Laurence Labayen
Nov 03, 2019

Lab #5

CS2302 Data Structures - MW 10:30am

Professor: Dr. Olac Fuentes
T.A.: Anindita Nath

Fall 2019

Description:

With Lab 5, we were asked to implement the same functionality as the previous lab assignment which was to retrieve word embeddings using different data structures but this time, using hash tables. Using both Linear probing and Chaining implementation to solve the same problem. Additionally, we were assigned to use different hash functions as follows:

- *The length of the string % n*
- *The ascii value (ord(c)) of the first character in the string % n*
- *The product of the ascii values of the first and last characters in the string % n*
- *The sum of the ascii values of the characters in the string % n*
- *The recursive formulation h(",n) = 1; h(S,n) = (ord(s[0]) + 255*h(s[1:],n))% n*
- *Another function of your choice*

Solution design and Implementation:

Using the same approach as the previous lab, I opened and read each line of the GLoVe file from the NLP website. While reading the lines, I inserted the word and it's embedding into a Node that holds both, then I inserted each node into the corresponding data structure. To find the correct position in the table, I used the word inside each node with the hash functions that were given to us. I first did experimental runs using chaining, it worked perfectly with longer running times than the previous lab for the first 4 hash functions. When I got to the linear probing implementation, things were much slower. I decided to only read a small portion of the GLoVe file to shorten the running times and be able to compare everything in more detail. I ended up reading 6,400,000 lines from the file to get substantial running time data without taking all day to construct each different hash functions with different load factors. Upon doing so, I decided to have different options for the load factor. Before creating the object for each hash table implementation, I set up a menu to ask which function the user would like to use along with the desired load factor.

Chaining test function

```python
def HashChain_Test():

    choice, table_size = menu()

    H = HashTableChain(table_size)
    # Pattern to be used to remove words with unwanted characters
    pattern=re.compile("[A-Za-z]+")
    print('Loading glove file...')
    # Open glove file
    file = open('glove.6B.50d.txt','r')
    count=0
    # Start counter
    start = time.perf_counter()

    # readlines limited to a small sample of the GLoVe file to reduce times of certain
    # hash functions
    for line in file.readlines(6400000):
         row = line.strip().split(' ')
         # Check if word matches the pattern of characters
         if pattern.fullmatch(row[0]) is not None:
             # Insert into Hash Table with word and its embedding
             H.insert(WE_Node(row[0],[(i) for i in row[1:]]),choice)
             count+=1
    # Stop counter
    end = time.perf_counter()
```

Linear Probing test function

```python
def HashTableLP_Test():

    choice, table_size = menu()

    H = HashTableLP(table_size)

    # Pattern to be used to remove words with unwanted characters
    pattern=re.compile("[A-Za-z]+")
    print('Loading glove file...')
    # Open glove file
    file = open('glove.6B.50d.txt','r')

    # Start counter
    start = time.perf_counter()
    count=0
    # readlines limited to a small sample of the GLoVe file to reduce times of certain
    # hash functions
    for line in file.readlines(6400000):
         row = line.strip().split(' ')
         # Check if word matches the pattern of characters
         if pattern.fullmatch(row[0]) is not None:
             # Insert into Hash Table with word and its embedding
             H.insert(WE_Node(row[0],[(i) for i in row[1:]]),choice)
             count+=1
    # Stop counter
    end = time.perf_counter()
```

Given Hash Functions:

```python
# Hash function with length of string k % size of table
def lenword_hash(self,k):
    if isinstance(k, WE_Node):
        k=k.word

    return len(k)%len(self.bucket)
# Hash function with ASCII value of the first character of k % size of table
def ascii_first_hash(self,k):
    if isinstance(k, WE_Node):
        k=k.word
    return ord(k[0])%len(self.bucket)

# Hash function with product of ASCII values from first and last char % size of table
def ascii_product_hash(self,k):
    if isinstance(k, WE_Node):
        k=k.word
    return (ord(k[0])*ord(k[-1]))%len(self.bucket)

# Hash function with the sum of the ASCII values in k % size of table
def ascii_sum_hash(self, k):
    if isinstance(k, WE_Node):
        k=k.word
    return sum(map(ord, k))%len(self.bucket)

# Recursive Hash function that multiplies the ASCII values of all the characters
# in k (plus 255 on each value) % size of table
def recursive_hash(self, k):
    if isinstance(k, WE_Node):
        k=k.word

    if len(k) == 0:
        return 1
    return (ord(k[0]) + 255 * self.recursive_hash(k[1:])) % len(self.bucket)
```

Custom Hash Functions:

For the custom hash function, I decided to create 2 different ones as I was curious to see if it would make a huge impact using recursion vs iteration. For the first one, the function iterates through the word found inside the word embedding node. Each character is converted to its ASCII value then raised to the power of the current iteration. It returns the total sum modulo table length

```python
# Custom Hash function done with a loop to add all the ASCII
# values in k to the power of it's index % size of table
def custom_hash(self, k):
    if isinstance(k, WE_Node):
        k=k.word
    total=0
    for i in range(len(k)):
        total+=ord(k[i])**i

    return total % len(self.bucket)
```

As for the second custom hash function, I based it on the recursive function that was given in the instructions. The table size is integer divided to the ASCII value of each character in the input string/word then. It returns the total product modulo table size.

```python
# Custom recursive Hash function that uses the size of the table and int divides
# to the ASCII value of each character in k, then each is multiplied by the next
# character. Returns the product of all these values % of the size of the table
def custom_hash2(self, k):
    if isinstance(k, WE_Node):
        k=k.word
    if len(k) == 0:
        return 1
    return (len(self.bucket)//ord(k[0]) * self.custom_hash(k[1:])) % len(self.bucket)
```

Similarity test function

```python
def Similarity(H,choice, file_choice, num_pairs):

#    numpairs = int(input('Enter # of pairs to compare: ') )

    # Open and read pairs word file and insert into a list as list of pairs
    pairs=[line.strip().split(' ') for line in open(file_choice,'r')]
    random.shuffle(pairs)

    # Assign timer to 0
    timer=0

    # Loop to iterate through list of pairs line by line
    for i in range(num_pairs):
        # Start timer
        start = time.perf_counter()

        # word1 gets the first column in each line from pairs list
        word1=pairs[i][0]
        # word2 gets the second column in each line from pairs list
        word2=pairs[i][1]

        #word1emb and word2emb gets the embedding that is found by
        word1emb=(H.find_emb(word1,choice))
        word2emb=(H.find_emb(word2,choice))

        # Check if word1emb or word2emb is not found
        if word1emb is None or word2emb is None:
            continue

        # Formula to find cosine distance between both word embeddings
        cosine_distance = np.dot(word1emb, word2emb)/(np.linalg.norm(word1emb)* np.linalg.norm(word2emb))
        # Stop timer for every iteration
        end = time.perf_counter()
```

Experimental Results:

