1.5em 0pt

Jared Shanklin
June 23, 2023

COSC 3320: Algorithm
Homework NUM: 2

1 Q1

Best case scenario we get two lists that are empty or only contain one element leaving the program to only run once making the lower bound $O(n)$

Algorithm 1 Function to check if two lists have a element in common

```
List A,B
for I := 1 → len(l) do
    if A[i] = B[i] then
        Break;
        Print(Lists have common element)
    else
        Print(Lists do not have common element)
    end if
end for
```

2 Q2

For this question I will use an array as my data structure. In order to make it better I will use a sorted array. This means that when inserting into the array the array will be sorted each time.

Algorithm 2 Insert element into Array

```
  Input Element to be added
  Output Array with newly added element
if n ≥ sizeof(A) then return n
end if
for i = n − 1 → i ≥ 0ANDA[i] > key do
    A[i+1] = A[i]
end for
A[i+1] = key
return n+1
```

Time complexity for insert is O(n). To insert an element within the array we are going to sort while adding. This means we compare the element to each element in the array. When comparing if the element in the array is less than the one being inserted we move the element to the left and continue searching. Once we find an element that is larger than the element being inserted we place the element there.
Below is how we will delete an element

The time complexity for deleting is also O(n). When deleting the array is already sorted so we must first search for the key that is being deleted. Once found we need to remove that element from the arrray and shift all other elements one position over.

The time complexity for finding n/5 is $O(n^2)$

Algorithm 3 Delete element from array

Input Array A, int n(elements), int key(to be deleted)
Output Array A without element key

$pos = binarySearch(A, 0, n-1, key)$
if pos = -1 **then**
print(Element not found) **return** n
end if
int i
for $i = pos \rightarrow i < n-1$ **do**
A[i] = A[i+1]
end for
return n-1

Algorithm 4 Find n/5

Input Array A
Output Medians for arrays created

for $i = 0 \rightarrow A.length$ **do**
A.slice(i, i + 5)
end for
if $n \mod 2! = 0$ **then return** A[n/2]
end if
return $A[(n-1)/2] + A[n/2]$

3 Q3

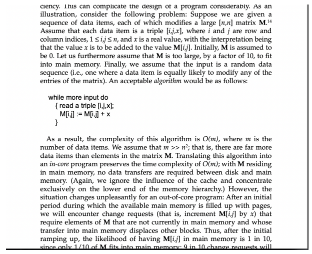
The Memorized Matrix Chain Algorithm is how we will find average work of a given sequence. When multiplying two matrices of P*q and q*r, we create a matrix of size p*r and the number of multiplications are shown by p*q*r. We want to set up an array A of type[1...n,1...n] and store in the element A[i,j]. We need to include the Lookup Chain algorithm in order to complete the algorithm, so we are going to assume that it is already completed. The lookup chain algorithm checks the conditions if the value of A[i,j] is less than infinity then returns previous cost of A[i,j]. If it is not less than, it computes the cost of A[i,j] and return the values. Average work in this question is referring to the optimal cost of the algorithm.

Algorithm 5 Memorized Matrix Chain

$n = length[x] - 1$
for $i = 1 \rightarrow n$ **do**
for $j = i \rightarrow n$ **do**
$A[i, j] =$
end forreturn LOOKUP$_C HAIN(x, 1, n)$**end for**
end forLOOKUP$_C HAIN(x, i, j)$ **if** A[i,j] ¡ **thenreturn**$A[i, j]$
end for
if $i = j$ **then**
$A[i, j] = 0$
end if$k = i \rightarrow j - 1$
$y = LOOKUP_C HAIN(x, i, k) + LOOKUP_C HAIN(x, k+1, j) + xi - xkxj$
if $y < A[i, j]$ **then**
A[i,j] = y
end ifreturn A[i,j]
end for

