Yunjia Zeng
yz682

Homework 1

ANLY 550
February 7, 2019

Problem 1

1. **Recursive Method:**

   After one minute of the machine time, we have the recursive method goes to the 38th Fibonacci number 39088169. After modification for modulo, we obtain the recursive method goes to the 38th Fibonacci number after modulo, which is 21547.

   The Fibonacci sequence is represented as F_0, F_1, $\ldots$, where $F_0 = 0$, $F_1 = 1$, and for all $n \geq 2$, F_n is defined as $F_{n-1} + F_{n+1}$. The recursive methods strictly follows the methods of general computation of Fibonacci numbers. Therefore, it is clear that $T(0) = 0$, and $T(1) = 1$, and for any $n \geq 2$, we have

   $$T(n) = T(n-1) + T(n-2) + 1$$

   It is obvious that the running time for the recursive method is the slowest among the three methods, since the function calls itself recursively to solve for next number. After we test for all three methods, we verified that the recursive method is the slowest.

2. **Iterative Method:**

   After one minute of the machine time, we have the iterative method goes to the 16756th Fibonacci number. After modification for modulo, we obtain the iterative method goes to the 21500th Fibonacci number. Since the last two methods compute very large Fibonacci number, we don't state them out here.

   The iterative method improves the recursive method, by avoid repeated calculations. We calculate just the successive values for iterative method, and we will do only $n-1$ additions. The iterative method is much faster than the recursive method, since for each next Fibonacci number, we only computed by the previous two number in the predefined list, and store it in the next position in the list.

3. **Matrix Method:**

   After one minute of the machine time, we have the matrix method goes to the 59273th Fibonacci number. After modification for modulo, we obtain the matrix method goes to the 1302513th Fibonacci number. Since the last two methods compute very large Fibonacci number, we don't state them out here.

   The matrix method is based on the matrix notation of the Fibonacci number computation

   $$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}.$$

   The computation suffices to raise the 2 by 2 matrix to the nth power, and each multiplication takes 12 arithmetic operations. To raise the matrix to the nth power, we compute $X^{n/2}$ and then square it, therefore the recurrence is

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

Problem 2

A	B	O	o	Ω	ω	Θ
$\log n$	$\log(n^2)$	Yes	No	Yes	No	Yes
$\sqrt[3]{n}$	$(\log n)^6$	No	No	Yes	Yes	No
$n^2 2^n$	3^n	Yes	Yes	No	No	No
$\frac{n^2}{\log n}$	$n \log n^2$	No	No	Yes	Yes	No
$(\log n)^{\log n}$	$\frac{n}{\log n}$	No	No	Yes	Yes	No
$100n + \log n$	$(\log n)^3 + n$	Yes	No	Yes	No	Yes

Problem 3

1. $f_1(n) = n$.

   Proof. Since $f_1(n) = n$, then $f_1(2n) = 2n$. Therefore, we have $2n \leq 3n$ when $n \geq 1$, which is equivalent to $f_1(2n) \leq 3f_1(n)$. $\square$

2. $f_2(n) = 3^n$.

   Proof. Since $f_2(n) = 3^n$, then $f_2(2n) = 3^{2n} = 9^n$, which is greater than 3^n for any $n \neq 0$. Therefore, there is no such a constant c and N that satisfy $9^n \leq 3^n$ for any $n \geq N$, since 9^n is growing exponentially faster than $c3^n$. Thus we conclude $f_2(2n)$ is not $O(f_2(n)$ for $f_2(n) = 3^n$. $\square$

3. *Proof.* We know that $f(n)$ is $O(g(n)) \implies f(n) \leq cg(n)$, such that c and N_1 are some positive constants and $n \geq N_1$. Similarly, $g(n)$ is $O(h(n)) \implies g(n) \leq dh(n)$, such that d and N_2 are some positive constants and $n \geq N_2$. Therefore, we have $cg(n) \leq cdh(n) \implies f(n) \leq cg(n) \leq cdh(n)$. Let k represent cd, therefore, $f(n) \leq kh(n)$ for all $n \geq N*$, where k, N^* are some positive constant, and $N* \in max\{N_1, N_2\}$. Thus by definition, $f(n)$ is $O(h(n))$. $\square$

4. False. If $f(n) = n$, and $g(n) = n^{sin(x)+1}$, then $f(n)$ is not $g(n)$ and $g(n)$ is not $f(n)$.

5. True.

   Proof. Suppose we have $f(n)$ is $o(g(n))$, then $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. Therefore, there exist an positive constant N, which is large enough, such that for all $n \geq N$ we have $f(n) < g(n)$. Thus, there obviously exist a positive constant c such that $f(n) \leq cg(n)$, for all $n \geq N$. By definition, we have $f(n)$ is $O(g(n))$. $\square$

Problem 4

Proof. We prove the correctness of StoogeSort by induction. Let the length of the list be n.

