear a 'YES' as a response on the first repeat on the first question, the opponent is not playing optimally). After these $i+2$ questions, we have two possible cases. First we may year a (N Y), or we may hear another (NN). The prior indicates that the lie was made, while the latter indicates that we have reached the end of the string. If a lie was made before the end of the string, we can finish the game in $2(s-\frac{i}{2})+2$ more moves. In particular, there was no benefit in exposing the lie. We would still have to double ask questions in the possibility that we reach the end of the substring. Thus the total number of questions to ask in this case is $i+2+2(s-\frac{i}{2})+2 = 2s+4$ questions.

(Now for the second case). On the other hand, suppose we reach the end of the substring in i moves. Then, we have $i+2$ questions after determining that we have reached the end of the substring. Now suppose that there are are s total bits in the string. Similarly, we would ask $2(s-\frac{i}{2})+2$ more questions and ask a total of $2s+4$ questions.

Both cases result in the same number of questions asked. This indicates, that, in the worst case, they are equivalent. Additionally, both fare far better than the repetitive version of the same algorithm, which took $4s$ questions.

Now we perform average case analysis. Notice that the only difference is that the responder will not necessarily begin his response with a 'No'. Since our algorithm specifies that if a response to either of these questions is 'YES', we assume Carole is telling the truth, the worst case algorithm always begins its response to each pair of questions ($s \frown 0$ and $s \frown 1$) with a 'NO'. On the

average, the responder will begin with a 'No' half the time, and a 'Yes' the other half. Thus, it will take an average of 3/2 questions per move. The additional four questions still stand. Thus, for both cases, the total number of questions to ask is $3/2 \cdot s + 4$.

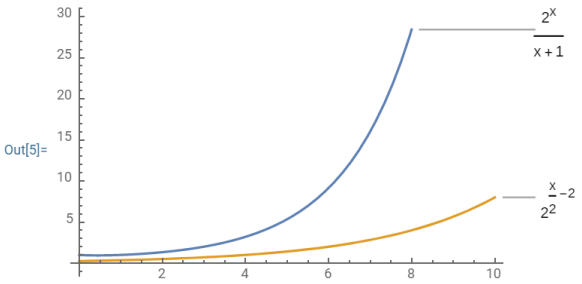
4 Conclusion

If n is the search size and k is the number of questions, then we can rephrase our algorithm in terms of ns and ks. For the sake of a clean form, we can assume n is a power of 2. We showed that If $k > 2s + 4$, then we can determine the hidden element. Then, we have $s = \log_2 n$. Thus:

$$k > 2\log_2 n + 4 \implies 2^k > 2^{2(\log_2 n)+4} \implies n < 2^{k/2-2}.$$

If $n <= 2^{k/2-2}$, then the questioner wins and if $n > \frac{2^k}{k+1}$, the responder wins. The bounds for the losing position cannot be compared because the bound for substring search was lifted directly from the arbitrary q-ary question bound. The bound for the questioner's win is $n < \frac{2^{k-1}}{k+1}$. This is greater than $2^{k/2-2}$ for all values of k. This is true via straightforward induction. For a visual depiction, a graph is presented:

```
In[5]:= x2 = Plot[{(2^x)/(x+1), 2^(x/2 - 2)}, {x, 0, 10}, PlotLabels → "Expressions"]
```



We cannot compare directly to prefix questions because Pelc [4] does not give an explicit bound.

Note if any of the conditions on substring search are relaxed, then the game becomes less interesting. If we are allowed to ask about position of the substrings, then it is easy to show that this reduces to a standard Remyi Ulam liar game. They same is true if we are allowed to compound ANDs and ORs to our questions.Furthermore, searching for nonadaptive algorithms is fruitless. Substring search algorithms will not necessarily contribute to faster error correcting codes. Furthermore, it is highly unlikely that they will perform better than adaptable algorithms.

5 Acknowledgements

I would like to thank Ms. Spigner for reading over my paper and giving appreciated advice. I would also like to thank my parents for their support.

6 Bibliography

[1] Ellis, R.B., Ponomarenko, V. Yan, C.H. How to play the one-lie Renyi-Ulam game (2007). *Journal of Combinatorial Theory. 48(2)*165–173

[2]Guzicki, W. (1990). Ulam's Searching Game with Two Lies. *Journal of Combinatorial Theory . 54(1)* 1-19.

[3]Pelc, A. (1987). Solution of Ulam's Problem on searching with a lie. *Journal of Combinatorial Theory . 44(1)* 129-140.

[4] Pelc, A. Prefix search with a lie. (1988). *Journal of Combinatorial Theory. 48(2)*165–173

[5] Rivest, R. L., Meyer, A. R., Kleitman D. J., Winklmann, K. Coping with Errors in Binary Search Procedures. (1980). *Journal of Computer Science and System Sciences. 20(3)*227-232

[6] Spencer, J. (1984). Guess a Number- with Lying. *Mathematical Association of America. 57(2)* 105-108.