CSE_417

Assignment #5

1. (a) $n\log_2 n$

$\log_2(n+1) \fallingdotseq \log_2 n$ (when $n \gg 1$)

(i) $$\frac{(n+1)\log_2(n+1) - n\log_2 n}{n\log_2 n} \fallingdotseq \frac{1}{n}$$ (slower for $\frac{1}{n} \times 100\%$)

(ii) $$\frac{(2n)\log_2(2n) - n\log_2 n}{n\log_2 n} = \frac{2\times n \times [\log_2 n + 1] - n\log_2 n}{n\log_2 n}$$

$$= \frac{n\log_2 n + 2n}{n\log_2 n} = 1 + \frac{2}{\log_2 n} = 1 + \frac{2\log 2}{\log n}$$

(slower for $\left(1 + \frac{2\log 2}{\log n}\right) \times 100\%$)

(b) n^2

(i) $$\frac{(n+1)^2 - n^2}{n^2} = \frac{2n+1}{n^2}$$ (slower for $\frac{2n+1}{n^2} \times 100\%$)

(ii) $$\frac{(2n)^2 - n^2}{n^2} = 3$$ (slower for $3 \times 100\%$)

(c) $100n^2$

(i) $$\frac{100(n+1)^2 - 100n^2}{100n^2} = \frac{200n + 100}{100n^2} = \frac{2n+1}{n^2}$$ (slower for $\frac{2n+1}{n^2} \times 100\%$)

(ii) $$\frac{100(2n)^2 - 100n^2}{100n^2} = 3$$ (slower for $3 \times 100\%$)

(d) n^3

(i) $\frac{(n+1)^3 - n^3}{n^3} = \frac{3n^2+3n+1}{n^3}$ (slower for $\frac{3n^2+3n+1}{n^3} \times 100\%$)

(ii) $\frac{(2n)^3 - n^3}{n^3} = 7$ (slower for $7 \times 100\%$)

(e) 2^n

(i) $\frac{2^{n+1} - 2^n}{2^n} = \frac{2^n}{2^n} = 1$ (slower for $1 \times 100\%$)

(ii) $\frac{2^{(2n)} - 2^n}{2^n} = \frac{(2^n)^2 - 2^n}{2^n} = 2^n - 1$ (slower for $(2^n-1) \times 100\%$)

HW5_CSE417_Yao-Chung Liang_1826630

(2)

Algorithm:

```
For each components C of G:
    Set up a starter node s and label it to be A
    Set up a set P (set of visited node)
    Set up layers L (there are L(0), L(1),…..)
    Put s into P
    Let L(0)=s
    For i = 0,1,2,….
        For each node u in L(i)
            For each edge (u,v) incident to v
                If v is not visited
                    Mark v to be visited, add v into P and to L(i+1)
                    If the judgement (u,v) was "same"
                        Label v the same as u
                    Else
                        Label v to be different from u
For each edge (u,v)
    If the judgement was "different"
        If v and u have the same labels
            It is inconsistent
    Else
        If v and u have different labels
            It is inconsistent
```

Description:

If there are n specimens and m judgements, all specimens will belong to one of two different species, say A and B. First, we need to build up a graph G={V,E}. V means vertex which is each specimen and E means edge which is the judgement between two vertices (include "same" or "different" but exclude "arbitrary"). And from several components of G, we need to assign a start vertex, Let's say component Gi, which means its starter would be Si and be set it as A to be default setting. Let's use BFS, if I go from Vo to Vp and if the judgement between them is "same", I will label Vp the same as Vo. If the judgement is "different", I will label Vp different from Vo. After we use BFS to go through and label all vertices to either A or B species, we can start to check the judgements if they are consistent or not.

Time analysis:

To construct a graph G, it takes O(m+n) since it has n vertices and m edges. Using BFS takes O(m+n). It takes O(m) to check all judgements' consistency. Thus, the total running time is O(m+n).

Correctness:

Because we use BFS to check all node from different component of G. And right now we check all judgements to determine the consistency or inconsistency. If there is an inconsistency inside of the graph, which means m judgements are not consistent. On the other hand, if all labeling are consistent, it means the judgements are consistent.

