Algorithm First Homework

Zhang Yuan

Student ID: 2015E8013261189

1 Find Median

1.1 The Method Solving Problem

Assume A and B are two given n-sized databases, A[i] and B[i] represent the ith smallest value of A and B. We can get the nth smallest value through divide and conquer method. We cut some items in lists by comparing two medians in A and B to reduce the problem.

```
Procedure Find_Median(A,B):
    n=Len(A)+Len(B)
    return fms(A,B,n/2)

Procedure fms(A,B,k):
    LA=Len(A)
    LB=Len(B)
    if LA>LB then return fms(B,A,k)
    else
        if LA equals to 0 then return query(B,k-1)
        if k equals to 1 then return min(query(A,0),query(B,0))
        pa=min(k/2,LA)
        pb=k-pa
        if query(A,pa-1) <= query(B, pb-1)
        then return fms(A[pa:],B,k-pa)
        else return fms(A,B[pb:],k-pb)
```

1.2 Reduction Graph

T(n) $\cdots O(1)$
└ $T(\frac{n}{2}) \cdots O(1)$
  └ $\cdots$

1.3 Prove The Correctness

Assume we split L into A and B, if A[$\frac{n}{2}$] is less than B[$\frac{n}{2}$], we can infer the fact that the nth smallest value can not exit in A[:$\frac{n}{2}$], so now, the nth smallest value must exits in A[$\frac{n}{2}$:] and B as the $\frac{n}{2}$th value. The confirm code shows below:

```
#coding=utf-8
def Load_Data():
    A=[1,3,6,10,14]
    B=[2,5,7,8,9]
    return A, B


def findMedianSortedArrays(A, B):
    if (len(A) + len(B)) % 2 == 0:
        return (fms(A, B, (len(A) + len(B))/2) + fms(A, B, (len(A) + len(B
    else:
        return fms(A, B, (len(A) + len(B))/2 + 1)

def fms(A, B, k):
    if len(A) > len(B):
        return fms(B, A, k)
    else:
        if len(A) == 0:
            return B[k-1]
        if k == 1:
            return min(A[0], B[0])
        pa = min(k/2, len(A))
        pb = k - pa
        if A[pa-1] <= B[pb-1]:
            return fms(A[pa:], B, k-pa)
        else:
            return fms(A, B[pb:], k-pb)

A,B=Load_Data()
print findMedianSortedArrays(A,B)
```

1.4 Time Complexity

$$T(n) = T(n/2) + O(1)$$

Using Master Theorem:

$$T(n) = \Theta(\log n)$$

2 Find Inversions

2.1 The Method Solving Problem

We can use the same idea from the numbers of inversions and merge sort to solve the problem. Assume we split L into A and B, we multiply B by 3 to B', and find the numbers of inversions between A and B', then still merge A and B to a sorted array.

```
Procedure Sort_And_Count(L):
    n=Len(L)
```

```
    if n=1 then return 0, L
    else
        k=n/2
        A,B=split_at(L,k)
        (C1, A)=Sort_And_Count(A)
        (C2, B)=Sort_And_Count(B)
        (C3, L)=Merge_And_Count(A,B)
        return C=C1+C2+C3, L

Procedure Merge_And_Count(L,R):
    inversion_Count=0, i=0, j=0
    Let R_3 be 3*R
    for k=0 to ||L||+||R_3||-1 do
        if L[i]>R_3[j] then
            j++
            inversion_Count+=Len(L)-i
        else:
            i++

    for k=0 to ||L||+||R||-1 do
        if L[i]>R[j] then
            T[k]=R[j]
            j++
        else:
            T[k]=A[i]
            i++
    return inversion_Count, T
```

2.2 Reduction Graph

T(n) $\cdots O(n)$
- $T(\frac{n}{2}) \cdots O(\frac{n}{2})$
  - $\cdots$
- $T(\frac{n}{2}) \cdots O(\frac{n}{2})$
  - $\cdots$