The time complexity for this algorithm is $O(n^3)$. Space complexity is $O(n^2)$

4 Q4

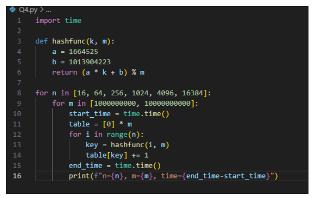


ciency. This can complicate the design of a program considerably. As an illustration, consider the following problem: Suppose we are given a sequence of data items, each of which modifies a large $[n,n]$ matrix **M**.[14] Assume that each data item is a triple $[i,j,x]$, where i and j are row and column indices, $1 \le i,j \le n$, and x is a real value, with the interpretation being that the value x is to be added to the value **M**$[i,j]$. Initially, **M** is assumed to be 0. Let us furthermore assume that **M** is too large, by a factor of 10, to fit into main memory. Finally, we assume that the input is a random data sequence (i.e., one where a data item is equally likely to modify any of the entries of the matrix). An acceptable *algorithm* would be as follows:

```
while more input do
   { read a triple [i,j,x];
     M[i,j] := M[i,j] + x
   }
```

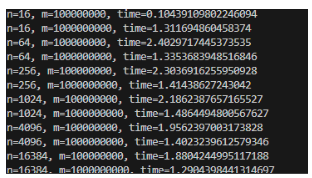
As a result, the complexity of this algorithm is $O(m)$, where m is the number of data items. We assume that $m >> n^2$; that is, there are far more data items than elements in the matrix **M**. Translating this algorithm into an *in-core* program preserves the time complexity of $O(m)$; with **M** residing in main memory, no data transfers are required between disk and main memory. (Again, we ignore the influence of the cache and concentrate exclusively on the lower end of the memory hierarchy.) However, the situation changes unpleasantly for an out-of-core program: After an initial period during which the available main memory is filled up with pages, we will encounter change requests (that is, increment **M**$[i,j]$ by x) that require elements of **M** that are not currently in main memory and whose transfer into main memory displaces other blocks. Thus, after the initial ramping up, the likelihood of having **M**$[i,j]$ in main memory is 1 in 10, since only 1/10 of **M** fits into main memory; 9 in 10 change requests will

Figure 1: P.126 Algorithm



```python
import time

def hashfunc(k, m):
    a = 1664525
    b = 1013904223
    return (a * k + b) % m

for n in [16, 64, 256, 1024, 4096, 16384]:
    for m in [1000000000, 10000000000]:
        start_time = time.time()
        table = [0] * m
        for i in range(n):
            key = hashfunc(i, m)
            table[key] += 1
        end_time = time.time()
        print(f"n={n}, m={m}, time={end_time-start_time}")
```

Figure 2: Algorithm implemented in Python



```
n=16, m=100000000, time=0.10439109802246094
n=16, m=1000000000, time=1.311694860458374
n=64, m=100000000, time=2.4029717445373535
n=64, m=1000000000, time=1.3353683948516846
n=256, m=100000000, time=2.3036916255950928
n=256, m=1000000000, time=1.41438627243042
n=1024, m=100000000, time=2.1862387657165527
n=1024, m=1000000000, time=1.4864494800567627
n=4096, m=100000000, time=1.9562397003173828
n=4096, m=1000000000, time=1.4023239612579346
n=16384, m=100000000, time=1.8804244995117188
n=16384, m=1000000000, time=1.2904398441314697
```

Figure 3: Results from running the program

In this picture you see results for 100 000 000 and 1 000 000 000 because the numbers that were provided resulted in a memory error. I tried to split the for loop into 3 loops using batched looping but due to the algorithm I could not find a way to do it. I will continue to try of course but for the sake of the assignment I will leave it as is for now. When looking at the times, we see that when m = 100 000 000 the time decreases as n increases where as when m = 1 000 000 000 the time increases while n increases.

