中山大学计算机学院人工智能本科生实验报告

课程：Artificial Intelligence
姓名： 学号：

一、实验题目

1.二分查找
2.矩阵加乘法
3.字典遍历

二、实验内容

1.算法原理

(1)二分查找

对于有序数组，以升序为例，只需设置两个指针以指定搜索区间上下限，比较区间中点元素与指定值即可。如果比指定值小，说明指定值应在后半区间，此时更新前指针；如果比指定值大，说明指定值应在前半区间，此时更新后指针；如果相等，则说明查找成功。

由于每次都将区间长度折半，因此算法效率很高。

(2)矩阵加乘法

只需按照数学原理实现即可。对于方阵A, B，经过运算后产生新方阵C，那么对于矩阵加法$A + B = C$，显然$C_{ij} = A_{ij} + B_{ij}$。

对于矩阵乘法$A \times B = C$，将方阵重写为向量组形式：

$$A = [a_1^T, a_2^T, ..., a_n^T]$$

$$B = [b_1, b_2, ..., b_n]$$

那么$C_{ij} = a_1^T b_1$，也就是新方阵的i行j列元素是A方阵i行向量与B方针j列向量的数乘。

(3)字典遍历

利用 `dict.items()` 函数生成键值对元组之列表，对其从前往后遍历、在新字典中添加新键值对即可。

2.伪代码

(1)二分查找

```
接受一个升序list, 一个指定值:
    设定前后指针，初始指向最前最后元素
    while (前指针 <= 后指针):
        中指针 = (前指针 + 后指针) // 2
        判断 中指针.值 与 指定值 的大小
        小于: 前指针 = 中指针 + 1
        大于: 后指针 = 中指针 - 1
        等于: 返回中指针
    退出循环未找到，返回-1
```

(2)矩阵加乘法

加法:

```
接受两个list[list]:
    遍历行：
        遍历列: 新方阵此行列元素为两方阵此行列元素之和
    返回新方阵
```

乘法:

```
接受两个list[list]:
    遍历行：
        遍历列: 新方阵此行列元素为第一方阵此行向量与第二方针此列向量之数乘
    返回新方阵
```

(3)字典遍历

```
接受一个dict:
    遍历此字典键值对：新字典添加键值对为当前值键对
    返回新字典
```

3.关键代码展示

(1)二分查找

```python
def BinarySearch(nums: list, target: int):
    pre, back = 0, len(nums) - 1 #双指针法，首先将前后指针分别指向最前和最后元素
    while (pre <= back): #在不越界的情况下
        mid = (pre + back) // 2 #访问区间中间元素
        if (nums[mid] < target): pre = mid + 1 #如果比期望小，说明期望在后半区间
        elif (nums[mid] > target): back = mid - 1 #如果比期望大，说明期望在前半区间
        else: return mid #恰好等于，直接返回
    return -1 #越界，未找到
```

(2)矩阵加乘法

加法:

```python
def MatrixAdd(A: list[list], B: list[list]):
    returner = [] #创建返回对象
    for r in range(0, len(A)):
        temp = [] #创建临时对象
        for c in range(0, len(A[r])): temp.append(A[r][c] + B[r][c]) #新矩阵[i][j]的元素是第一第二
        returner.append(temp)
    return returner
```

乘法:

```python
#这里多设计了两个函数：向量数乘VectorMul和矩阵转置MatrixTrans
#这是为了从数学层面更加清晰体现矩阵乘法的意义，即：
#新矩阵的r行c列元素是第一矩阵r行向量与第二矩阵c列向量的数乘
#由于多了矩阵转置的操作，相比起直接按照对原矩阵计算的写法，必然导致时间和空间性能损失
#但考虑到Cache的存在，将第二矩阵列向量换为行向量，可能能提高缓存命中率

import copy

def VectorMul(A: list, B: list):
    sum = 0
    for i in range(0, len(A)): sum = sum + A[i] * B[i] #sum自增两个向量对应坐标值之积
    return sum

def MatrixTrans(A: list[list]):
    returner = copy.deepcopy(A) #创建返回对象，初始化为原矩阵之深复制
    for r in range(0, len(A)):
        for c in range(0, len(A[r])): returner[c][r] = A[r][c] #新矩阵的c行r列元素是原矩阵r行c列元
    return returner

def MatrixMul(A: list[list], B: list[list]):
    returner = []
    BTrans = MatrixTrans(B)
    for r in range(0, len(A)):
        temp = []
        for c in range(0, len(A[r])): temp.append(VectorMul(A[r], BTrans[c])) #新矩阵[i][j]的元素
        returner.append(temp)
    return returner
```