2.3 Prove The Correctness

Assume we split L into two sorted array, A and B. To avoid the sensitivity, the case that i <j, A[i] >3*B[j] become the inversion, so we can multiply B by 3 to B', then the problem turns to be the number of inversions between A and B', after counting the number, we should still merge A and B, instead of A and B'. The confirm code shows below:

```
#coding=utf-8
def load_data():
    A=[28,24,4,3,12,7,5,1]
    return A
```

```
def Sort_And_Count(A):
    if len(A)<=1:
        return 0, A
    k=len(A)/2
    L=A[:k]
    R=A[k:]
    Count_L, L=Sort_And_Count(L)
    Count_R, R=Sort_And_Count(R)
    Count, A=Sense_Merge_And_Count(L,R)
    return Count_L+Count_R+Count, A

def Merge_And_Count(L,R):
    RC=0
    i=0
    j=0
    A=[]
    len_L=len(L)
    len_R=len(R)

    while len(L)>0 and len(R)>0:
        if L[0]>R[0]:
            A.append(R.pop(0))
            j+=1
            RC+=len_L-i
        else:
            A.append(L.pop(0))
            i+=1

    if len(L)>0:
        A.extend(L)
    else:
        A.extend(R)

    return RC,A

def Sense_Merge_And_Count(L,R):
    RC=0
    i=0
    j=0
    A=[]
    len_L=len(L)
    len_R=len(R)

    sub_L=[]
    for i in range(len(L)):
        key=L[i]
        sub_L.append(key)
    sub_R=[]
```

```
    for i in range(len(R)):
        key=R[i]
        sub_R.append(3*key)
    sub_A=[]
    sub_i=0
    sub_j=0
    sub_RC=0

    while len(sub_L)>0 and len(sub_R)>0:
        if sub_L[0]>sub_R[0]:
            sub_A.append(sub_R.pop(0))
            sub_j+=1
            sub_RC+=len_L-sub_i
        else:
            sub_A.append(sub_L.pop(0))
            sub_i+=1
    print  sub_RC

    while len(L)>0 and len(R)>0:
        if L[0]>R[0]:
            A.append(R.pop(0))
            j+=1
        else:
            A.append(L.pop(0))
            i+=1

    if len(L)>0:
        A.extend(L)

    if len(R)>0:
        A.extend(R)

    return sub_RC, A



A=load_data()
print Sort_And_Count(A)
```

2.4 Time Complexity

$$T(n) = 2 * T(\frac{n}{2}) + O(n)$$

Using Master Theorem:

$$T(n) = \Theta(n \log n)$$

3 Find The Local Minimum

3.1 The Method Solving Problem

To find the local minimum, we can traverse the tree, starting with the root as the current vertex:
1.Probe the current vertex's label, and the labels of its two children.
2.If the current label is the smallest, halt and report that the current vertex is a local minimum.
3.Else set the current vertex to the child with the smallest label, and return to step 1.

```
Procedure Find_Local_Min(T):
    if T has children, then
        let L, R be T's left child, right child
        probe XL, XR, XT
            if XL < XT and XL < XR then return Find_Local_Min(L)
            else if XR < XT and XR < XL then return Find_Local_Min(R)
            else return T
    else return T
```

3.2 Reduction Graph

T(n+1) $\cdots O(1)$
└─ $T(\frac{n+1}{2}) \cdots O(1)$
  └─ $\cdots$

3.3 Prove The Correctness

Since the tree is a complete binary tree, each vertex has at most three neighbors, its parent and two children (the root has no parent), so a vertex is a local minimum if its label is less than the labels of its two children and parent. Hence we can traverse the node at most three probes.

3.4 Time Complexity

$$n = 2^d - 1$$
$$d = \log(n+1)$$
$$T(n) = 3 * \log(n+1)$$

4 Strassen Algorithm

4.1 The Method Solving Problem

The traditional matrix multiplication always costs n^3 arithmetic operations including additions and multiplications, we can reduce numbers of additions and multiplications using Strassen Algorithm.

```
Procedure Strassen(A,B):
    if n=1 then return A*B
    else:
        Compute A11, B11, ..., A22, B22
```

```
        P1=Strassen(A11,B12-B22)
        P2=Strassen(A11+A12,B22)
        P3=Strassen(A21+A22,B11)
        P4=Strassen(A22,B21-B11)
        P5=Strassen(A11+A22,B11+B22)
        P6=Strassen(A12-A22,B21+B22)
        P7=Strassen(A11-A21,B11+B12)
        C11=P5+P4-P2+P6
        C12=P1+P2
        C21=P3+P4
        C22=P1+P5-P3-P7
        return C
```

