Name: Chaoyu Li
Date: 9/26/2021

CSCI 570 - Fall 2021 - HW 5
Due date : 30th September 2021

1. a) $T(n)=4T(n/2)+n^2\log n$

   $f(n)=n^2\log n$, $n^{\log_b a}=n^{\log_2 4}=n^2$

   Thus, there is no difference in power between $f(n)$ and $n^{\log_b a}$, match case#2 of the master theorem. Note that $f(n)=(n^{\log_b a}\log^k n)$, $k=1$ here.

   Therefore, $T(n)=\Theta(n^2\log^2 n)$

   b) $T(n)=8T(n/6)+n\log n$

   $f(n)=n\log n$, $n^{\log_b a}=n^{\log_6 8}=n^{1.16}$

   Thus, the power of $n^{\log_b a}>f(n)$, matches case#1 of the master theorem.

   $T(n)=\Theta(n^{\log_6 8})$

   c) $T(n)=\sqrt{6000}T(n/2)+n^{\sqrt{6000}}$

   $f(n)=n^{\sqrt{6000}}$, $n^{\log_b a}=n^{\log_2 \sqrt{6000}}$

   Because of the power of $n^{\sqrt{6000}} > n^{\log_2 \sqrt{6000}}$, we need to check if it matches case#3 of the master theorem.

   $af(n/b)=\sqrt{6000}(\frac{n}{2})^{\sqrt{6000}}$, so $\frac{af(n/b)}{f(n)}=\frac{\sqrt{6000}n^{\sqrt{6000}}}{2^{\sqrt{6000}}}*\frac{1}{n^{\sqrt{6000}}}=\frac{\sqrt{6000}}{2^{\sqrt{6000}}}<1$

   Thus, $af(n/b)<cf(n), c<1$, it matches the condition of case#3 of the master theorem.

   $T(n)=\Theta(n^{\sqrt{6000}})$

   d) $T(n)=10T(n/2)+2^n$

   $f(n)=2^n$, $n^{\log_b a}=n^{\log_2 10}$

   Because of the power of $2^n>n^{\log_2 10}$ for sufficiently large n, we need to check if it matches case#3 of the master theorem.

   $af(n/b)=10*2^{\frac{n}{2}}$, so for sufficiently large n, $2^n> 10*2^{\frac{n}{2}}$.

   Thus, $af(n/b)<cf(n), c<1$, it matches the condition of case#3 of the master theorem.

   $T(n)=\Theta(2^n)$

   e) $T(n)=2T(\sqrt{n})+\log_2 n$

   $f(n)=\log_2 n$, set $x=\log_2 n$, then $n=2^x$, $T(n)=T(2^x)=2T(2^{\frac{x}{2}})+\log_2(2^x)$

   Set $T'(x)=T(2^x)$, then $T'(x)=2T'(x/2)+x$

   Here, $f'(x)=x$, $x^{\log_b a}=x$

Thus, there is no difference in power between f(x) and $x^{Log_b^a}$, match case#2 of the master theorem.

T'(x)=Θ(xlogx)=Θ(logn*log(logn))

2. a) Let us suppose that there does not always exist a local minimum for A.

It means that there is not any A[i] meet that A[i-1]≥A[i] and A[i+1]≥A[i].

We already knew that A[1]≥A[2], so for any 2≤j≤n, A[j+1]<A[j]. Otherwise, A[j] will be a local minimum.

Therefore, A[n]<A[n-1].

However, A[n]≥A[n-1], this is a contraction.

Thus, there always exists a local minimum for A.

b) Algorithm:

```
If n=3:
    Return A[2]
Endif
If n>3:
    Find_Localminimum(start, end):(start=1, end=n initally)
        If there are noly two or less number from start to end:
            The current interval does not exist a local minimum
        Endif
        mid=(start+end)/2 (Round down)
        If A[mid-1]≥A[mid] and A[mid+1]≥A[mid]:
            Return A[mid]
        Else:
            If A[mid]>A[mid-1]:
                Find_Localminimun(start, mid)
            Else:
                Find_Localminimum(mid+1, end)
            Endif
        Endif
```

Complexity analysis:

Set T(n)=aT(n/b)+f(n) as the runtime of this algorithm. Each time we recurse, we get one subproblem that is one-half the size of the previous one, a=1,b=2. Both divide and combine have only constant complexity. Therefore, T(n)=T(n/2)+2.

f(n)=2, $n^{Log_b^a}=n^{log_2^1}=1$

Thus, the power of $n^{Log_b^a}$=f(n), matches case#2 of the master theorem.