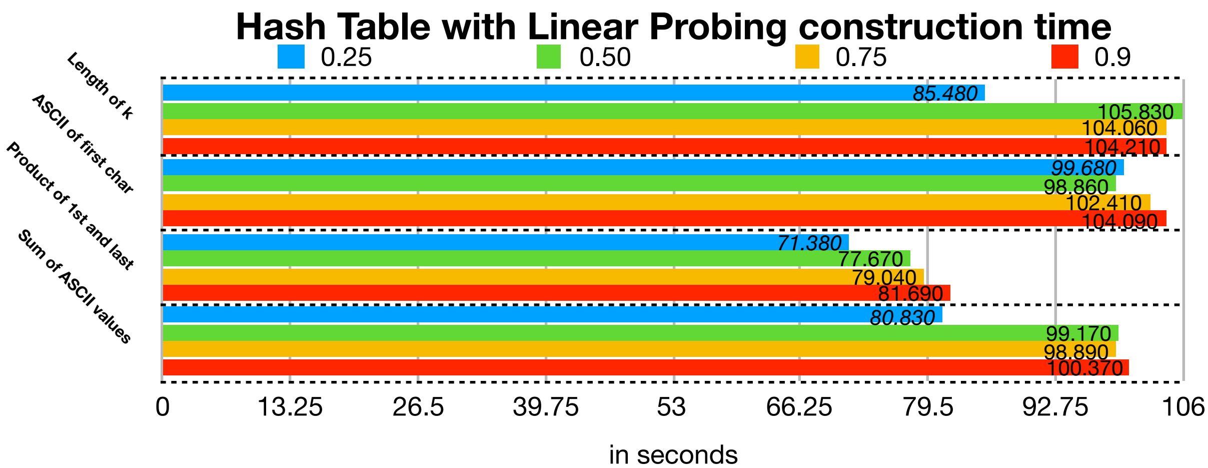
The construction results are based on comparing different load factors (0.25, 0.5, 0.75, 0.9) and reading only 6,400,000 from the GLoVe file. Additionally, I've separated the first 4 hash functions from the recursive and custom functions as they the difference makes the data chart difficult to read. A complete comparison of all the functions is also shown below the separated charts which includes comparisons with the Binary Search Tree and B-Tree. Similarity search results are all based on 300 pairs of words in a "pairs.txt"

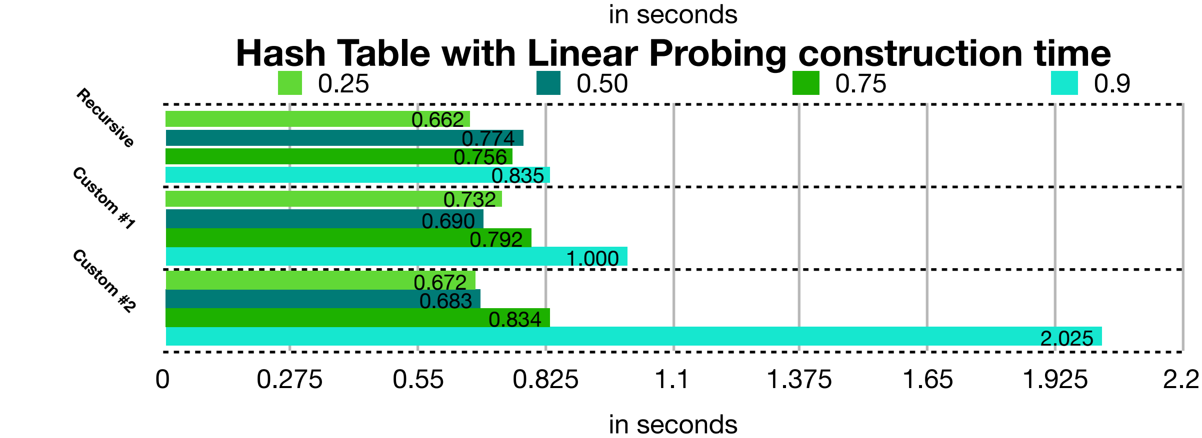
```
Running time for similarities: 0.6406

Hash Table with Linear Probing stats with choice 4
Running time for construction: 80.83442

Table size: 56432
Load factor: 0.25
```

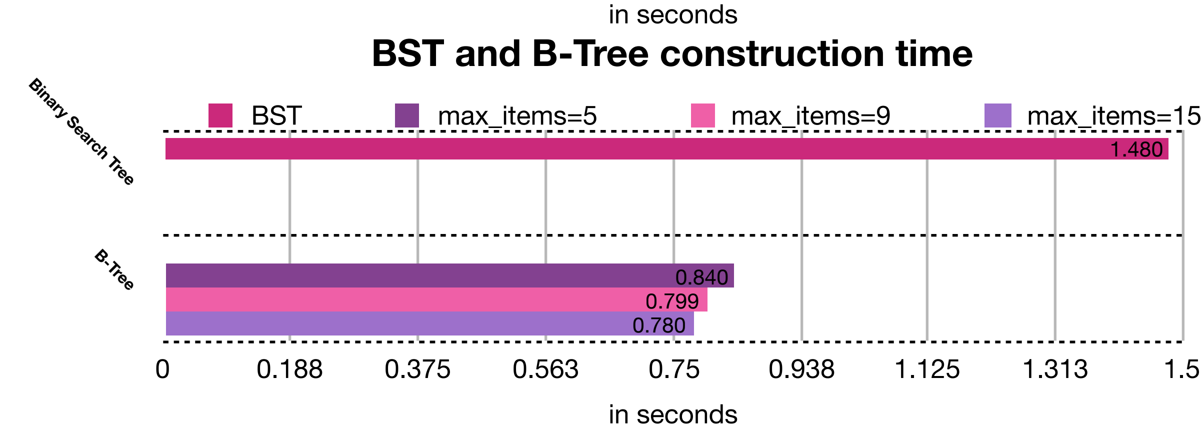
```
Running time for similarities: 0.0189

Hash Table with Linear Probing stats with choice 5
Running time for construction: 0.662101

Table size: 56432
Load factor: 0.25
```

```
Running time for similarities: 0.0233

Hash Table with Linear Probing stats with choice 6
Running time for construction: 1.002347

Table size: 15505
Load factor: 0.9099
```

```
Running time for similarities: 0.0217

Hash Table with Linear Probing stats with choice 7
Running time for construction: 2.025544

Table size: 15505
Load factor: 0.9099
```





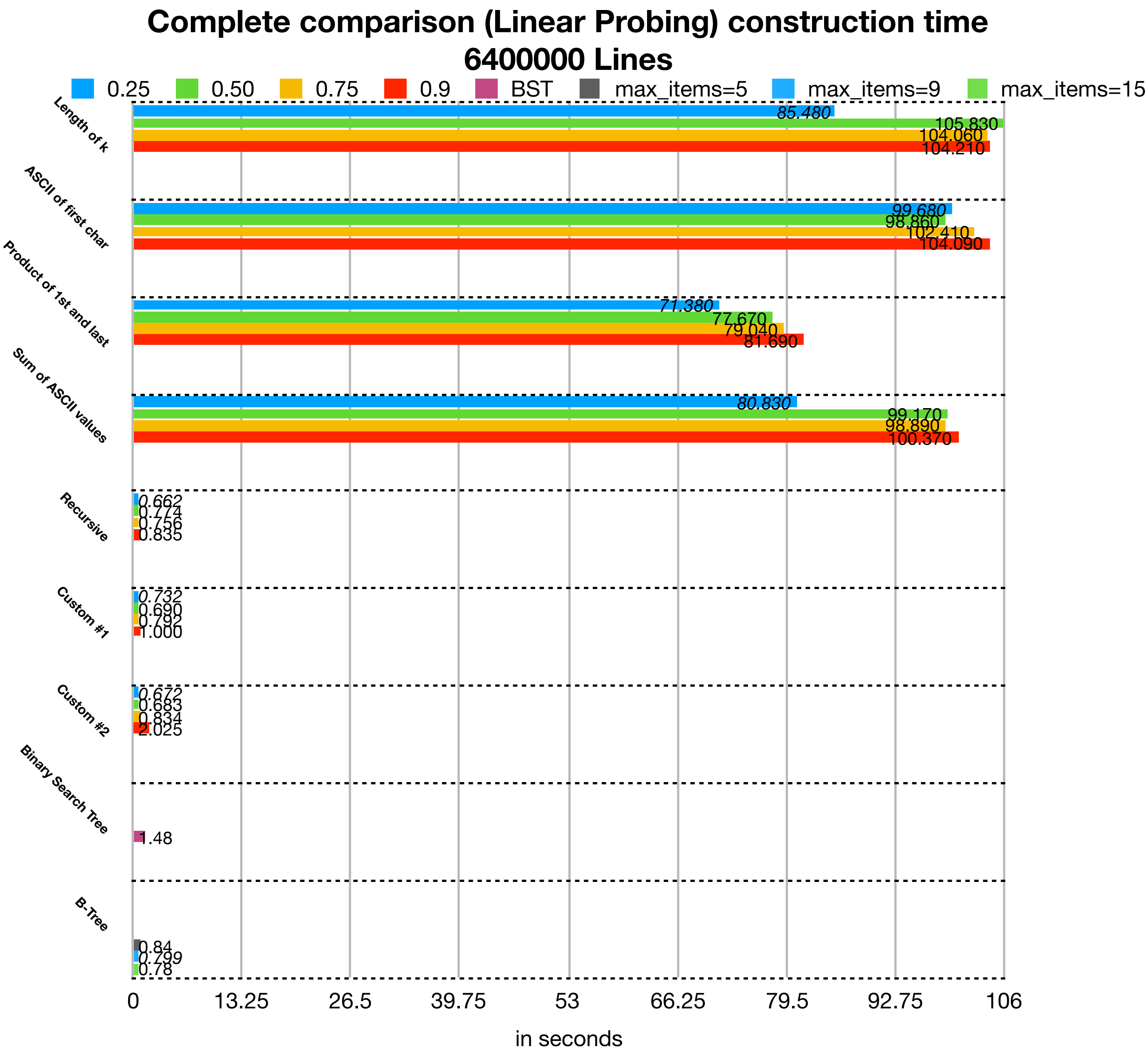


```
Running time for similarities: 0.0213

Hash Table with Linear Probing stats with choice 6
Running time for construction: 0.732167

Table size: 56432
Load factor: 0.25
```

```
Running time for similarities: 0.0247

Hash Table with Linear Probing stats with choice 7
Running time for construction: 0.672744

Table size: 56432
Load factor: 0.25
```

