CS 6363.003 Spring 2021 Author. Yaokun Wu (yxw200031)

Homework 1 Problem 1

(a) Truthfully write the phrase ***"I have read and understand the policies on the course website."***

Solution: I have read and understand the policies on the course website. ■

(b) Prove that every non-negative integer can be written as the sum of distinct, non-consecutive Fibonacci numbers.

Solution: Case 1: If an integer k is already a number in the Fibonacci sequence, $k = F_i$, where i is the index in the Fibonacci sequence, for which the argument is true.

Case 2: Assume an arbitrary non-negative integer k that $F_{n-1} < k < F_n$ where n is the index of Fibonacci sequence.

Hypothesis: Assume all integer p < k can be written as the sum of distinct, non-consecutive Fibonacci numbers.

k can be written as

$$k = F_{n-1} + (k - F_{n-1})$$

Let integer $x = k - F_{n-1}$, since x < k, according to the hypothesis, x can be written as the sum of distinct, non-consecutive Fibonacci numbers. Also,

$$0 < x < F_n - F_{n-1} = F_{n-2}$$

Therefore, the largest Fibonacci number in x will be less than F_{n-2}, and k can be written as $k = x + F_{n-1}$, which doesn't violate the non-consecutive rule. Therefore, the argument is true for an every non-negative integer.

■

(c) Prove that every positive integer can be written as the sum of distinct Fibonacci numbers *with no consecutive gaps*.

Solution: Assume an arbitrary non-negative integer k that $F_{n-1} <= k < F_n$ and $k >= 1$ where n is the index of Fibonacci sequence.

Hypothesis: Assume all integer p < k can be written as the sum of distinct Fibonacci numbers with no consecutive gaps such that for p, where $F_{m-1} <= p < F_m$, the addend includes F_{m-2} as it's largest Fibonacci component, m is the index in the Fibonacci sequence.

Base case: It is obvious that the argument is true when the integer is 1 as $1 = F_1$ or $1 = F_2$

k can be written as

$$k = F_{n-2} + (k - F_{n-2})$$

Let integer $x = k - F_{n-2}$, since x < k, according to the hypothesis, x can be written as the sum of distinct Fibonacci numbers with no consecutive gaps. Also, from the first assumption,

$$F_{n-3} = F_{n-1} - F_{n-2} <= x < F_n - F_{n-2} = F_{n-1}$$

which implies that the largest Fibonacci number in x will be either F_{n-3} when $x >= F_{n-2}$, or F_{n-4} when $x < F_{n-2}$, and k can be written as $k = x + F_{n-2}$, which, in either case, doesn't violate the no consecutive gaps rule. Therefore, the argument is true for an every non-negative integer. ■

CS 6363.003 Spring 2021 Author. Yaokun Wu (yxw200031)
Homework 1 Problem 2

Using Θ-notation, provide asymptotically tight bounds in terms of n for the solution to each of the following recurrences.

(a) $A(n) = 8A(n/3) + n^2$

Solution: $A(n) = \Theta(n^2)$

Because $8/(3^2)$ is less than 1, which means that the decreasing pattern applies here and the largest term is cn^2. ■

(b) $B(n) = 27B(n/9) + n^{1.5}$

Solution: $B(n) = \Theta(n^{1.5}logn)$

Because $27/(9^{1.5})$ is 1, which means that the equal pattern applies here. Then, since each level it will require $cn^{1.5}$ work, and the number of level is $logn$, the total work is $n^{1.5}logn$ ■

(c) $C(n) = 7C(n/5) + n$

Solution: $C(n) = \Theta(n^{log_5 7})$

Because $7/5$ is greater than 1, which means that the increasing pattern applies here. Then, since the number of leaves is $7^{log_5 n}$, which equals to $n^{log_5 7}$. ■

(d) $D(n) = 2D(n/4) + T(n/2) + n$

Solution: $D(n) = \Theta(nlogn)$

Because according to the recursion tree, the work in each level will be cn, which means that the equal pattern applies here. And there are at most $log_2 n$ levels, the total work would be $nlog$ ■