(3)

(a)

Huffman code:

Algorithm:

Generate a list of 8 Fibonacci numbers (21,5,8,13,2,1,3,1) and sort them to be K

Let K be divided by sum(K) to be a proportional list P

Put P into a stack S (in order of bigger value)

Result = Huffman(S)

```
def Huffman(stack S):
    if S.size==1:
        return T
    else:
        p1=S.pop()           # the one in the top would be pop out
        p2=S.pop()
        p_total = p1 + p2
        S.put(p_total)       # put the combined one in the top of the stack
        Set p1 be left node
        Set p2 be right node
        p_total to be p1's and p2's parent
        let T equals to the tree structure contains p_total, p1 and p2
        Huffman(S)
```

(b)

In a general case, the Huffman code will generate a maximally unbalanced tree. If there are n terms of Fibonacci numbers in the tree, then the first node will have depth of n-1 and the n^{th} node will have depth of 1.

(4)

Algorithm:

There are n classes: C1, C2, ……, Cn

Set K contains all m cards

Result = Find_equi_class(K)

```
Def Find_equi_class (set of card K):
    If size of K==0:
        return None
    If size of K == 1:
        return the only one card                              Base cases
    If size of K==2:
        Check if the two cards in K to be equivalent or not
        If equivalent:
            return either of the 2 cards
    Let K1 be the set of first [1 to m/2] cards
    Let K2 be the set of [(m/2)+1 to m] cards (remaining)
    returned_card=Find_equi_class(K1)
    If returned_card != None                         #there is a returned card
         Check all other cards with the returned_card
    If no returned_card is in the majority equivalence
         returned_card=Find_equi_class (K2)
         If returned_card != None                    #there is a returned card
             Check all other cards with the returned_card
         Else (returned_card == None)
             return None
    If a card from the majority equivalence class is found
        Return all cards from the majority equivalence class
```

Description:

I separate the cards into two piles almost equally (maybe odd or even number) and recursively. After enough recursion, I assume there is a majority of class say U, then at least there should be one of the two sides will have more than half of the cards to be equivalent to class U. Thus, it will return a card for us to check with all the cards. And if there are more than n/2 cards that are equivalent in a class U, then at least there should be one of the two sides that have more than half of the cards equivalent to class U. So if my assumption is right, at least one of the two recursive calls will return a card that has equivalent class U.

Correctness:

Termination :

If n=0, it will check if input size is empty and return an empty output.

If n=1, there will be only one card to be returned and it will go check with the other cards. However, there is only one card there, so it must be in the majority class and the output must be true.

If n=2, it will check the two cards to be equivalent or not. If equivalent, then either one will be return and go check with the other cards, however, there are only two cards there and they are already equivalent, so they must be in majority class and output card must be true.

If n>2, then they will be recursively reduced to n=2 case to compare and get results, and compare with the other cards recursively, thus there is no problem for different size of n.

Analysis:

Let T(n) be the max number of tests that this algorithm needs to run for set of n cards. Every run, I will divide the two piles, and outside of the recursion, I will do 2n test at most. Thus, I will get :

$$T(n)=0 \quad \text{if } n = 1$$
$$T(n)=1 \quad \text{if } n = 2$$
$$T(n)=2T(n/2)+2n-2 \quad \text{if } n>= 3$$

From master recurrence, I know a=2, b=2, c=2 and k=1.

And because $a=b^k=2$, I can infer $T(n)= \Theta(n \log n)$ for master recurrence theorem.

(5)

Input : array A = [[1,6,1],[10,13,2],[2,4,2],[11,12,1],[5,8,3],[3,6,4]]

[Left,Right,Height]