4.2 The Strassen algorithm for Matrix Multiplication in Python

```
#coding=utf-8

def load_data():
    m=[[1,2,3,4],[4,5,6,7],[7,8,9,10],[1,2,3,4]]
    n=[[3,2,1,4],[6,5,4,7],[9,8,7,10],[1,2,3,6]]
    return m,n

def init_matrix(size, value=0):
    """generate a matrix of given size.
    """
    return [[value for i in xrange(size)] for j in xrange(size)]


def add(mat1,mat2):
    n=len(mat1)
    new_mat=init_matrix(n)
    for i in xrange(n):
        for j in xrange(n):
            new_mat[i][j]=mat1[i][j]+mat2[i][j]

    return new_mat

def sub(mat1,mat2):
    n=len(mat1)
    new_mat=init_matrix(n)
    for i in xrange(n):
        for j in xrange(n):
            new_mat[i][j]=mat1[i][j]-mat2[i][j]
    return new_mat


def primitive_mul(mat1,mat2):
    """The original O(n^3) matrix multiplication"""
```

```
    n=len(mat1)
    new_mat=init_matrix(n)
    for i in xrange(n):
        for j in xrange(n):
            for k in range(n):
                new_mat[i][j]+=mat1[i][k]*mat2[k][j]
    return new_mat

def strassen(mat1,mat2,leaf_size=2):
    """Strassen Matrix Multiplication Algorithm"""
    n=len(mat1)
    if n<=leaf_size:
        return primitive_mul(mat1,mat2)
    else:
        size=n/2
    #initialize sub matrices
        a11,a12,a21,a22,b11,b12,b21,b22=[init_matrix(size) for i in range(8
        for i in xrange(size):
            for j in range(size):
                a11[i][j]=mat1[i][j]
                a12[i][j]=mat1[i][j+size]
                a21[i][j]=mat1[i+size][j]
                a22[i][j]=mat1[i+size][j+size]

                b11[i][j]=mat2[i][j]
                b12[i][j]=mat2[i][j+size]
                b21[i][j]=mat2[i+size][j]
                b22[i][j]=mat2[i+size][j+size]


        p1=strassen(add(a11,a22),add(b11,b22))
        p2=strassen(add(a21,a22),b11)
        p3=strassen(a11,sub(b12,b22))
        p4=strassen(a22,sub(b21,b11))
        p5=strassen(add(a11,a12),b22)
        p6=strassen(sub(a21,a11),add(b11,b12))
        p7=strassen(sub(a12,a22),add(b21,b22))

        c11=add(sub(add(p1,p4),p5),p7)
        c12=add(p3,p5)
        c21=add(p2,p4)
        c22=add(add(sub(p1,p2),p3),p6)

        mat_c=init_matrix(n)
        for i in xrange(size):
            for j in xrange(size):
                mat_c[i][j]=c11[i][j]
                mat_c[i][j+size]=c12[i][j]
```

```
                mat_c[i+size][j]=c21[i][j]
                mat_c[i+size][j+size]=c22[i][j]

        return mat_c

mat1,mat2=load_data()
mat3=add(mat1,mat2)
print mat3
mat3=sub(mat1,mat2)
print mat3
mat3=primitive_mul(mat1,mat2)
print mat3
mat3=strassen(mat1,mat2)
print mat3
```

4.3 Prove The Correctness

$$C^{11} = P_5 + P_4 - P_2 + P_6$$
$$C^{12} = P_1 + P_2$$
$$C^{21} = P_3 + P_4$$
$$C^{22} = P_1 + P_5 - P_3 - P_7$$

$$P_1 = A^{11} * (B^{12} - B^{22})$$
$$P_2 = (A^{11} + A^{12}) * B^{22}$$
$$P_3 = (A^{21} + A^{22}) * B^{11}$$
$$P_4 = A^{22} * (B^{21} - B^{11})$$
$$P_5 = (A^{11} + A^{22}) * (B^{11} + B^{22})$$
$$P_6 = (A^{12} - A^{22}) * (B^{21} + B^{22})$$
$$P_7 = (A^{11} - A^{21}) * (B^{11} + B^{12})$$

