Tharun Saravanan

Professor Simha

Algorithms

10 October 2022

Assignment 2 Part 1

Exercise 1:

This function repeatedly divides the number of summations performed through the use of an iterating counter. The first run of the outer loop will make a total of n summations as it will perform n/1 summations in the inner loop. The next run will thus make a total of n/2 summations, the third will make n/3 summations, and so on. This repeated division of calculations is characteristic of a logarithmic function. Because the first run of the loop will execute n times, and subsequent runs will continue to decrease from this first run using this logarithmic pattern, we can express the Big-Oh efficiency as O(nlog(n)).

Exercise 2:

$$O\ (n * (n^2 + log(n) * (3*n + (3*n^2)/log(n)\ )\ )\ )$$

Simplifying the following expression we get:

$O(n * (n^2 + (3n \log(n) + 3n^2)))$

$O(n * (3n \log(n) + 4n^2))$

$O(4n^3 + 3n^2 \log(n))$

Taking the most significant term of this expression, we have an efficiency of $O(n^3)$

Exercise 3:

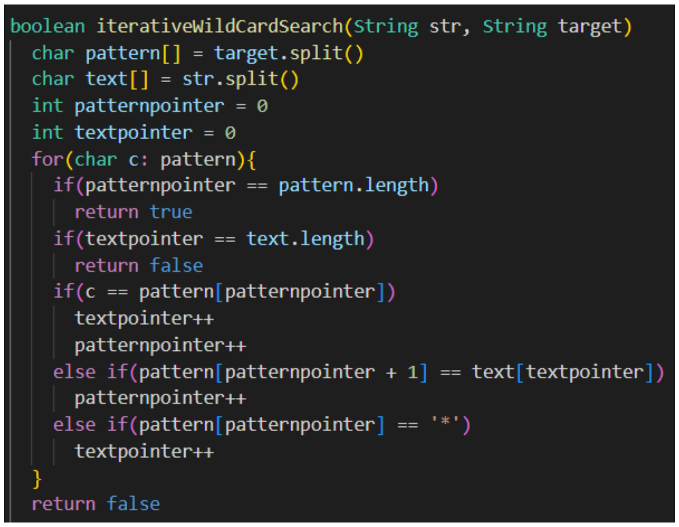
The total number of nodes in a full trie is equal to 2^{n+1} - 1 using the geometric series formula (Where n represents the height). If we then create a table of full tries and how many respective nodes they have:

Total Nodes	Number of Leaves	Number of Internal Nodes
1	1	0
3	2	1
7	4	3
15	8	7
31	16	15
63	32	31

As we can see here, the number of internal nodes is equal to the total number of the nodes in the previous row, while the number of leaves is equal to the next power of 2. Summing all of the powers of 2 (number of nodes) excluding the last row (leaf level) is roughly equal to the total number of leafs because the major operation in calculating the number of nodes is 2^{n+1}. This operation doubles the number of nodes each time we reach a new full trie layer, which is why if we equate all of the nodes added at the leaf layer (all of the new nodes) to O(n), the internal nodes can be expressed as O(n) as well.

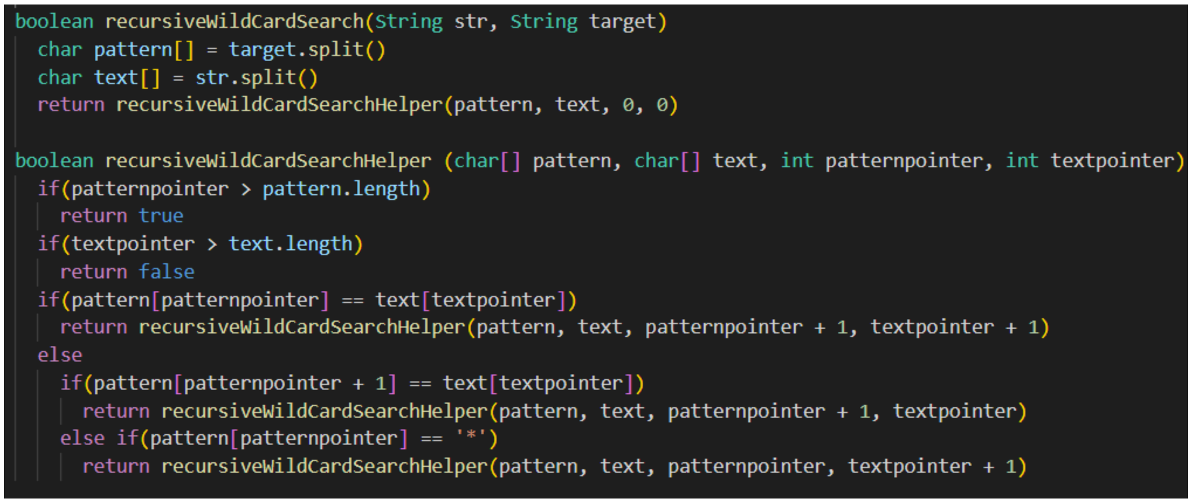
Exercise 4:

Iterative wildcard search:



```
boolean iterativeWildCardSearch(String str, String target)
  char pattern[] = target.split()
  char text[] = str.split()
  int patternpointer = 0
  int textpointer = 0
  for(char c: pattern){
    if(patternpointer == pattern.length)
      return true
    if(textpointer == text.length)
      return false
    if(c == pattern[patternpointer])
      textpointer++
      patternpointer++
    else if(pattern[patternpointer + 1] == text[textpointer])
      patternpointer++
    else if(pattern[patternpointer] == '*')
      textpointer++
  }
  return false
```

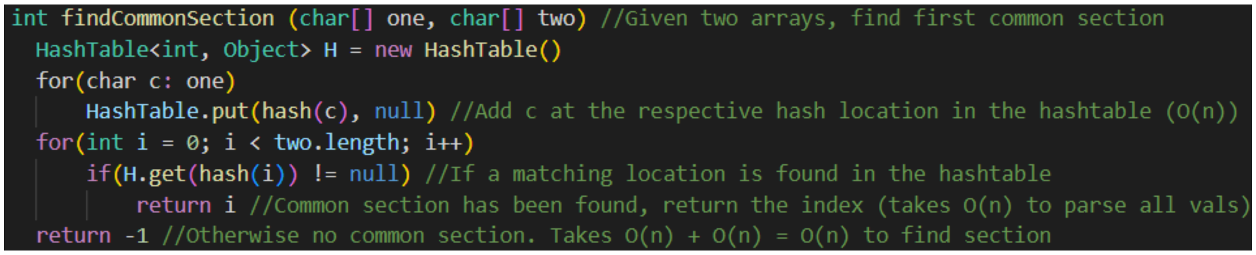
Recursive wildcard search:



```
boolean recursiveWildCardSearch(String str, String target)
  char pattern[] = target.split()
  char text[] = str.split()
  return recursiveWildCardSearchHelper(pattern, text, 0, 0)

boolean recursiveWildCardSearchHelper (char[] pattern, char[] text, int patternpointer, int textpointer)
  if(patternpointer > pattern.length)
    return true
  if(textpointer > text.length)
    return false
  if(pattern[patternpointer] == text[textpointer])
    return recursiveWildCardSearchHelper(pattern, text, patternpointer + 1, textpointer + 1)
  else
    if(pattern[patternpointer + 1] == text[textpointer])
      return recursiveWildCardSearchHelper(pattern, text, patternpointer + 1, textpointer)
    else if(pattern[patternpointer] == '*')
      return recursiveWildCardSearchHelper(pattern, text, patternpointer, textpointer + 1)
```

Exercise 5:



```
int findCommonSection (char[] one, char[] two) //Given two arrays, find first common section
  HashTable<int, Object> H = new HashTable()
  for(char c: one)
      HashTable.put(hash(c), null) //Add c at the respective hash location in the hashtable (O(n))
  for(int i = 0; i < two.length; i++)
      if(H.get(hash(i)) != null) //If a matching location is found in the hashtable
          return i //Common section has been found, return the index (takes O(n) to parse all vals)
  return -1 //Otherwise no common section. Takes O(n) + O(n) = O(n) to find section
```