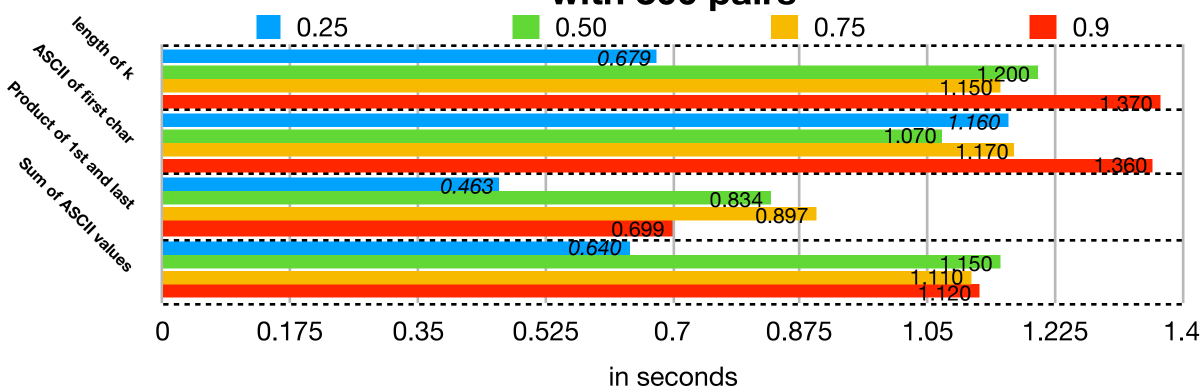
```
Running time for similarities: 0.8972

Hash Table with Linear Probing stats with choice 3
Running time for construction: 79.049898

Table size: 18810
Load factor: 0.750027
```

```
Running time for similarities: 1.115

Hash Table with Linear Probing stats with choice 4
Running time for construction: 99.171415

Table size: 28216
Load factor: 0.5
```

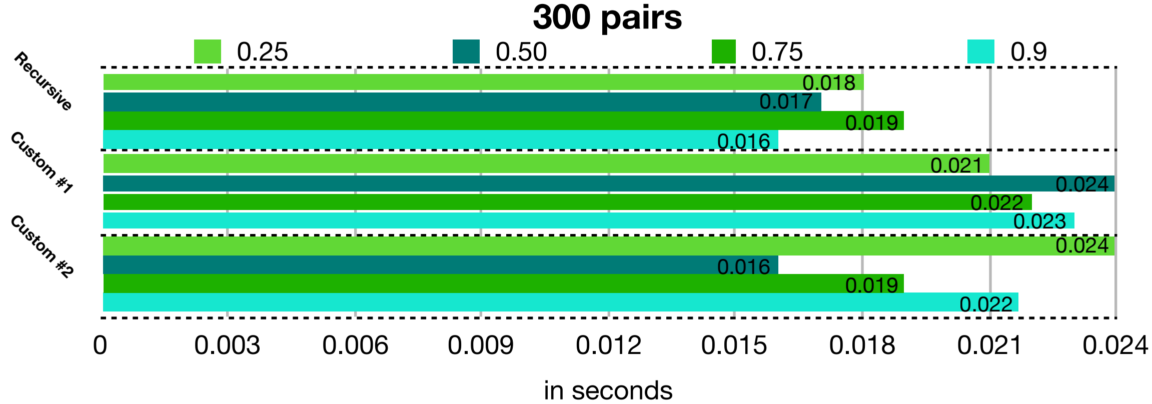


```
Similarity [club,bat] = 26.14189%
Similarity [toe,feet] = 44.78605%
Similarity [eye,glasses] = 52.75459%
Similarity [socks,foot] = 41.0482%
Similarity [glove,hand] = 54.83486%
Similarity [closet,clothes] = 61.17403%
Similarity [mechanic,tools] = 31.49849%
Similarity [doctor,professional] = 50.66509%
Similarity [element,atom] = 59.87259%
Similarity [bench,chair] = 57.81706%
Similarity [garage,car] = 69.51435%
Similarity [output,input] = 65.4709%
Similarity [mexico,spain] = 75.13765%
Similarity [africa,america] = 62.50182%
Similarity [europe,asia] = 83.4683%
Similarity [italy,spain] = 86.16418%
Similarity [logical,reasoning] = 79.4257%
Similarity [moral,ethics] = 66.82629%
Similarity [psychology,sociology] = 89.05489%
Similarity [statistics,numbers] = 66.82225%
Similarity [history,world] = 70.91538%
Similarity [digital,analog] = 78.40965%
```

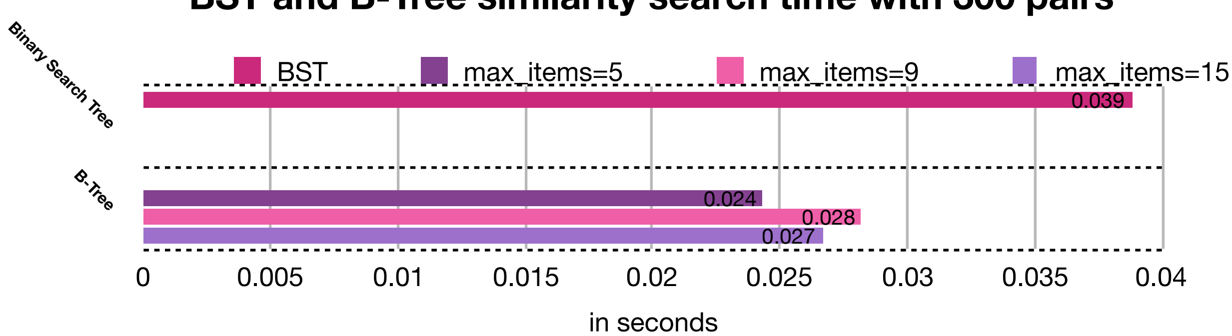
Hash Table with Linear Probing similarity search time with 300 pairs



