Theory Assignment 1

Chenting Zhang

September 23, 2023

1 Problem 1

```
1  import numpy as np
2  def sum(vector_a, vector_b):
3      sum = 0
4      for i in range(vector_a.shape[0]):  # n iteration
5          sum = sum + vector_a[i]*vector_b[i]
6      return sum
7
8  def matrixMatrixProduct(A,B):
9      B_transpose = B.T
10     if A.shape[1] == B.shape[0]:
11         C = np.zeros((A.shape[0], B.shape[1]))
12         for i in range(A.shape[0]):   # m iteration
13             for j in range(B.shape[1]):   # k iteration
14                 C[i][j] = sum(A[i], B_transpose[j])
15     else: print("invalid input array shape")
16     return C
```

The test code is illustrated as follows:

```
1  ## test
2  A = np.array([[1,2],[4,5],[1,3]])   # 3*2
3  B = np.array([[1,2,3,1],[2,2,2,2]])     # 2*4
4  C = matrixMatrixProduct(A,B)   # 3*4
5  print(C)
```

For the time complexity, the running time is $O(m \times k \times n)$, the reason is illustrated as follows: For the transpose operation in line 9, the running time is $O(n \times k)$. Then the initialization of matrix C is shaped as $m \times k$, thus it takes $O(m \times k)$. For the nested loop part, it goes through m times iteration for the outer loop and k times iteration for the inner loop. In each operation in line 14, it calls the *sum* function which goes through n times iteration to compute the exact element in the position $C[i][j]$. Overall, the time complexity is $O(n \times k + m \times k + m \times n \times k) = O(m \times k \times n)$

2 Problem 2

The time complexity is $O(n)$. In the computeMean function, there is a for loop, which means it is $O(n)$, and to compute the standard deviation, there is another for loop. So the overall time complexity is $O(2n) = O(n)$

3 Problem 3

```
1      def binarySearchReverseOrder(L, x, left = None, right = None):
2      left = 0
3      right = len(L) - 1
4      while left <= right:
5          mid = (left + right) // 2
6          if L[mid] == x:
```

```
            return mid
        elif L[mid] < x:
            right = mid - 1
        else:
            left = mid + 1
    return -1
```

For every search, it cuts half of the list to search, so the time complexity is $O(logn)$. That is the point of binary search. Instead of comparing the element one by one. For sorted list, it is more efficient.

4 Problem 4

```
1  def findindex(L):
2      left = 0
3      right = len(L) - 1  ## 6
4      index = 0
5      while left <= right:
6          mid = (left + right) // 2
7          if L[mid] == 0:
8              if L[mid + 1] == 0:
9                  left = mid + 1
10             else:
11                 index = mid
12                 break
13
14         else:
15             if L[mid - 1] == 1:
16                 right = mid - 1
17             else:
18                 index = mid - 1
19                 break
20     return index
21
22 L = [0, 0, 0, 0, 0, 1, 1]  ## index j = 4
23 index = findindex(L)
24 print(index)
```

Binary Search is used in this algorithm, Line 2,3,4 are constant, in the while loop, we compute the index of the middle element and compare with the neighboring element. In the worst case where entry is at the very beginning or the very end, the time complexity is $O(log_2 n)$.

5 Problem 5

5.1

The running time is $O(n^3)$. The iterative times is dependent on i, while i is increasing from 0 to n-1, thus the inner loops computes $(0+1+2+\cdots(n-1)) = ((n-1)n)/2$ times. The outer loop computes n times, so overall the time complexity is $O(n \times (n-1) \times n/2) = O(n^3)$

5.2

```
def computeResult(myList):
    n = len(myList)
    result = n//2
    return result
```

This is the last simplified version. As we could infer from the original algorithm, it iterates through all the indices of myList, and check if it is even or odd. When the index is even(except for index 0), result will increment 1, when the index is odd, the result remains. Thus, we could deduce that the

result is associated with the number of even indices in the list, meaning that it depends on the size of the list and halve them and get the floor value. Since there is no for loop, the time complexity is $O(1)$.