5 Q5

For this question we must implement three If statements to handle the value of M. Once we have the value and the size of M we are going to create an Avl tree where each node consists of these two values. We are going to use random values from 0 to 299 to insert as the value. While creating this Avl tree we must keep in mind that it can never exceed 50 nodes, so we must remove the least recent node each time once we are at 50 nodes. We want to see the time each insertion and deletion takes.

```python
myTree = AVL_Tree()
random_num = np.random.randint(300)
M = 0
if random_num == 0%3:
    M = 2^20
elif random_num == 1%3:
    M = 2^19 + 2^18
elif random_num == 2%3:
    M = 2^18+2^17
root = None
nums = np.random.randint(low=1, high=299, size = 10)
nums = (nums + M).astype(int)
for num in nums:
    root = myTree.insert(root, num)
```

Figure 4: Code to find M using the value

```python
def insert(self, root, key):

    if not root:
        return TreeNode(key)
    elif key < root.val:
        root.left = self.insert(root.left, key)
    else:
        root.right = self.insert(root.right, key)


    root.height = 1 + max(self.getHeight(root.left),
                          self.getHeight(root.right))

    balance = self.getBalance(root)


    if balance > 1 and key < root.left.val:
        return self.rightRotate(root)

    if balance < -1 and key > root.right.val:
        return self.leftRotate(root)

    if balance > 1 and key > root.left.val:
        root.left = self.leftRotate(root.left)
        return self.rightRotate(root)

    if balance < -1 and key < root.right.val:
        root.right = self.rightRotate(root.right)
        return self.leftRotate(root)

    return root
```

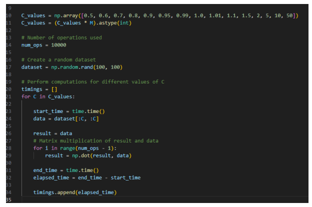
Figure 5: AVL Insertion

```python
def delete(self, root, key):

    if not root:
        return root

    elif key < root.val:
        root.left = self.delete(root.left, key)

    elif key > root.val:
        root.right = self.delete(root.right, key)

    else:
        if root.left is None:
            temp = root.right
            root = None
            return temp

        elif root.right is None:
            temp = root.left
            root = None
            return temp

        temp = self.getMinValueNode(root.right)
        root.val = temp.val
        root.right = self.delete(root.right,
                                 temp.val)

    if root is None:
        return root

    root.height = 1 + max(self.getHeight(root.left),
                          self.getHeight(root.right))

    balance = self.getBalance(root)

    if balance > 1 and self.getBalance(root.left) >= 0:
        return self.rightRotate(root)

    if balance < -1 and self.getBalance(root.right) <= 0:
        return self.leftRotate(root)

    if balance > 1 and self.getBalance(root.left) < 0:
        root.left = self.leftRotate(root.left)
        return self.rightRotate(root)

    if balance < -1 and self.getBalance(root.right) > 0:
        root.right = self.rightRotate(root.right)
        return self.leftRotate(root)

    return root
```

Figure 6: AVL Deletion

6 Q6

In order to design this program I am going to assume we are using LRU algorithm as discussed before in class and that each page size is 4KB. In this program we must load a data set of C into the cache and carry out a number of operations that is severl orders larger than C. When looking at the plot we see a pretty constant linear increase in



```python
C_values = np.array([0.5, 0.6, 0.7, 0.8, 0.9, 0.95, 0.99, 1.0, 1.01, 1.1, 1.5, 2, 5, 10, 50])
C_values = (C_values * M).astype(int)

# Number of operations used
num_ops = 10000

# Create a random dataset
dataset = np.random.rand(100, 100)

# Perform computations for different values of C
timings = []
for C in C_values:

    start_time = time.time()
    data = dataset[:C, :C]

    result = data
    # Matrix multiplication of result and data
    for i in range(num_ops - 1):
        result = np.dot(result, data)

    end_time = time.time()
    elapsed_time = end_time - start_time

    timings.append(elapsed_time)
```

Figure 7: python implementation of program

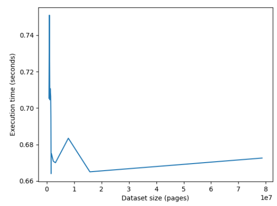


Figure 8: Plot of the timings for Matrix Multiplication shown in Figure 7

timing once we have a dataset size of 1.5 pages. It begins at .67 seconds and gradually increases to .68 seconds but never reaches it so overall the program runs ¡.68 seconds. However, before we get to 1.5 pages the timings are very scrambled. Some of the first multiplication operations range from .67 seconds to as high as .75 seconds.