4.4 Time Complexity

The combing cost and add/sub costs $O(n^2)$:

$$T(n) = 7 * T(\frac{n}{2}) + O(n^2)$$

Using Master Theorem:

$$T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.8})$$

4.5 The Performance

After testing 16*16, 32*32, 64*64, 128*128, 256*256, and 512*512 matrices, and the results shown below:

16*16:
Tradition ways: 0.00117611885071s
Strassen ways: 0.00817203521729s

32*32:
Tradition ways: 0.00691509246826s
Strassen ways: 0.0587410926819s

64*64:
Tradition ways: 0.0562977790833s
Strassen ways: 0.416481018066s

128*128:
Tradition ways: 0.443516016006s
Strassen ways: 2.97132205963s

256*256:
Tradition ways: 3.64289307594s
Strassen ways: 20.584084034s

512*512:
Tradition ways: 34.4611940384s
Strassen ways: 145.553637028s

Although Strassen ways cost more time than traditional ways, but traditional ways rise faster than Strassen ways.

5 Karatsuba Algorithm

5.1 The Method Solving Problem

The algorithm of Karatsuba is a formula that allows us to compute the product of two large numbers x and y using multiplications of smaller problem.

```
procedure karatsuba(num1, num2):
    if (num1<10) or (num2<10) then return num1*num2
    m=max(size_base10(num1),size_base10(num2))
    m2=m/2
    high1, low1=split_at(num1,m2)
    high2, low2=split_at(num2,m2)
    z0=karatsuba(low1,low2)
    z1=karatsuba((low1+high1),(low2+high2))
    z2=karatsuba(high1,high2)
    return z2*10^(2*m2)+(z1-z2-z0)*10^(m2)+z0
```

5.2 The Karatsuba Algorithm for Multiplication Problem in Python

```
from math import ceil

def prepend_zeros(string,n):
    length=len(string)
    new_str=""
    if length<n:
        for i in range(n-length):
            new_str+="0"
        new_str+=string
    else:
        new_str=string
    return new_str


def karatsuba(x,y):
    """Multiplication using karatsuba
        x=10^(n/2)*a+b
        y=10^(n/2)*c+d
        x*y=10^n*ac+10^(n/2)*(ad+bc)+bd
        ad+bc=(a+b)(c+d)-ac-bd
    """
    str_x, str_y=str(x),str(y)
    n=max(len(str_x),len(str_y))
    if n<=1:
        return x*y
    else:
        pass

    str_x=prepend_zeros(str_x,n)
    str_y=prepend_zeros(str_y,n)
    n_2=n/2

    a,b=int(str_x[:n_2] or 0),int(str_x[n_2:] or 0)
    c,d=int(str_y[:n_2] or 0),int(str_y[n_2:] or 0)

    ac=karatsuba(a,c)
    bd=karatsuba(b,d)
    ad_bc=karatsuba((a+b),(c+d))-ac-bd

    n_2=int(ceil(n/2.0))
    n=n if n%2 ==0 else n+1
    return (10**(n)*ac)+(10**n_2*ad_bc)+bd
```

```
print karatsuba(12345,43215)
```

5.3 Prove The Correctness

$$x = x_1 * B^m + x_0$$
$$y = y_1 * B^m + y_0$$
$$xy = (x_1 * B^m + x_0) * (y_1 * B^m + y_0)$$
$$z_2 = x_1 * y_1$$
$$z_1 = (x_1 + x_0) * (y_1 + y_0) - z_2 - z_0$$
$$z_0 = x_0 * y_0$$
$$xy = z_2 * B^{2m} + z_1 * B^m + z_0$$

5.4 Time Complexity

Since the additions, subtractions, and multiplications by powers of B in Karatsuba's basic step take time O(n), then:

$$T(n) = 3 * T(\frac{n}{2}) + cn + d$$

Using Master Theorem:

$$T(n) = \Theta(n^{\log_2 3})$$

5.5 The Performance

Test case: 12345*789, test numbers: 100, repeat numbers: 3.

Karatsuba way time costs:
0.008887052536010742s,
0.007916927337646484s,
0.007441997528076172s.

Traditional way time costs:
0.09397387504577637s,
0.09383106231689453s,
0.08646297454833984s.

Karatsuba way performs better than traditional way in Big Number Multiplication.