(3)字典遍历

```python
def ReverseKeyValue(diction: dict):
    returner = {}
    for key, value in diction.items(): returner[value] = key #从前往后遍历交换键和值
    return returner
```

4.创新优化

本实验的创新之处在于，在矩阵的乘法中单独抽出矩阵转置与向量数乘的操作，是对数学原理的忠实体现。

在优化方面，虽然没有对算法优化，但考虑了Cache的组织形式，以提高缓存命中率为目的，使这套函数在大型矩阵情况下可能有更高的效率。

第一，对于矩阵转置函数 `MatrixTrans` ，最内层的操作是 `returner[c][r] = A[r][c]` 而不是 `returner[r][c] = A[c][r]` ，尽管两者在交换数值的效果上是等价的，但是Python作为行优先存储语言，前一种写法每次都是取内存中临近的数。

第二，在矩阵乘法函数 `MatrixMul` 中，由于直接换为了等价的行向量，对元素的访问也是在内存中相邻的。

三、实验结果分析

1.实验结果展示

给出如下测试用程序段：

```
nums = [1,3,5,7,9,11,33,44,55,66,77,88] #二分查找用例
matrix1 = [[1,2,3],[2,3,4],[3,4,5]] #矩阵加乘用例
matrix2 = [[1,2,3],[4,5,6],[7,8,9]] #矩阵加乘用例
diction = {1:'one', 2:'two', 3:'three', 4:'four'} #字典遍历用例

print("在nums列表中寻找数值0~11:")
for i in range(0, 12): print(i, BinarySearch(nums, i)) #二分查找验证
print("矩阵加法与乘法结果:")
Answer1 = MatrixAdd(matrix1, matrix2)
Answer2 = MatrixMul(matrix1, matrix2)
print(Answer1) #矩阵加法验证
print(Answer2) #矩阵乘法验证
print("原字典与字典遍历结果:")
Answer3 = ReverseKeyValue(diction)
print(diction) #原字典
print(Answer3) #字典遍历验证
```

运行结果如下：

```
在nums列表中寻找数值0~11:
0 -1
1 0
2 -1
3 1
4 -1
5 2
6 -1
7 3
8 -1
9 4
10 -1
11 5
矩阵加法与乘法结果:
[[2, 4, 6], [6, 8, 10], [10, 12, 14]]
[[30, 36, 42], [42, 51, 60], [54, 66, 78]]
原字典与字典遍历结果:
{1: 'one', 2: 'two', 3: 'three', 4: 'four'}
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

2.评测指标展示分析

(1)二分查找

二分查找就是简单地构建了一棵二分查找树，即使是最坏情况下也最多只需$log_2 n$次查找，时间复杂度为$O(log_2 n)$。

(2)矩阵加乘

按照 `list[list]` 构建的矩阵，矩阵加法必须要遍历所有元素，在算法上没有可以优化的空间，时间复杂度是$O(n^3)$。但由于各元素间并不相干，为了缩短运行时间可以考虑使用多线程。

对于矩阵乘法，矩阵转置的复杂度是$O(n^2)$，向量数乘复杂度为$O(n)$，乘法复杂度为$O(n^3)$。但考虑$Cache$，大型矩阵情况下此函数可能有更高效率。

(3)字典遍历

字典遍历就是从前往后遍历，显然时间复杂度为$O(n)$。

四、思考题

无。

五、参考资料

无。