Output: array O=[[1,1],[2,2],[3,4],[6,3],[8,0],[10,2],[13,0]]

```
Def silhouette(Array A):
    If A is empty:                  # in case there is no input rectangle
        return []
    if len(A) == 1:
        L = A[0][0]                                                    Base cases
        R = A[0][1]
        H = A[0][2]
        return [ [L,H] , [R,0] ]    # left point and right point
    mid=len(A)/2
    LEFT = silhouette (A[0:mid])
    RIGHT= silhouette (A[mid+1:])
    LEFT counter=0
    RIGHT counter=0
    result = []
    height for left=None
    height for right=None
    while (length of LEFT is bigger than LEFT counter) and (length of RIGHT is bigger
    than RIGHT counter):
        If LEFT's left is smaller than RIGHT's left:
            new point = [ LEFT's left, max(height for left, height for right) ]
            if (result is empty) or (height of second one from the right from the
            result != height of new point ):
                result += [new point]
            LEFT counter += 1
        Else if LEFT's left is bigger than RIGHT's left:
            new point = [RIGHT's left, max(height for left, height for right) ]
            if (result is empty) or (height of second one from the right from the
            result != height of new point ):
                result += [new point]
            RIGHT counter += 1
        Else:
            new point = [RIGHT's left, max(height for left, height for right) ]
            if (result is empty) or (height of second one from the right from the
            result != height of new point ):
```

```
                    result += [new point]
                LEFT counter +=1
                RIGHT counter += 1
    While (length of LEFT is bigger than LEFT counter):
        if (result is empty) or (height of second one from the left from the
        result != height of new point ):
            result += all LEFT's right
        LEFT counter += 1
    While (length of RIGHTT is bigger than RIGHT counter):
        if (result is empty) or (height of second one from the right from the
        result != height of new point ):
            result += all RIGHT's right
        LEFT counter += 1
    return result
```

Claim. silhouette (A) correctly returns the array of points that construct a

Description:

My algorithm is separating the rectangles recursively till it be only one left and return it's left point (L,H) and right point (R,0). So when it recursively get those points from the left to the right , merge them, and they will be compared under different conditions and get the result set of point.

Correctness:

If n=0, it will check if input size is an empty array and return an empty output.

If n=1, there will be only one pair of [[L,H],[R,0]] as output.

If n=2, they will be separated into two parts, in the RIGHT and in the LEFT, and compare their left values and right values to determine their relative positions and compare their heights to get a dominating one.

For the induction hypothesis,

If n>2, then they will be recursively reduced to n=2 case to compare and get results, and compare recursively, thus there is no problem for different size of n.

Analysis:

$T(n) = 0$ if $n = 1$

$T(n) = 2T(n/2) + 2(n-1)$ if $n > 1$

$=aT(n/b)+c*n^k$ (from master recurrence)

$a=2, b=2, c=2, k=1$

because $a = b^k=2$, thus $T(n)= \Theta(n \log n)$

6. extra credit

(a) version 1:

If A, B and C are n by n matrix, C=A ＊ B

Then there should be (n/2)^3 * 8 = n^3 scalar multiplications.

(b) Version 2

Every time I use the multiplication function MMult, I will separate the matrix into 4 blocks with 8 multiplications.

T(n)=8*T(n/2)+n^2

(c)

T(n) = 1 if n=1

T(n)=8*T(n/2)+n^2 if n>1

=aT(n/b)+c*n^k

a=8, b=2, c=1, k=2

because a=8 and b^k=4 and a>b^k , thus T(n) = $\Theta(n^{\log_b a})$= $\Theta(n^3)$

(d)

T(n)<= 8*T(n/2)+c*n^2 c >= 1

(e)

If A, B and C are n by n matrix, C=A ＊ B

Then there should be (n/2)^2 * 7 = (1.75)*n^2 scalar multiplications.

T(n) = 1 if n=1

T(n)=7*T(n/2)+n^2 if n>1

=aT(n/b)+c*n^k

a=7, b=2, c=1, k=2

because a=7 and b^k=4 and a>b^k , thus T(n) = $\Theta(n^{\log_b a})$= $\Theta(n^{2.807})$

Original time bound would be $\Theta(n^3)$

So it’s a little be improved.

(f)

Every time I use the multiplication function MMult, I will separate the matrix into 4 blocks with combinations of 7 multiplications.

T(n)=7*T(n/2)+n^2

T(n) = 1 if n=1

T(n)=7*T(n/2)+n^2 if n>1

=aT(n/b)+c*n^k

a=7, b=2, c=1, k=2

because a=7 and b^k=4 and a>b^k , thus T(n) = $\Theta(n^{\log_b a})$= $\Theta(n^{2.807})$

(g)

Total number of multiplication would be 7*(n/2)^2.

V. Strassen: (15)*(n/2)^2

Original: (4)*(n/2)^2

After I consider total number of additions, it is worse than previous algorithm. And also in real practice, it will occupy too much memory space which is impractical. My answer is depending on n.