Hash Table with Linear Probing similarity search time with 300 pairs



BST and B-Tree similarity search time with 300 pairs

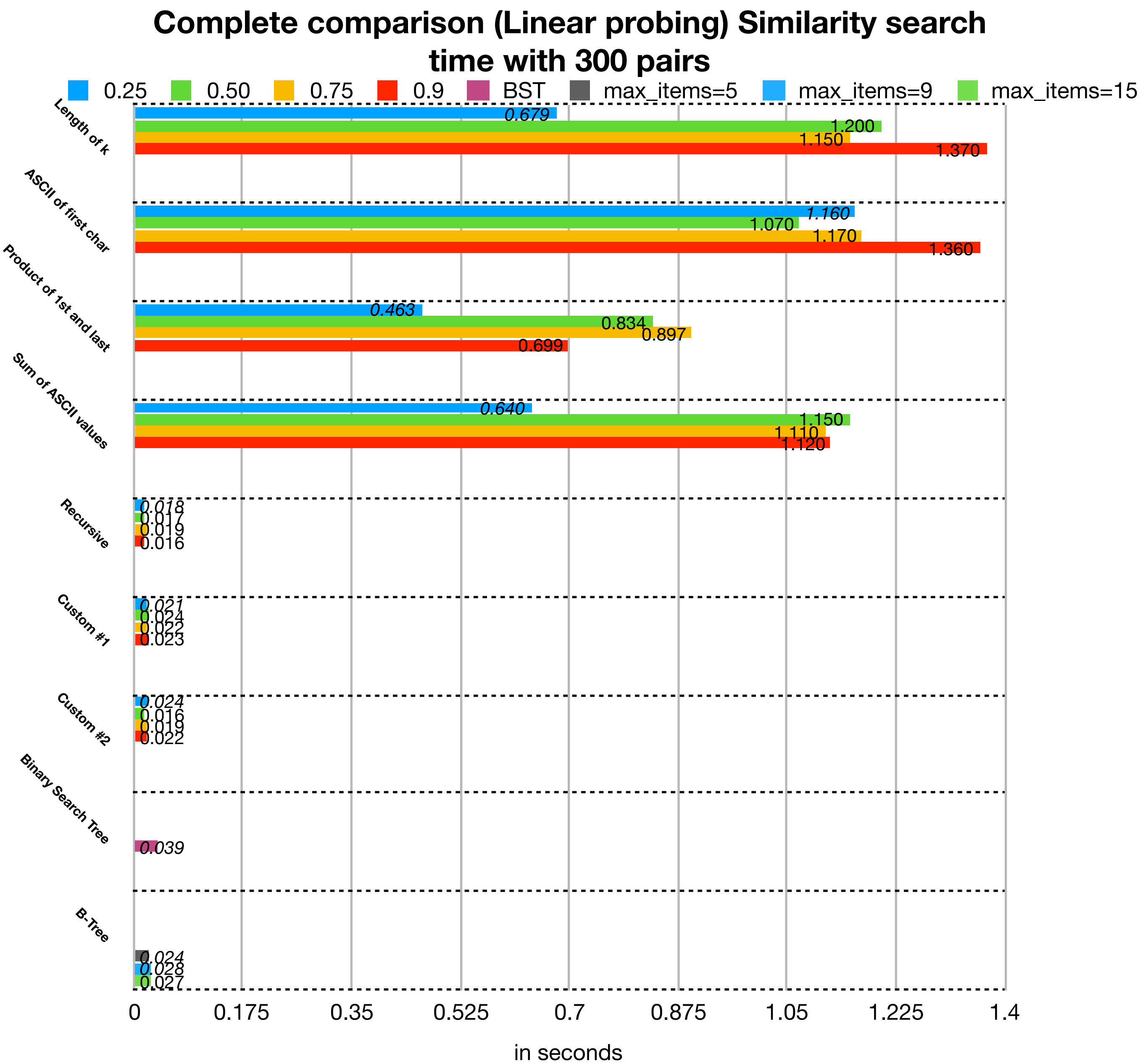


```
Running time for similarities: 0.0198

Hash Table with Linear Probing stats with choice 7
Running time for construction: 0.834602

Table size: 18810
Load factor: 0.750027
```

```
Running time for similarities: 0.0217

Hash Table with Linear Probing stats with choice 7
Running time for construction: 2.025544

Table size: 15505
Load factor: 0.9099
```

```
Running time for similarities: 1.1274

Hash Table with Linear Probing stats with choice 4
Running time for construction: 100.37853

Table size: 15505
Load factor: 0.9099
```

```
Running time for similarities: 0.0233

Hash Table with Linear Probing stats with choice 6
Running time for construction: 1.002347

Table size: 15505
Load factor: 0.9099
```

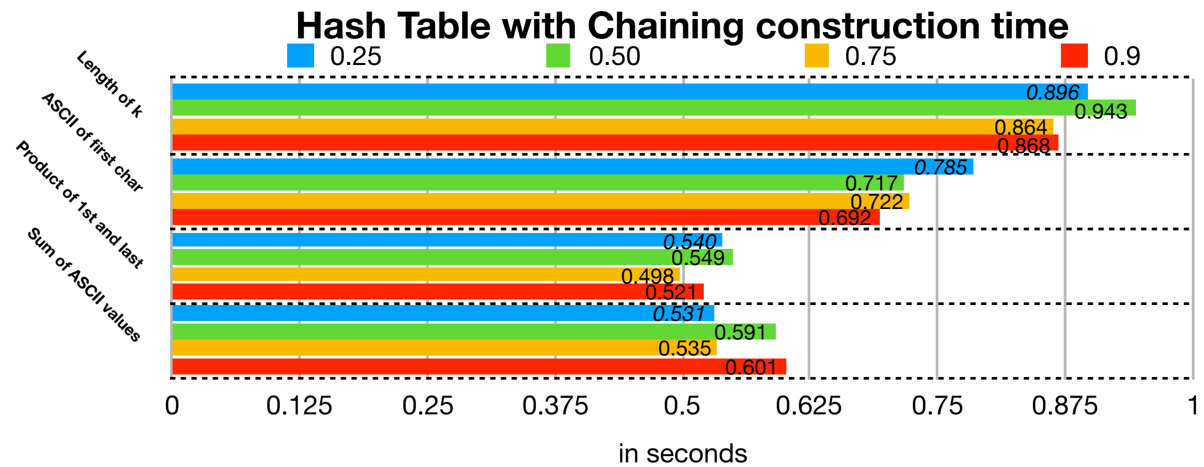


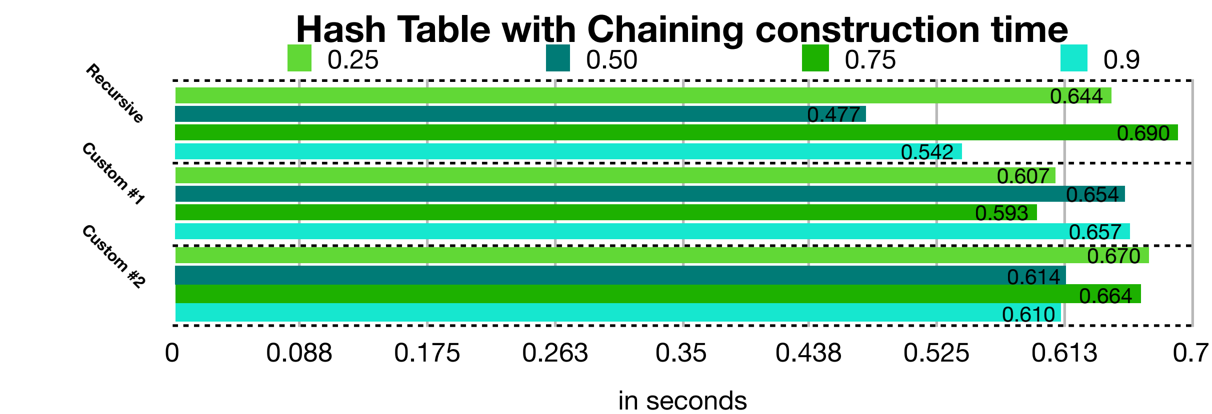
```
Running time for similarities: 0.0202

Hash Table with Chaining stats with choice 3
Running time for construction: 0.549652

Table size: 28216
Load factor: 0.5
```

