Q2.2) Suppose the secret word is randomly chosen from a dictionary with exactly 2^k–1 words, where *k* is a positive integer. Describe an algorithm that guarantees that you can identify the secret word in at most *k* guesses. Clearly justify how and why your algorithm works.

We will use Binary Search to solve this problem. **Binary search** is an efficient method for searching a word in the dictionary of words.

The idea behind this algorithm is as follows, if the word to search starts with the letter S, one usually starts searching the dictionary around the half-way mark. Since, it is commonly known that S is located in the second half of the alphabet, it would not make sense to start searching for S from the dictionary's start.

Algorithm/Pseudocode

```
function wordGuess(arr, x):
   Input: a lexicographical sorted array 'arr' of words
   Output: number of guesses taken to guess the word
   arr = []
   # random word is selected by the computer from the list
   word = random.choice(arr)

   # calculations performed by the user
   def user_guess():
      # setting the lowerbound
      low = 0
      # setting the upperbound
      high = len(arr)
      mid = math.ceil((low+high)/2)
      guessesTaken=0
      while high>low or high==low:
         guessesTaken+=1
         # setting midpoint of the subarray
         mid = math.ceil((low+high)/2)
         n = guessing_algorithm(arr[mid])
         if n == 0:
            # setting the lowerbound of the subarray
            low = mid + 1
        elif n == 1:
            # setting the upperbound of the subarray
            high = mid - 1
        # condition where word is guessed correctly
        elif n == -1:
            break
    print(f"The user guessed {word} in {guessesTaken} guesses!")

# computer output displayed to the user
def guessing_algorithm(guess):
    print("User guessed -", guess)
    if guess < word:
        print("My word is after " + str(guess))
        return 0
    elif guess > word:
        print("My word is before " + str(guess))
        return 1
    elif guess == word:
        print("You guessed the word!!")
        return -1
```

Runtime Analysis

The time complexity of the algorithm is O(logn) in the worst case because the array index is divided into halves at each recursive call and the middle value of dictionary is checked. In best case, when the middle value of the dictionary is equal to the word, time complexity is O(1). **However, the average time complexity to guess the word will be O(logn), where n is the number of words in the array, which means that given 2^k - 1 words in a dictionary, the algorithm identifies the word in at most k guesses.**

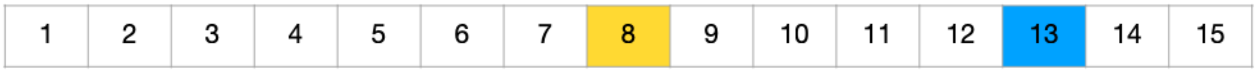
WordGuess Algorithm Explanation

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero. The time complexity of binary search algorithm is O(logn).

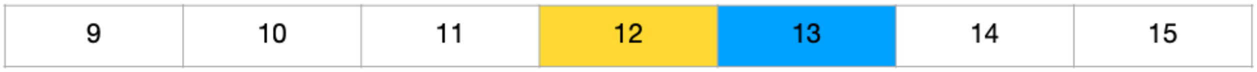
Guess - 🟨

Match - 🟦

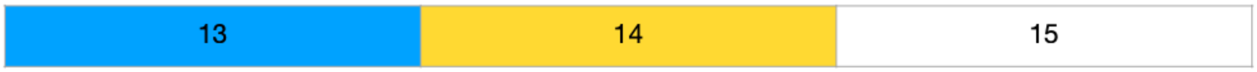
The input in this problem is dictionary of 2^k - 1 words, where k = 4, so 15 words in the array ranging between minIndex=1 and maxIndex=15. Suppose the word to be guessed is at index 13. The guessing game starts. We will equally divide the range in two halves and guess the middle word which is at index 8.



We will get a message 'My word is after the word at index 8'. So, now our minIndex=9 and maxIndex=15 and the subarray will be from indices 9 to 15. We will equally divide the range in two halves and guess the middle word which is index 12.



Now, we will get a message 'My word is after the word at index 12'. So, now our minIndex=13 and maxIndex=15. We will again equally divide the range in two halves and guess the middle word which is 14.