T(n)=Θ(logn)

Proof of correctness:

If n=3, it is easy to get A[2] must be a local minimum. For n>3, we divide the whole size to ½ and we only consider the number in the middle of the current interval. If A[mid-1]≥A[mid] and A[mid+1]≥A[mid], A[mid] is a local minimum and we will return it. If we do not get any number in this step, it means that A[mid]>A[mid-1] or A[mid]>A[mid+1]. If A[mid]>A[mid-1], then there must be a local minimum in [1,mid] because A[1]≥A[2]. If A[mid]>A[mid+1], then there must be a local minimum in [mid+1,n] because A[n]≥A[n-1]. Therefore, according to a), this algorithm will always find a local minimum in [1,n].

3. For T(n), $f(n)=n^2$, $n^{Log_b a}=n^{\log_2 7}$
   Thus, the power of $n^{Log_b a}>f(n)$, matches case#1 of the master theorem.
   $T(n)=\Theta(n^{\log_2 7})$

   For T'(n), $f(n)=n^2\log n$, $n^{Log_b a}=n^{\log_4 a}$
   Because we need ALG' grow asymptotically faster than ALG, $T'(n)\leq\Theta(n^{\log_2 7})$
   1) If $n^{\log_4 a}<n^2$, that is, a<16, the power of $n^{Log_b a}<f(n)$, we need to check if it matches case#3 of the master theorem.
      $af(n/b)=(\frac{an^2log(\frac{n}{4})}{16})$, $\frac{af(n/b)}{f(n)}=\frac{alog(\frac{n}{4})}{16logn}<1$ because a<16 and log(n/4) grows slower than logn.
      Thus, T'(n) matches the condition of case#3 of the master theorem, $T'(n)=\Theta(n^2\log n)<\Theta(n^{\log_2 7})$
   2) If $^{\log_4 a}=n^2$, that is, a=16, the power of $n^{Log_b a}=f(n)$.
      Thus, T'(n) matches case#2 of the master theorem, $T'(n)=\Theta(n^2\log^2 n)<\Theta(n^{\log_2 7})$
   3) If $^{\log_4 a}>n^2$, that is, a>16, the power of $n^{Log_b a}>f(n)$.
      Thus, T'(n) matches case#1 of the master theorem, $T'(n)=\Theta(^{\log_4 a})$.
      To make $T'(n)\leq\Theta(n^{\log_2 7})$, $\log_4 a\leq\log_2 7 \Rightarrow a\leq 7^2=49$.
   To combine 1), 2) and 3), we get the largest number of a is 49.

4. Algorithm:

```
Run Find_Majority(1,n)
Find_Majority(start, end):
    If end-start=0:
        Return the only card
    Endif
    If end-start=1:
        If two cards are equivalent:
            Return one of them
        Else:
            Return nothing
        Endif
```

```
        Endif
        mid=(start+end)/2 (Round down)
        Set x1 as the return of Find_Majority(start,mid)
        If x1 has value:
            Test x1 against all other cards from [start,end]
            If the number of cards that are equivalent with x1> (end-start+1)/2:
                Return x1
            Endif
        Endif
        Set x2 as the return of Find_Majority(mid+1,end)
        If x2 has value:
            Test x2 against all other cards from [start,end]
            If the number of cards that are equivalent with x2> (end-start+1)/2:
                Return x2
            Endif
        Endif
        If x1 is nothing and x2 is nothing:
            Return nothing
        Endif
    Set x as the answer of the Find_Majority()
    Return x and its majority
```

Complexity analysis:

Set $T(n)=aT(n/b)+f(n)$ as the runtime of this algorithm. Each time we recurse, we get two subproblems that are one-half the size of the previous one, a=2,b=2. For divide, its complexity is constant. For combine, each time we need to check the whole current interval, so the complexity is O(n). Therefore, $T(n)=2T(n/2)+O(n)$.

$f(n)=O(n)$, $n^{\log_b a}=n^{\log_2 2}=n$. It matches case#2 of the master theorem.

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

Proof correctness:

Let us suppose that there is a majority user with more than n/2 cards, then there are more than n/2 cards that have the same equivalence class in the entire [1,n].

Therefore, for any of [1,n/2] or [n/2,n] contains more than half the cards have the same equivalence. At least one of the two intervals will return a card that belongs to the biggest equivalence class.