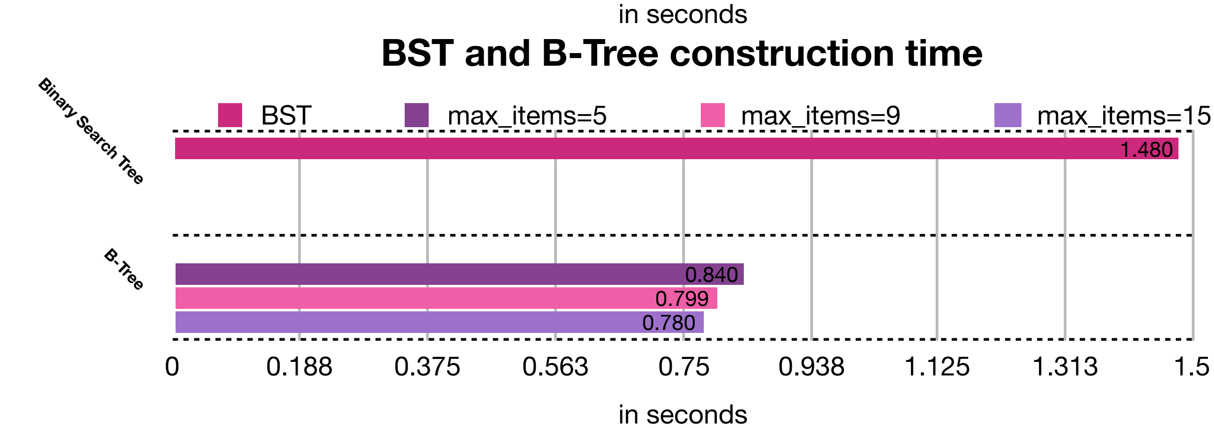
```
Running time for similarities: 0.0217

Hash Table with Chaining stats with choice 6
Running time for construction: 0.654061

Table size: 28216
Load factor: 0.5
```

```
Running time for similarities: 0.0174

Hash Table with Chaining stats with choice 7
Running time for construction: 0.664146

Table size: 18810
Load factor: 0.750027
```

```
Running time for similarities: 0.0772

Hash Table with Chaining stats with choice 1
Running time for construction: 0.864377

Table size: 18810
Load factor: 0.750027
```







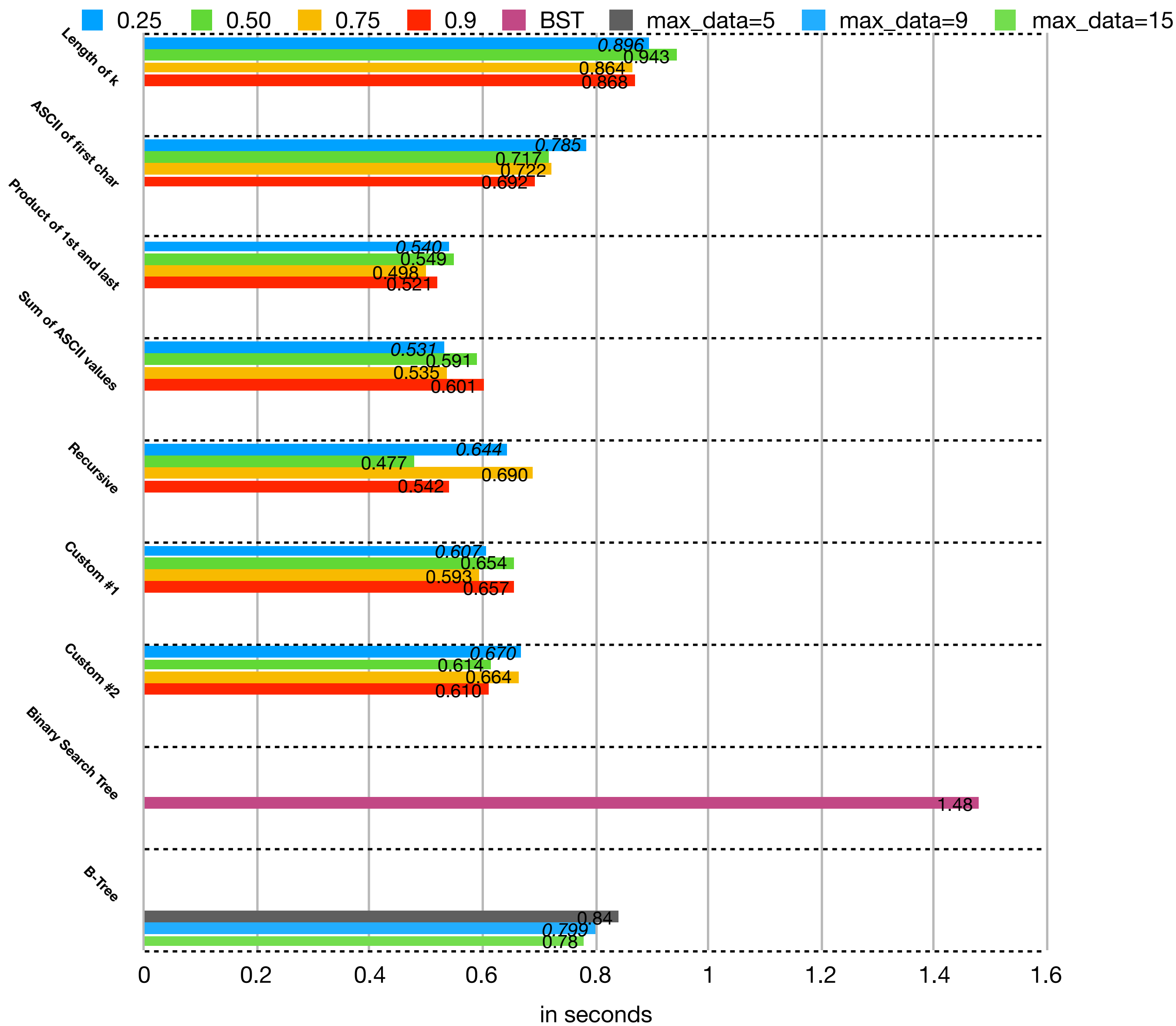
```
Running time for similarities: 0.0217

Hash Table with Chaining stats with choice 6
Running time for construction: 0.654061

Table size: 28216
Load factor: 0.5
```

```
Running time for similarities: 0.0224

Hash Table with Chaining stats with choice 3
Running time for construction: 0.54013

Table size: 56432
Load factor: 0.25
```

```
Running time for similarities: 0.0772

Hash Table with Chaining stats with choice 1
Running time for construction: 0.864377

Table size: 18810
Load factor: 0.750027
```