(e) $E(n) = 2E(n/2) + n\lg^2 n$

Solution: $E(n) = \Theta(nlg^3(n))$

$$E(n) = 2E(n/2) + nlg^2 n$$
$$E(n) = 2(2E(n/4) + (n/2)lg^2(n/2)) + nlg^2 n$$
$$E(n) = 4E(n/4) + nlg^2(n/2) + nlg^2 n$$
$$E(n) = 8E(n/8) + nlg^2(n/4)) + nlg^2(n/2)) + nlg^2 n$$
$$E(n) = 2^{lg(n)}E(1) + nlg^2(2) + nlg^2(4) + nlg^2(8) + ... + nlg^2(n/4)) + nlg^2(n/2)) + nlg^2 n$$
$$E(n) = 0 + n(lg^2(2) + lg^2(4) + lg^2(8) + ... + lg^2(n)))$$
$$E(n) = n(1^2 + 2^2 + 3^2 + ... + lg^2(n))$$
$$E(n) = n(lg(n)(lg(n) + 1)(2lg(n) + 1)/6)$$
$$E(n) = O(nlg^3(n))$$

■

CS 6363.003 Spring 2021 Author. Yaokun Wu (yxw200031)

Homework 1 Problem 3

(a) Prove by induction that CRUEL correctly sorts any input array.

Solution: First, I am going to prove that UNUSUAL function merge two sorted array A[1...m] and A[m+1..n] into B[1...n].

Base case: when $n = 2$, the UNUSUAL always put $min(A[1],A[2])$ to $B[1]$ and $max(A[1],A[2])$ to $B[2]$.

Hypothesis: Assume that for array length of k, $A[1...k]$, where $k < n$, the UNUSUAL function can merge $A[1...m]$ and $A[m+1...k]$ in sorted order into $B[1...k]$, where m is $k/2$.

So, for array of size n, $A[1...n]$, the first half $A[1...m]$ and second half $A[m+1...n]$ are already in sorted order based on the Hypothesis since $m < n$ and $n-(m+1) < n$. After swapping 2nd and 3rd quarters, $B[1...m]$ contains the smaller half from $A[1...m]$ and from $A[m+1,n]$. $B[m+1...n]$ contains the larger half from $A[1...m]$ and from $A[m+1,n]$.

After calling UNUSUAL($B[1...m]$) and UNUSUAL($B[m+1...n]$), since $m < n$ and $n-(m+1) < n$, the $B[1...m]$ and $B[m+1...n]$ will be both in sorted order. Therefore, $B[1...m/2]$ contains the smallest elements of $B[1...n]$ in sorted order and $B[m+1+n/4...n]$ contains the largest elements of $B[1...n]$ in sorted order. Then, After calling UNUSUAL($B[m/2...m+1+n/4]$), the middle $n/4$ will be in sorted order since $(m+1+n/4-m/2 < n$. Therefore, the array $B[1...n]$ is in sorted order.

Second, I am going to prove that CRUEL function correctly sorts any given the input array $A[1...n]$.

Base case: when $n = 1$, do nothing, the array is already in sorted order.

Hypothesis: Assume CRUEL correctly sorts an array of length k, $k < n$.

Then, CRUEL($A[1...n/2]$) will sorts $A[1...n/2]$ in sorted order since $n/2 < n$. CRUEL($A[n/2+1,n]$) will sorts $A[n/2+1,n]$ in sorted order since $n-(n/2+1) < n$.

And as proved in part I, the UNUSUAL($A[1...n]$) will correctly merge the two sorted arrays $A[1...n/2]$ and $A[n/2+1...n]$ into a totally sorted array. Therefore, CRUEL will correctly sort any given input array $A[1...n]$.

■

(b) What is the running time of UNUSUAL? Justify your answer.

Solution: The running time is $O(n^{log_2 3})$.

Let $T(n)$ represent the time spend for an array of input size n. We have, $T(n) = 3T(n/2)+n/4$. Since 3/2 is greater than 1, it means that the increasing pattern applies here. Then, since the number of leaves is $3^{log_2 n}$, which means that the time complexity is $O(n^{log_2 3})$. ■

(c) What is the running time of CRUEL? Justify your answer.

