problem_set1

February 6, 2019

1. How far does each process get after one minute of machine time? For each method, what is the largest Fibonacci number you can compute in 1 minute of machine time? Submit your source code with your assignment. Please give a reasonable English explanation of your experience with your programs.
The recursive implementation was able to calculate the 38th fibonacci number modulo 65536 after 1 minute runtime. The iterative implementation was able to calculate the 15283th fibonacci number modulo 65536 after 1 minute runtime. The matrix implementation was able to calculate a fibonacci number modulo 65536 that was higher than the 15283th fibonacci number modulo 65536. It make sense for the recursive implementation to be the slowest because there is a lot of redundant steps involved in computing the numbers. The iterative implementation was the 2nd fastest. The time complexity is $O(n)$ because the for loop is the dominant term, which has n-2 steps. The matrix method also uses a for loop, but the for loop does not range from 2 to n. The for loop in the matrix method ranges from 1 to i+1, and i+1 is less than n. i represent the number of times matrix multiplication is done. Also, the matrix method assumes that $n = 2^i$. So, the matrix method is not linearly computing all the fibonacci numbers, whereas the iterative method linearly computes all the fibonacci numbers up until the max fibonacci number it can compute in 1 minute.

2. Indicate, for each pair of expressions (A, B) in the table below, the relationship between A and B. Your answer should be in the form of a table with a yes or no written in each box. For example, if A is O(B), then you should put a "yes" in the first box.

O	o	Ω	ω	Θ
Yes	No	Yes	No	Yes
No	No	Yes	Yes	No
Yes	Yes	No	No	No
No	No	Yes	Yes	No
No	No	Yes	Yes	No
Yes	No	Yes	No	Yes

3. Find with proof a function f_1 such that f_1 is O(f_1).
Let $f(x) = x$. Then, $f_1(2n) = 2n$ and $f_1(n) = n$. There exists a constant c such that $f_1(2n)$ is O($f_1(n)$). In other words, there exists a constant c such that the following is true: $2n \leq cn$ for $n \geq N$ Let $c = 2$. Then, $2n \leq 2n$ for $n \geq 1$

3. Find with proof a function f_2 such that $f_2(2n)$ is not O($f_2(n)$).

Let $f_2(x) = 3^x$. Then, $f_2(2n) = 3^{2n}$ and $f_2(n) = 3^n$, and $\lim_{n\to\infty}(3^{2n}/3^n) = 3^{2n-n} = 3^n = \infty$
Therefore, $f_2(2n)$ is $\omega(f_2(n))$ This implies that $f_2(2n)$ is not $O(f_2(n))$

3. Prove that if $f(n)$ is O($g(n)$), and $g(n)$ is O($h(n)$), then $f(n)$ is O($h(n)$).
   Assume that $f(n)$ is $O(g(n))$.
   Then by definition,
   $f(n) \leq c_1 g(n)$ for $n \geq N$.
   Assume that $g(n)$ is $O(h(n))$.
   Then by definition,
   $g(n) \leq c_2 h(n)$ for $n \geq N$.
   $= c_1 g(n) \leq c_1 c_2 h(n)$
   $= f(n) \leq c_1 g(n) \leq c_1 c_2 h(n)$
   Let $c_3 = c_1 c_2$. Then,
   $f(n) \leq c_3 h(n)$.
   Therefore, $f(n)$ is $O(h(n))$

3. Give a proof or a counterexample: if f is not $O(g)$, then g is $O(f)$.
   Counter example:
   Let $f(n) = n^2$ if n is odd and let $f(n) = n^4$ if n is even. Also, let $g(n) = n^3$.
   Then, $f(n)$ is not $O(g(n))$ for $n \geq N$.
   $= n^2 \leq cn^2$ or $n^4 \nleq cn^3$
   The reason being is that $n^4 \geq cn^3$ and $n^4 > cn^3$.
   Therefore, $f(n)$ is not $O(g(n))$ because it is possible for $f(n) > cg(n)$ when n is large and even.
   Therefore, $f(n)$ satisfies the premise.
   However, $g(n)$ is not $O(f(n))$.
   $= n^3 \nleq cn^2$ if n is odd or $n^3 \leq cn^4$ if n is even. This means that, when n is large and is odd, $g(n)$ is not $O(f(n))$

3. Give a proof or a counterexample: if f is $o(g)$, then f is $O(g)$.
   Assume f is $o(g)$. Then by definition,
   $\lim_{n\to\infty}(f/g) = 0$.
   By definition, if f_1 is $O(g_1)$, then
   $\lim_{n\to\infty}(f/g) < \infty$
   Since $\lim_{n\to\infty}(f/g) = 0$ is less than infinty, then f is $O(g)$

4. Prove rigorously that StoogeSort correctly sorts. To keep things simple, assume that the length of the input is divisible by 3. Proof should use induction. Unfortunately StoogeSort can be slow. Derive a recurrence describing its running time, and use the recurrence to bound the asymptotic running time of StoogeSort.
   Input: Array A[1,2,3,...,n] is an unsorted array of length n. A is a list of distinct numbers.
   Output: Permutation B of A such that B[1] < B[2] < B[3] < ... < B[n]. Array B is sorted. Base case:
   Suppose $n = 1$. If $n = 1$, then array A is sorted. Therefore, return A.
   Inductive case:
   Let X represent 1st 1/3 of A, Y represent 2nd 1/3 of A, and Z represent 3rd 1/3 of A. Assume the length of A is divisible by 3 and $n > 1$. This means that $n = 3^i$ for some positive integer i. Assume StoogeSort returns correct sorted array for input size less than n. Suppose we call StoogeSort on input of size n. Then StoogeSort will recursively call StoogeSort on input