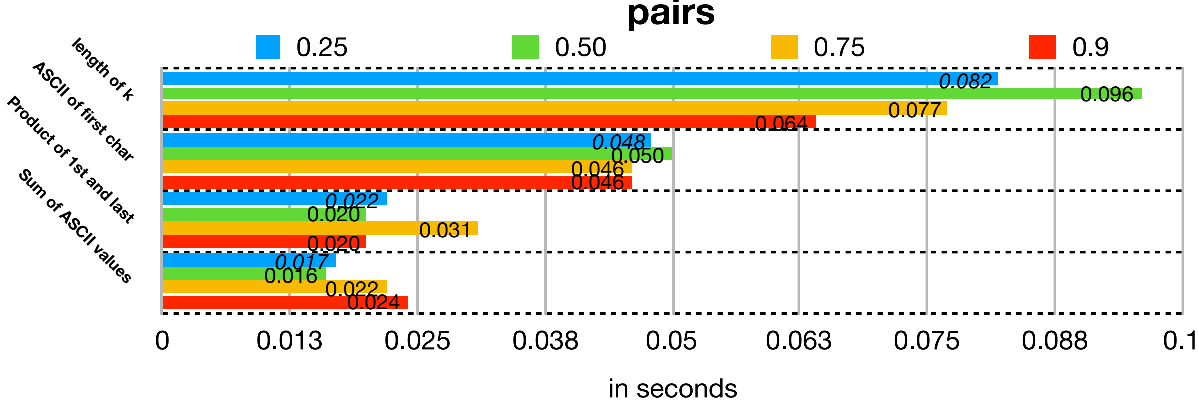
```
Running time for similarities: 0.046

Hash Table with Chaining stats with choice 2
Running time for construction: 0.692885

Table size: 15505
Load factor: 0.9099
```

Complete comparison (Chaining) construction time
6400000 Lines

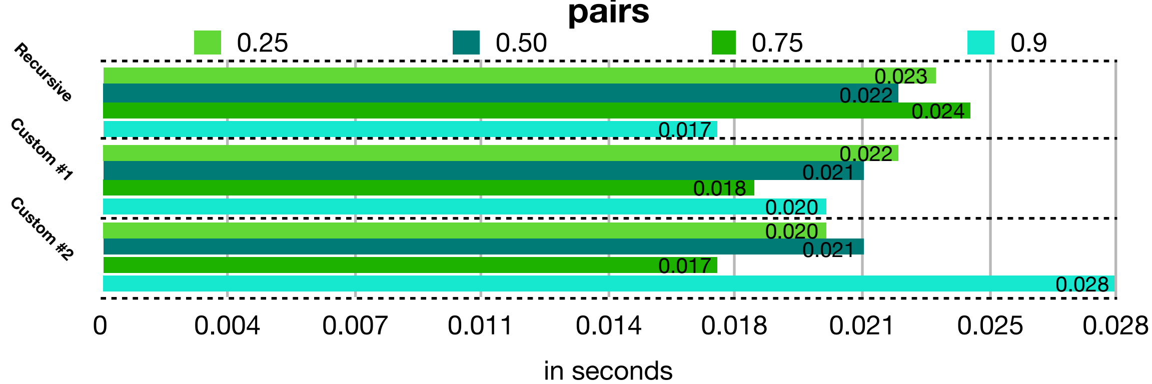


```
Similarity [forest,tree] = 67.838%
Similarity [keyboard,mouse] = 52.7055%
Similarity [branch,stick] = 28.97224%
Similarity [cow,moo] = 14.19595%
Similarity [cow,burger] = 34.5987%
Similarity [chicken,sandwich] = 81.52272%
Similarity [shirt,pants] = 85.74433%
Similarity [log,trunk] = 54.48938%
Similarity [hello,goodbye] = 85.37959%
Similarity [crown,head] = 45.59879%
Similarity [free,buy] = 57.00362%
Similarity [tax,money] = 80.00363%
Similarity [beautiful,pretty] = 71.95422%
Similarity [free,liberty] = 46.86742%
Similarity [chips,potato] = 49.91344%
Similarity [class,school] = 63.97571%
Similarity [tuna,salmon] = 80.25383%
Similarity [rice,beans] = 70.259%
Similarity [knife,fork] = 37.71736%
Similarity [lemon,lime] = 84.93173%
Similarity [bin,box] = 26.7474%
Similarity [book,words] = 66.16241%
```

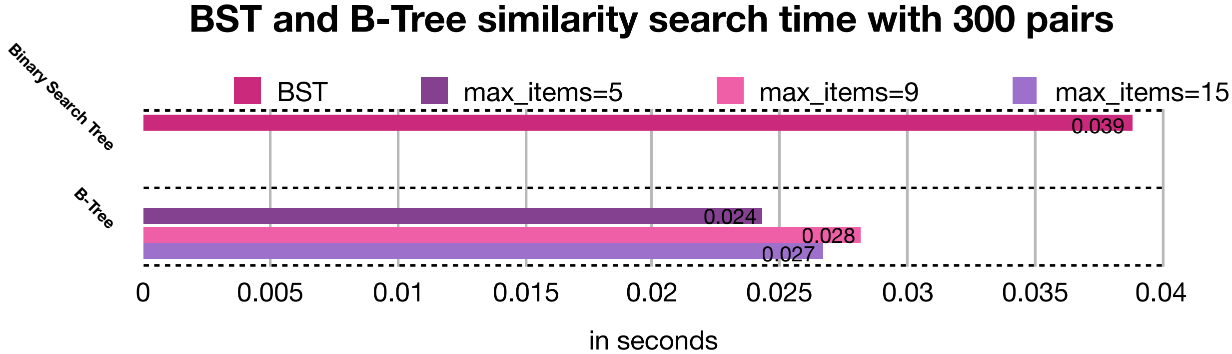
Hash Table with Chaining similarity search time with 300 pairs



Hash Table with Chaining similarity search time with 300 pairs



BST and B-Tree similarity search time with 300 pairs

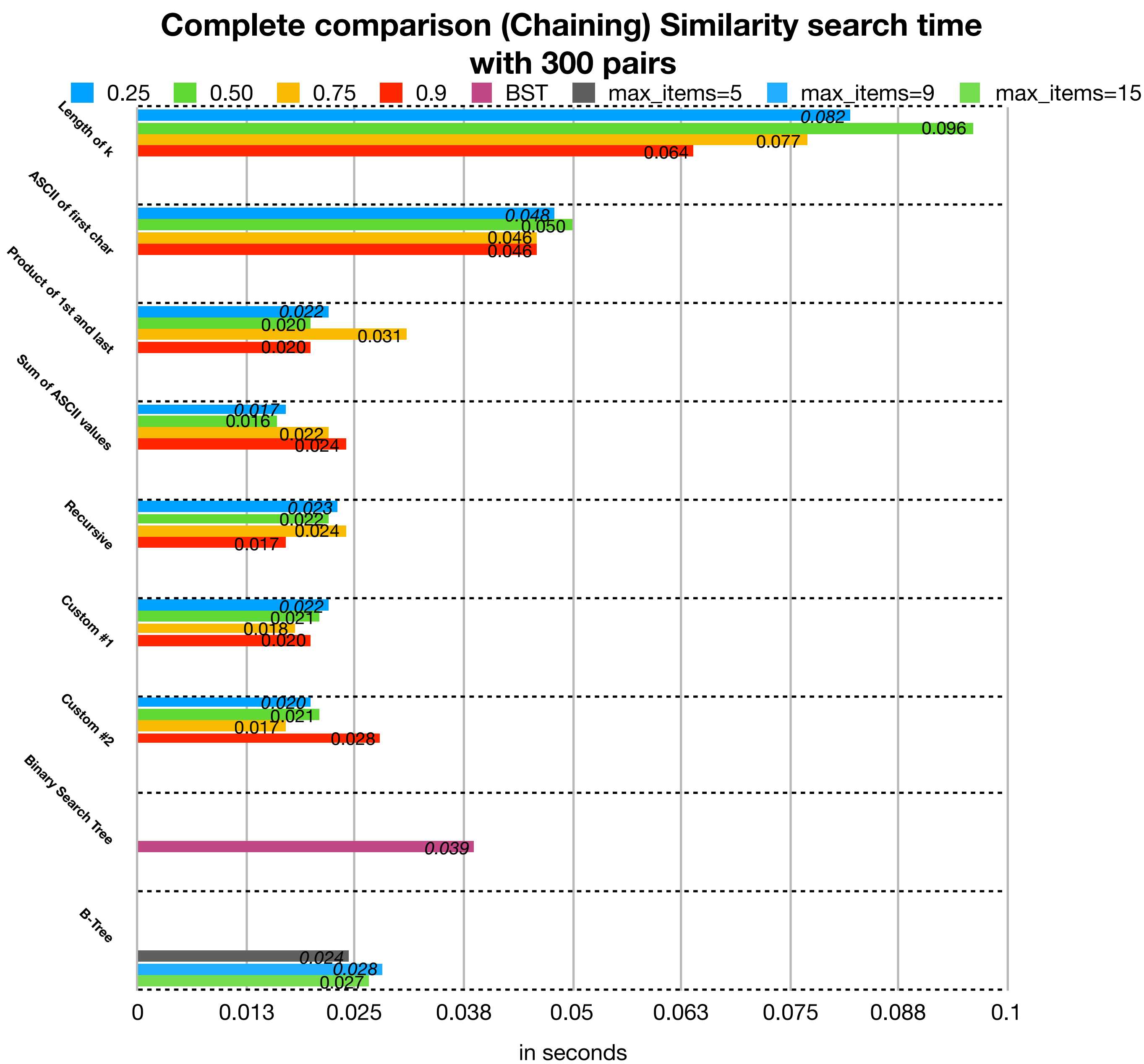


```
Running time for similarities: 0.0219

Hash Table with Chaining stats with choice 4
Running time for construction: 0.535644

Table size: 18810
Load factor: 0.750027
```

```
Running time for similarities: 0.0827

Hash Table with Chaining stats with choice 1
Running time for construction: 0.896795

Table size: 56432
Load factor: 0.25
```

```
Running time for similarities: 0.0199

Hash Table with Chaining stats with choice 3
Running time for construction: 0.521264

Table size: 15505
Load factor: 0.9099
```

```
Running time for similarities: 0.0217

Hash Table with Chaining stats with choice 6
Running time for construction: 0.654061

Table size: 28216
Load factor: 0.5
```