Base Case: For $n \leq 3$, we clearly will have the list correctly sorted.

Induction: Assume StoorgeSort correctly sort all lists of size less than n, for $n \geq 3$.

Let P, Q, G represent the first 1/3, second 1/3, and last 1/3 of the list. Then after first phase, the sub-list $[P, Q]$ is sorted, such that every element in Q is now greater than that of in P. After second phase, the sub-list $[Q, R]$ is sorted, and every element in R is now greater than that of in Q. Therefore, all element in R are now greater than other elements in the current list. Finally after the third phase, the sub-list $[P, Q]$ is sorted again and every element in Q is now greater than that of in P, which means all elements in P not are smaller than other elements in the current list. Now the list is correctly sorted. □

The running time of StoogeSort can be described as the recurrence:

$$T(n) = 3T(\frac{2}{3}n) + \Theta(1)$$

Therefore, since here we have $a = 3$, $b = 3/2$, and $k = 0$, $T(n)$ is $O(n^{\log_{3/2} 3})$.

Problem 5

1. Here we have $a = 4$, $b = 2$, $c = 1$, and $k = 3$, since $a < b^k$, $T(n)$ is $O(n^3)$.

2. Here we have $a = 17$, $b = 4$, $c = 1$, and $k = 2$, since $a > b^k$, $T(n)$ is $O(n^{\log_4 17})$.

3. Here we have $a = 9$, $b = 3$, $c = 1$, and $k = 2$, since $a = b^k$, $T(n)$ is $O(n^2 \log n)$.

4. Let $m = \log_2 n$, such that $n = 2^m$, and therefore $T(2^m) = T(2^{m/3}) + 1 = F(m) = F(m/3) + 1$, where F is another recurrence. Therefore, for $F(m)$ we have $a = 1$, $b = 3$, $c = 1$, and $k = 0$, and since $a = b^k$, $F(m)$ is $O(logm)$. Since $T(n) = T(2^m) = F(m)$, we have $T(n) = O(\log(\log_2 n))$.

Problem 6

Let A devotes the amount of time that will be spent for singing all words in the inner loop once, and let B devotes the amount of time that will be spent for singing all words in the outer loop that except the words in the inner loop once. Let C devotes the total amount of time that will be spent for singing the words outside of two loops. Therefore, we have the recurrence as

$$T(n) = n(B + \sum_{j=1}^{n} A_j) + C = n \cdot B + n \cdot \sum_{j=1}^{n} A_j + C$$

Since $\sum_{j=1}^{n} A_j = ((1+n))/2 = (n+n^2)/2$, we now have

$$\begin{aligned} T(n) &= n \cdot B + (n+n^2)/2 \cdot A + C \\ &= n \cdot B + (n \cdot A + n^2 \cdot A)/2 + C. \end{aligned}$$

Therefore, we have $O(T(n)) = O(n^2)$. Since the inner loop contains 2 words, and it takes 1/2 seconds, and the outer loop contains 44 words, and it takes 11 seconds, so we have $B = 11$, and $A = 0.5$. Since the words outside the outer loop is constant, which is 50, therefore $C = 12.5$. Thus,

$$T(n) = 11n + (n+n^2)/4 + 12.5$$

We can find a $c = 17$, $N = 2$, we have $11n + (n+n^2)/4 + 12.5 \leq c \cdot n^2$, for all $n \geq N$.

Problem 7

Note: for testing the script mergesort.py, please simply input the string of numbers by only the numbers themselves and commas, without any other symbols.

Referring to the mergesort.py, the algorithm that I implemented are divided into two parts as what described in the lecture notes. The first part is the merge function, which takes two arrays as inputs, then checks for if the two inputs arrays are empty arrays. If one of the two is empty, the function will just return the other array. If both arrays are not empty, we simply compare them from start of each, and append the bigger one in the output array. The output array is empty initially.

The second part of the algorithm is the mergesort function, which takes a full array from user input, then split the array of size $n/2$, where n is the length of the unsorted array. Then we do the same thing to both the two sub-arrays, and then until we reach the minimum unit for each sub-array, we call the merge function to merge the minimum units. At last the function output the sorted array.