However, if one of the intervals finds a card that belongs to an equivalence class that contains more than half the cards, it does not mean that this equivalence class is what we want because maybe the other interval has not any card belonging to this equivalence class.

Therefore, when we get a card from the recursion, we need to test it against all other cards in the current whole interval to check if it meets the condition.

5. Algorithm:

    Set min as the smaller number of x and y, and set max as the other
    We start from the root node r, run Find_Lowestparent(r, min, max)
    Find_Lowestparent(p, min, max):
        If min≤p.value≤max:
            Return p as the node we want
        Endif
        If p.value>max:
            Find_Lowestparent(p.leftChild, min, max)
        Else:
            Find_Lowestparent(p.rightChild, min, max)
        Endif

    Complexity analysis:

    Set T(n)=aT(n/b)+f(n) as the runtime of this algorithm. Each time we recurse, we get one subproblem that is one-half the size of the previous one, a=1,b=2. For both divide and combine, the complexity is constant. Therefore, T(n)=2T(n/2)+1.

    f(n)=1, $n^{\log_b a}=n^{\log_2 1}=1$. It matches case#2 of the master theorem.

    Thus, T(n)=Θ(logn)

    Proof correctness:

    According to the definition of the binary search tree, for each node, the value of all nodes of its left subtree must be smaller than it, and the value of all nodes of its right subtree must be greater than it.

    Therefore, we start with the root node, if the value of the current node > the value of x and y, then the lowest ancestry of x and y must in the left subtree.

    If the value of the current node < the value of x and y, then the lowest ancestry of x and y must in the right subtree.

    If the value of the current node is between x and y, then the current node is the lowest ancestry of x and y. It MUST be the lowest ancestry because if it is not, there must be a child of it whose value is also between x and y, it does not meet the definition of a binary search tree.

6. a) Algorithm:

    Set i=1 as the index of skyline1
    Set j=1 as the index of skyline2
    Set h_1=0 as the height of the key point of skyline1

Set h_2=0 as the height of the key point of skyline2
Set array Result to store the information of new skyline
While i≤n or j≤m:
    Set x as the x-coordinate of combined skyline
    Set h as the height of combined skyline on x
    If i>n:
        Set x_i as infinity to let it always > x_j
    Endif
    If j>m:
        Set x_j as infinity to let it always > x_i
    Endif
    If x_i<x_j:
        x=x_i
        h_1=h_i
        h=max(h_1,h_2)
        Make pair of x and h as x,h
        i=i+1
    Else if x_i>x_j:
        x=x_j
        h_2=h_j
        h=max(h_1,h_2)
        Make pair of x and h as x,h
        j=j+1
    Else:
        x=x_i
        h_1=h_i
        h_2=h_j
        h=max(h_1,h_2)
        Make pair of x and h as x,h
        i=i+1
        j=j+1
    Endif
    If the height of the last point in the current Result ≠ h
        Add x,h to Result
    Endif
Endwhile
Return Result as the new skyline

Complexity analysis:

The algorithm above only has one loop and its stop condition is i and j completely traverse skyline1 and skyline2. Every time one of i and j will add one. Therefore, the whole algorithm will execute m+n times at most. The complexity is O(m+n).

b) Algorithm:

Set Combine_Skyline(skyline1, skyline2) as the function what we got in a)
Sort buildings according to their x-coordinates from smallest to largest
Run Get_Skyline(start, end)
Get_Skyline(start, end):
    If start=end:
        Add the only one building to Result
        Return Result
    Endif
    Set mid as the x-coordinate of the middle building
    mid=(start+end)/2 (Round down)
    Set Result1, Result2 to store skylines of left and right intervals
    Result1=Get_Skyline(start, mid)
    Result2=Get_Skyline(mid+1, right)
    Result=Combine_Skyline(Result1, Result2)
    Return Result

Complexity analysis:

First, we sort the whole n buildings, the complexity is O(nlogn).

Set T(n)=aT(n/b)+f(n) as the runtime of the divide and conquer part of this algorithm. Each time we recurse, we get two subproblems that are one-half the size of the previous one, a=2,b=2. For divide, its complexity is constant. For combine, according to a), the complexity is O(n/2+n/2). Therefore, T(n)=2T(n/2)+O(n).

f(n)=O(n), $n^{\log_b a}=n^{\log_2 2}=n$, it matches case#2 of the master theorem.

Thus, T(n)=Θ(nlogn)

To conclusion, the complexity of this algorithm is O(nlogn)+Θ(nlogn)=O(nlogn).