Conclusion:

This lab was a little more straightforward than the previous ones as we were given a list of the functions that needed to be implemented. Although I did run into a few problems with the Linear Probing part since the running times were much lengthier compared to the other implementations. I'm assuming due to the first 4 functions had more collisions than the rest. I decided to shorten the lines being read to a fraction of the entire file to get my experimental data for each hash function without spending a whole week running tests. As for the hash table with chaining, all the functions worked correctly and was comparable to running times from the Binary Search Tree and B-Trees.

Appendix:

```
"""
Laurence Justin Labayen
Lab 5
CS2302 Data Structures
MW 10:30
Professor: Olac Fuentes
TA: Anindita Nath
"""
import re
import time
import numpy as np

class WE_Node(object):
    def __init__(self,word,embedding):
        # word must be a string, embedding can be a list or and array of ints or floats
```

```
        self.word = word
        self.emb = np.array(embedding, dtype=np.float32) # For Lab 4, len(embedding=50)

class HashTableChain(object):
    # Builds a hash table of size 'size'
    # Item is a list of (initially empty) lists
    # Constructor
    def __init__(self,size):
        self.bucket = [[] for i in range(size)]

    # Hash function with length of string k % size of table
    def lenword_hash(self,k):
        if isinstance(k, WE_Node):
            k=k.word

        return len(k)%len(self.bucket)
    # Hash function with ASCII value of the first character of k % size of table
    def ascii_first_hash(self,k):
        if isinstance(k, WE_Node):
            k=k.word
        return ord(k[0])%len(self.bucket)

    # Hash function with product of ASCII values from first and last char % size of table
    def ascii_product_hash(self,k):
        if isinstance(k, WE_Node):
            k=k.word
        return (ord(k[0])*ord(k[-1]))%len(self.bucket)

    # Hash function with the sum of the ASCII values in k % size of table
    def ascii_sum_hash(self, k):
        if isinstance(k, WE_Node):
            k=k.word
        return sum(map(ord, k))%len(self.bucket)
```

```
# Recursive Hash function that multiplies the ASCII values of all the characters
# in k (plus 255 on each value) % size of table
def recursive_hash(self, k):
    if isinstance(k, WE_Node):
        k=k.word

    if len(k) == 0:
        return 1
    return (ord(k[0]) + 255 * self.recursive_hash(k[1:])) % len(self.bucket)

# Custom Hash function done with a loop to add all the ASCII
# values in k to the power of it's index % size of table
def custom_hash(self, k):
    if isinstance(k, WE_Node):
        k=k.word
    total=0
    for i in range(len(k)):
        total+=ord(k[i])**i

    return total % len(self.bucket)

# Custom recursive Hash function that uses the size of the table and int divides
# to the ASCII value of each character in k, then each is multiplied by the next
# character. Returns the product of all these values % of the size of the table
def custom_hash2(self, k):
    if isinstance(k, WE_Node):
        k=k.word
    if len(k) == 0:
        return 1
    return (len(self.bucket)//ord(k[0]) * self.custom_hash(k[1:])) % len(self.bucket)

# h function returns the selected hash function choice
```

```
def h(self,k,choice):
    if choice == 1:
        return self.lenword_hash(k)
    if choice == 2:
        return self.ascii_first_hash(k)
    if choice == 3:
        return self.ascii_product_hash(k)
    if choice == 4:
        return self.ascii_sum_hash(k)
    if choice == 5:
        return self.recursive_hash(k)
    if choice == 6:
        return self.custom_hash(k)
    if choice == 7:
        return self.custom_hash2(k)



def insert(self,k,choice):
    # Inserts k in appropriate bucket (list)
    # Does nothing if k is already in the table
    b = self.h(k,choice)
    if not k in self.bucket[b]:
        self.bucket[b].append(k)         #Insert new item at the end


def find_emb(self,k,choice):
    # Returns bucket (b) and index (i)
    # If k is not in table, i == -1
    b = self.h(k,choice)

    for j in self.bucket[b]:
        if j.word==k:
            return j.emb
    return
```

```
    def print_table(self):
        print('Table contents:')
        for b in self.bucket:
            for i in b:
                print(i.word)

    def load_factor(self):
        #number of elements/size
        num=0
        for i in self.bucket:
            num+=len(i)
        return num/len(self.bucket)

class HashTableLP(object):
    # Builds a hash table of size 'size', initilizes items to -1 (which means empty)
    # Constructor
    def __init__(self,size):
        self.item = np.zeros(size,dtype=np.object)-1

    def insert(self,k,choice):
        # initial position of k
        start = self.h(k.word,choice)
        for i in range(len(self.item)):
            # initial positon in the table to check
            pos = (start+i)%len(self.item)
            # check if current element is a WE_Node
            if isinstance(self.item[pos], WE_Node):
                # check if element to be inserted is already in the table
                if self.item[pos].word==k.word:
                    return -1
            # if it is not a WE_Node, check if it's less than 0
            elif self.item[pos] < 0:
```

```
            # insert k if current element is less than 0
            self.item[pos]=k
            return pos

    def find_emb(self,k,choice):
        # initial position of k
        if isinstance(k, WE_Node):
            k=k.word
        start=self.h(k,choice)
        for i in range(len(self.item)):
            # initial positon in the table to check
            pos = (start+i)%len(self.item)
            # if current element is in the table, return it's embedding
            try:
                if self.item[pos].word == k:
                    return self.item[pos].emb
            # if it throws an error, k is not in the table, return None
            except:
                if self.item[pos]<0:
                    return None

    # Hash function with length of string k % size of table
    def lenword_hash_LP(self,k):
        if isinstance(k, WE_Node):
            k=k.word
        return len(k)%len(self.item)

    # Hash function with ASCII value of the first character of k % size of table
    def ascii_first_hash_LP(self,k):
        if isinstance(k, WE_Node):
            k=k.word
        return ord(k[0])%len(self.item)
```

```
# Hash function with product of ASCII values from first and last char % size of table
def ascii_product_hash_LP(self,k):
    if isinstance(k, WE_Node):
        k=k.word
    return (ord(k[0])*ord(k[-1]))%len(self.item)

# Hash function with the sum of the ASCII values in k % size of table
def ascii_sum_hash_LP(self, k):
    if isinstance(k, WE_Node):
        k=k.word
    return sum(map(ord, k))%len(self.item)

# Recursive Hash function that multiplies the ASCII values of all the characters
# in k (plus 255 on each value) % size of table
def recursive_hash_LP(self, k):
    if isinstance(k, WE_Node):