Solution: The running time is $O(n^{log_2 3})$.

Let $T(n)$ represent the time spend for an array of input size n. We have, $T(n) = 2T(n/2) + n^{log_2 3}$. Since $2/2^{log_2 3}$ is less than 1, it means that the decreasing pattern applies here. The largest term is $cn^{log_2 3}$. which means that the time complexity is also $O(n^{log_2 3})$. ■

CS 6363.003 Spring 2021 Author. Yaokun Wu (yxw200031)
Homework 1 Problem 4

Describe and analyze an algorithm to find a central vertex in an arbitrary given binary tree.

Solution: The Algorithm works in following way:

1. Create a hashtable to store the number of nodes rooted at each node in the tree.

2. Use the SIZE function to calculate the number of nodes in bottom up order(divide and conquer) and store the result to the hashtable created above.

3. Use the function SearchVertex to go through each node in the tree, at each node, check the number of nodes of its parent, left child and right child, if all three are less than or equal to the total number of nodes divided by 2, we find a central vertex and return.

Below is the Pseudocode for the algorithm.

```
FindCentral(root):
    if leftChild = null and rightChild = null
        return null
    Create HashTable T
    N_node ← size(root, T)
    return SearchVertex(root, N_node)
```

```
Size(root, T):
    if root = null
        return 0
    T[root] ← size(leftChild, T) + size(rightChild, T) + 1
    return T[root]
```

```
SearchVertex(root, N_node):
    if root = null
        return root
    N_parent ← N_node − T[root]
    N_left ← 0
    N_right ← 0
    if leftChild != null
        N_left ← T[leftChild]
    if rightChild != null
        N_right ← T[rightChild]
    if N_parent <= N_node / 2 and N_left <= N_node / 2 and N_right
    <= N_node / 2
        return root
    else
        Left_result ← SearchVertex(leftChild, N_node)
        if Left_result != null
            return Left_result
        else
            Right_result ← SearchVertex(rightChild, N_node)
            return Right_result
```

Proof of correctness:

Correctness of SIZE:

Base case: when current tree node is null/empty, 0 is return.

Hypothesis: Assume that SIZE function gives the correct size of a tree of size k rooted at a tree node c, where k is less than n of size of the tree rooted at tree node p, who is the predecessor of c.

Let lc and rc be the child of tree node p. Since the size of tree rooted at lc and rc is less than n, the SIZE will get the correct size of lc and rc. Since the size of tree p, is nothing but one node more than the total number of nodes of lc and rc, the SIZE function will get the correct size of tree rooted at p of size n.

Correctness of SEARCHVERTEX:

Base case: when current tree node is null/empty, empty is return.

Hypothesis: Assume that SEARCHVERTEX correctly check whether all the successors node of the root p of the given tree is a central and return if there is one.

Case 1: When a central is on the left subtree of root p and right subtree of root p, the SEARCHVERTEX function will return the correct central node according to the hypothesis.

Case 2: When a central node is not on either left subtree or right subtree, the SEARCHVERTEX function will check whether the current node(itself) satisfy the requirements and return correctly.

Therefore, when the SEARCHVERTEX calls on the root node of the tree, it either returns the central node returned by its children or itself.

Correctness of FINDCENTRAL:

Case 1: When the given tree has only one node, the node itself is a central node according to the definition.

Case 2: Since the SIZE and SEARCHVERTEX functions are working correctly, the FINDCENTAL function will return the correct result from the output of SEARCHVERTEX.

Time and space complexity:

Assume that the number of nodes in the tree is n.

1. The running time for function SIZE is O(n) because the recursion is in bottom up manner and each node requires O(1) to process so that the overall running time is O(n).

2. The running time for function SEARCHVERTEX is also O(n) because the each node will be processed once and it takes constant time to process each node so that the overall running time is at most O(n).

3. The running time for FINDCENTRAL is therefore the maximum running time between the two function, which is also O(n).

Similarly, the space complexity is also O(n) because the hashtable need to store n entries. And recursion stack is at most O(n). Overall, it is O(n) space complexity. ■