of size $(2/3)n$ for the 1st phase of StoogeSort. The 1st phase of StoogeSort recursively calls StoogeSort on the 1st $(2/3)n$ of A. By the induction hypothesis, these calls will sort the new array correctly because the new array is smaller than n. This means the 1st 2/3 of A will be sorted correctly. After the 1st phase of StoogeSort, $X < Y$.
For 2nd phase of StoogeSort, StoogeSort will recursively call StoogeSort on input of size $(2/3)n$. The 2nd phase of StoogeSort uses the last 2/3 of A as the input. By the induction hypothesis, 2nd phase of StoogeSort correctly sorts because the input size for recursive calls is less than n. After the 2nd phase of StoogeSort, $Y < Z$.
After the 2nd phase of StoogeSort, $X < Z$ is now true because all large numbers that were in X were moved to Y in phase 1 and all large numbers in Y are moved to Z in phase 2. The opposite is also true where all small numbers in Z are moved to Y and all small numbers in Y are moved to X. However, the small numbers that were moved to Y from Z in phase 2 could be smaller than the numbers that are in X. In other words, $X < Y$ is not true anymore. Therefore, a 3rd step is needed to move the small numbers in Y to X, so $X < Y$.
Since the input size is $(2/3)n$ for phase 3, the inductive hypothesis says that phase 3 must sort correctly. After all 3 phases, Array A[1,2,3,...,n] = A[X<Y<Z].
Runtime:
At each of the 3 phases of StoogeSort, the algorithm recursively calls StoogeSort on a single Array of size $(2/3)n$. In the worst case, there is at most $(2/3)n - 1$ comparisions to sort each number into their correct place. Therefore, the runtime for a single phase of StoogeSort is $T(n) = T((2/3)n) + (2/3)n - 1$. Since there are 3 phases in StoogeSort, the runtime for the entire algorithm is the sum of $T(n) + T(n) + T(n)$. In other words, the overall runtime is $T(n) = 3T((2/3)n) + 2n - 3$. Since $a = 3, b = 3/2, c = 2, k = 1$, the master theorem says that $T(n) = O(n^{log_{3/2}(3)})$.

5. Give asymptotic bounds for T(n) in each of the following recurrences. You may invoke the Master Theorem.
$T(n) = 4T(n/2) + n^3 \rightarrow T(n)$ is $O(n^3)$
$T(n) = 17T(n/4) + n^2 \rightarrow T(n)$ is $O(n^{log_4(17)})$
$T(n) = 9T(n/3) + n^2 \rightarrow T(n)$ is $O(n^2 log(n))$
Let $n = 2^{2^m}$, and $m = loglog(n)$.
Plug in $n = 2^{2^m}$ into $T(n)$.
$T(n) = T(n^{1/2}) + 1 \rightarrow T(2^{2^m}) = T((2^{2^m})^{1/2}) + 1$
$= T(m) = T(m-1) + 1$
Plug in $m - 1$ into $T(m)$. Then we get, $T(m-1) = T(m-2) + 1$ Since $T(m-1) = T(m) - 1$, we can substitute this in $T(m-1) = T(m-2) + 1$ to get $T(m) - 1 = T(m-2) + 1$. Therefore, $T(m) = T(m-2) + 2$. If we continue to repeat this process, the equation will eventually become $T(m) = T(0) + m$. $T(0)$ means that a single recursive call is being made on an input of 0. In other words, this means that the recursion does not happen and there is no runtime associated with it. Therefore, the dominant term is m and $T(m) = m$ is the runtime. Since $m = loglog(n)$, $T(m) = O(loglog(n))$

6. How long would it take you to sing BARLEYMOW(n)?
When the function runs, the 1st 6 lines are 6 steps. For input of size n, the outer for loop iterates n times. This means that the 6 lines in the outer for loop represent 6n steps. Notice that when $i = 1$, the nested for loop occurs 1 time. When $i = 2$, the nested for loop occurs 2 times. When $i = 3$, it occurs 3 times. When $i = n$, the inner for loop occurs n times. Therefore, the total number of steps in inner for loop is $1 + 2 + 3 + 4 + 5 + 6 + ... + n =$

$n(n+1)/2$.
Let $T(n)$ be the runtime. Then,
$T(n) = 6 + 6n + n(n+1)/2 = 6 + 6n + (1/2)n^2 + (1/2)n$
$= 6 + 6.5n + (1/2)n^2$
$T(n) = 6 + 6.5n + (1/2)n^2 \leq cn^2$
There exists a constant c such that the inequality holds for some large n.
$T(n) = O(n^2)$
$6 + 6.5n + (1/2)n^2 \geq cn^2$
There exists a constant c such that the above is true. Therefore, $T(n) = \Omega(n^2)$.
Therefore, $T(n) = \Theta(n^2)$.
Extra math:
$6 + (6+1) + (6+2) + (6+3) + (6+4) + (6+5) + ... + (6+n)$
$= 6 + 6n + \Sigma_{i=1}^{n} i$
$= 6 + 6n + n(n+1)/2 = T(n)$

7. Implement merge sort in python.
   Merge sort takes an unsorted array as the input and splits the array into 2 equal parts recursively. Then the algorithm returns a sorted array from least to highest. The sorting is done by comparing the element in the 1st index of the two arrays. The smaller element is put into the 1st index of the merged array. This process is done recursively until the array is sorted. If the size of the array is 1, then the array is already sorted, so that is returned by the function. The runtime of this algorithm is $T(n) = 2T(n/2) + n - 1 = O(nlog(n))$ by the master theorem. The runtime follows from the fact that there is 2 recursive calls on input size of $n/2$, and at the sorting step, there can be at most $n - 1$ comparisons.