        k=k.word

    if len(k) == 0:
        return 1
    return (ord(k[0]) + 255 * self.recursive_hash_LP(k[1:])) % len(self.item)

# Custom Hash function done with a loop to add all the ASCII
# values in k to the power of it's index % size of table
def custom_hash_LP(self, k):
    if isinstance(k, WE_Node):
        k=k.word
    total=0
    for i in range(len(k)):
        total+=ord(k[i])**i

    return total % len(self.item)
```

```
# Custom recursive Hash function that uses the size of the table and int divides
# to the ASCII value of each character in k, then each is multiplied by the next
# character. Returns the product of all these values % of the size of the table
def custom_hash2_LP(self, k):
    if isinstance(k, WE_Node):
        k=k.word
    if len(k) == 0:
        return 1
    return (len(self.item)//ord(k[0]) * self.custom_hash_LP(k[1:])) % len(self.item)

def h(self,k,choice):
    if choice == 1:
        return self.lenword_hash_LP(k)
    if choice == 2:
        return self.ascii_first_hash_LP(k)
    if choice == 3:
        return self.ascii_product_hash_LP(k)
    if choice == 4:
        return self.ascii_sum_hash_LP(k)
    if choice == 5:
        return self.recursive_hash_LP(k)
    if choice == 6:
        return self.custom_hash_LP(k)
    if choice == 7:
        return self.custom_hash2_LP(k)


def print_table(self):
    print('Table contents:')
    print(self.item)

def load_factor(self):
    #number of elements/size
```

```
        num=0
        for i in self.item:
            if isinstance(i, WE_Node):
                num+=1
        return num/len(self.item)



def menu():
    print('1. The length of the string % n')
    print('2. The ascii value (ord(c)) of the first character in the string % n')
    print('3. The product of the ascii values of the first and last characters in the string % n')
    print('4. The sum of the ascii values of the characters in the string % n')
    print('5. h('',n) = 1; h(S,n) = (ord(s[0]) + 255*h(s[1:],n))% n')
    print('6. Custom function #1')
    print('7. Custom function #2')

    choice = int(input('select hash function: '))

    print('\n1. Load Factor 0.25')
    print('2. Load Factor 0.50')
    print('3. Load Factor 0.75')
    print('4. Load Factor 0.90')

#    lf=int(input('Choose load factor: '))
#    if lf==1:
#        table_size=56432
#    if lf==2:
#        table_size=28216
#    if lf==3:
#        table_size=18810
#    if lf==4:
#        table_size=15505
```

```
# Load factor options for 14108 words with 6400000 lines from GLoVe file
    lf=int(input('Choose load factor: '))
    if lf==1:
        table_size=56432
    if lf==2:
        table_size=28216
    if lf==3:
        table_size=18810
    if lf==4:
        table_size=15505

    return choice, table_size

def HashChain_Test():

    choice, table_size = menu()

    H = HashTableChain(table_size)
    # Pattern to be used to remove words with unwanted characters
    pattern=re.compile("[A-Za-z]+")
    print('Loading glove file...')
    # Open glove file
    file = open('glove.6B.50d.txt','r')
    count=0
    # Start counter
    start = time.perf_counter()

    # readlines limited to a small sample of the GLoVe file to reduce times of certain
    # hash functions
    for line in file.readlines(6400000):
        row = line.strip().split(' ')
        # Check if word matches the pattern of characters
        if pattern.fullmatch(row[0]) is not None:
```

```
            # Insert into Hash Table with word and its embedding
            H.insert(WE_Node(row[0],[(i) for i in row[1:]]),choice)
            count+=1
    # Stop counter
    end = time.perf_counter()

    Similarity(H, choice, 'pairs.txt', 300, )

    print('\nHash Table with Chaining stats:')
    print('Running time for construction: '+ str(round((end - start), 6))+'\n')
#    print('Total words:',count)
    print('Table size:', table_size)
    print('Load factor:',round(H.load_factor(),6))

    return H

def HashTableLP_Test():

    choice, table_size = menu()

    H = HashTableLP(table_size)

    # Pattern to be used to remove words with unwanted characters
    pattern=re.compile("[A-Za-z]+")
    print('Loading glove file...')
    # Open glove file
    file = open('glove.6B.50d.txt','r')

    # Start counter
    start = time.perf_counter()
    count=0
    # readlines limited to a small sample of the GLoVe file to reduce times of certain
    # hash functions
```

```
    for line in file.readlines(6400000):
        row = line.strip().split(' ')
        # Check if word matches the pattern of characters
        if pattern.fullmatch(row[0]) is not None:
            # Insert into Hash Table with word and its embedding
            H.insert(WE_Node(row[0],[(i) for i in row[1:]]),choice)
            count+=1
    # Stop counter
    end = time.perf_counter()

    Similarity(H, choice, 'pairs.txt', 300 )

    print('\nHash Table with Linear Probing stats:')
    print('Running time for construction: '+ str(round((end - start), 6))+'\n')
#    print('Total words:',count)
    print('Table size:', table_size)
    print('Load factor:',round(H.load_factor(),6))

    # Similarity test for more words
#    yn = input('\nTest similarities again with random words? Y/N ')
#    if yn.lower() == 'y':
#        num_pairs=0
#        while num_pairs>=0:
#            num_pairs=int(input('Enter number of random pairs: '))
#            Similarity(H, choice, 'pairs_new.txt', num_pairs)
    return H

def Similarity(H,choice, file_choice, num_pairs):

#    numpairs = int(input('Enter # of pairs to compare: ') )

    # Open and read pairs word file and insert into a list as list of pairs
    pairs=[line.strip().split(' ') for line in open(file_choice,'r')]
```

```
# Assign timer to 0
timer=0

# Loop to iterate through list of pairs line by line
for i in range(num_pairs):
    # Start timer
    start = time.perf_counter()

    # word1 gets the first column in each line from pairs list
    word1=pairs[i][0]
    # word2 gets the second column in each line from pairs list
    word2=pairs[i][1]

    #word1emb and word2emb gets the embedding that is found by
    word1emb=(H.find_emb(word1,choice))
    word2emb=(H.find_emb(word2,choice))

    # Check if word1emb or word2emb is not found
    if word1emb is None or word2emb is None:
        continue

    # Formula to find cosine distance between both word embeddings
    cosine_distance = np.dot(word1emb, word2emb)/(np.linalg.norm(word1emb)* np.linalg.norm(word2emb))
    # Stop timer for every iteration
    end = time.perf_counter()

    # Add each timed iteration of finding similaties to "timer"
    timer += end - start

    print('Similarity ['+word1+','+word2+'] =',str(round(100*cosine_distance,5))+'%')

print('\n\nRunning time for similarities: '+ str(round(timer,4)))
```

```
if __name__=="__main__":


    select=int(input('Press 1 for Chaining and 2 for Linear Probing: '))
    if select == 1:
        H=HashChain_Test()
    if select == 2:
        H=HashTableLP_Test()
```

Academic Honesty Statement:

"I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class."

-Laurence Labayen