Now, we will get a message ‘My word is before the word at index 14’. So, now our minIndex=13 and maxIndex=13. So we will guess the word at index 13 which is the only word left to guess.

13

Now, we will get a message ‘Well done! It took you 4 attempts to guess this word’. The game ends.

Here, we guessed the word in at most 4 guesses using the Binary search algorithm.

Now, if we check how many guesses it would take to guess all the words in the index range from 1 to 15, by guessing middle words in the subarray.

1 = 8, 4, 2, 1 = 4 guesses
2 = 8, 4, 2 = 3 guesses
3 = 8, 4, 2, 3 = 4 guesses
4 = 8, 4 = 2 guesses
5 = 8, 4, 6, 5 = 4 guesses
6 = 8, 4, 6 = 3 guesses
7 = 8, 4, 6, 7 = 4 guesses
8 = 8 = 1 guess
9 = 8, 12, 10, 9 = 4 guesses
10 = 8,12, 10 = 3 guesses
11 = 8, 12, 10, 11 = 4 guesses
12 = 8, 12 = 2 guesses
13 = 8, 12, 14, 13 = 4 guesses
14 = 8, 12, 14 = 3 guesses
15 = 8, 12, 14, 15 = 4 guesses

Here, we can see that the maximum number of guesses it takes to guess the word in the index range from 1 to 15 is **at most 4 guesses**.
Hence, the algorithm holds true and if the secret word is randomly chosen from a dictionary with exactly $\mathbf{2^k-1}$ words, where k is a positive integer, then we can identify the secret word in at most $\boldsymbol{k}$ guesses.

Proof of Correctness

At the start of each iteration of the while loop above the following is true

- $1 \leq \text{first} \leq \text{last} \leq n$
- If the word to be guessed is in the array and so there is some index such that a[index] = word and $1 \leq \text{index} \leq n$ then $\text{first} \leq \text{index} \leq \text{last}$. This just means that we are correctly picking the right range to search at every iteration.

Initialization:
Before the first iteration we know that first=1 and last=n and so (1) is trivially true. Also, if the word is in a then it must be that $1 \leq \text{index} \leq n$ where a[index]=word. Therefore, we are searching the correct range.

Maintenance:
Suppose the invariant holds before iteration i and that if word is in a then for some index where a[index]=word we have $\text{first} \leq \text{index} \leq \text{last}$. Let's look at iteration i+1. We first calculate the mid point in the range [first, last]. Based on this, we have three cases:

(1) a[m]==word. We therefore, return the number of guesses and the user wins the game.

(2) a[m]>word. Algorithm in this case searches the new range [first, m−1]. We know that $\text{first} \leq \text{index}$ by the inductive hypothesis and so we only need to prove that $\text{index} \leq m-1$. Since a[m]>word and since the array is lexicographically sorted then it must be that index<m is true otherwise a[index] comes after a[m] which indicates that a is not sorted.

(3) a[m]>word. Algorithm will then search the new range [m+1, last]. We can use a similar argument to (2) to prove that $\text{index} \in [m+1, \text{last}]$.

From (1), (2), (3) we conclude that the invariant holds before iteration i+1.

Termination:

We need to prove that if the search range is [first_i, last_i] in iteration i and also that the search range in iteration i+1 is [first_i+1, last_i+1] then it must be that $\text{last}_i+1-\text{first}_i+1<\text{last}_i-\text{first}_i$. To do so we can use a similar approach to the one we used to prove the maintenance step. We know there are three cases, in each case we can prove that the new range is shrinking.

Suppose m is the midpoint in [first_i, last_i]. There are three cases:

- a[m]==word. We therefore, return the guesses taken and we are done (binary search terminates).
- a[m]>word. Algorithm in this case searches the new range [first, m−1]. We see here that $m-1-\text{first}_i<\text{last}_i-\text{first}_i$.
- a[m]>word. Algorithm will then search the new range [m+1, last]. Similarly, we see here that $\text{last}-m+1<\text{last}_i-\text{first}_i$.

From the three cases we conclude that binary search must terminate.