550 Homework #1

Brody Vogel

1/29/2018

Worked with: Jinghao Yan, Min Xiao, April Chung

1.

(See files 'recursive.py' , 'iterative.py' , and 'matrix.py')

For the recursive algorithm, the largest Fibonacci number I could compute was F_{39}. Furthermore, when the program was modified to return Fibonacci numbers modulo 2^{16}, the algorithm still only made it to F_{39}. This means most of the time taken by the algorithm is spent in the recursion, and that little is spent working with large numbers.

The iterative algorithm runs much faster. The largest Fibonacci number I could compute with the regular algorithm was $F_{14,769}$, while the largest reached using the modulo 2^{16} modification was $F_{18,610}$. A fairly large chunk of the time used by the regular algorithm must be devoted to dealing with large numbers, then, as the modulo modification allowed it to compute over 20% more Fibonacci numbers.

There was a fair bit of variance across my runnings of the matrix algorithm. It seems to take longer than the iterative algorithm to compute small Fibonacci numbers, with the opposite being true for larger Fibonacci numbers. On average, I'd say, the largest Fibonacci number I could compute with the regular matrix algorithm was $\approx F_{10,000}$, and the largest I could reach using the modulo 2^{16} modification was $\approx F_{10,500}$. In this algorithm, therefore, it would seem the time spent working with large numbers has little effect on the runtime. This makes sense, especially compared to the iterative algorithm, since there are folds fewer operations performed on large numbers in this algorithm. My guess would be that, for longer time periods, this matrix algorithm would surpass the iterative algorithm. Something may be amiss here, though, because I got very different results when using the numpy module.

2.

A	**B**	**O**	**o**	Ω	ω	Θ
$logn$	$log(n^2)$	YES	NO	YES	NO	YES
$n^{1/3}$	$(logn)^6$	NO	NO	YES	YES	NO
$n^2 2^n$	3^n	YES	YES	NO	NO	NO
$n^2!$	n^n	NO	NO	YES	YES	NO
$\frac{n^2}{logn}$	$nlog(n^2)$	YES	NO	YES	YES	YES
$logn^{logn}$	$\frac{n}{logn}$	NO	NO	YES	YES	NO
$100n + logn$	$(logn)^3 + n$	YES	NO	YES	YES	YES

3.

) Let $f_1(n) = n$, $c = 4$, and $n_0 = 1$.

Then $0 \leq f_1(2n) \leq 4f_1(n) <=> 0 \leq 2n \leq 4n$, which is clearly true for all numbers $n_0 \geq 1$.

So, if $f_1(n) = n$, there exist constants $c > 0$ and $n_0 > 0$ such that, for all $n \geq n_0$, $0 \leq f_1(2n) \leq cf_1(n)$, which - by definition - means that $f_1(2n) = O(f_1(n))$.

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

If $f_2(2n)$ were $O(f_2(n))$, there would exist constants $c > 0$ and $n_0 > 0$ such that $0 \leq f_2(2n) \leq cf_2(n)$ for all $n \geq n_0$. Substituting, this would mean $0 \leq 2^{2n} \leq c2^n$, for some constants $c > 0$ and $n \geq n_0$.

Taking logarithms, the above can be rewriteen as: $0 \leq log(2^{2n}) \leq log(c2^n)$. Ignoring the first bit of the inequality, the implication is: $log(2n) \times log(2) \leq log(c) + log(n) \times log(2)$ for some constants $c > 0$ and $n \geq n_0$.

But this implies a contradiction, as $\lim_{n\to\infty} \frac{f_2(2n)}{f_2(n)} = \lim_{n\to\infty} \frac{log(2n)\times log(2)}{log(c)+log(n)\times log(2)} = \infty$; so $f_2(2n) \nleq cf_2(n)$ for some constnat $c > 0$ and all $n \geq n_0$.

Thus, because assuming it is true leads to a contradiction, if $f_2(n) = 2^n$, $f_2(2n)$ is *not* $O(f_2(n))$.

) If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then there exist positive constants c_1, c_2 and non-negative n_{0a}, n_{0b} such that $0 \leq f(n) \leq c_1 g(n), \forall n \geq n_{0a}$ and $0 \leq g(n) \leq c_2 h(n), \forall n \geq n_{0b}$.

Consider, then, a number n_{0c} such that $n_{0c} \geq max(n_{0a}, n_{0b})$.

Next, by inspection we recognize that $c_1 g(n) \leq c_1 c_2 h(n)$, since $c_1, c_2 > 0$.

So, putting everything together, $0 \leq f(n) \leq c_1 g(n) \leq c_1 c_2 h(n), \forall n \geq n_{0c}$.

Removing the middle term, we have: $0 \leq f(n) \leq c_1 c_2 h(n), \forall n \geq n_{0c}$, i.e., there exist constants $c_a = c_1 \times c_2 > 0$ and $n_{0c} = max(n_{0a}, n_{0b}) \geq 0$ such that $0 \leq f(n) \leq c_a h(n), \forall n \geq n_{0c}$, which proves - by definition - that $f(n) = O(h(n))$.

) Let

$$f(n) = \begin{cases} \frac{1}{n}, & \text{if } n \text{ is odd.} \\ n^n, & \text{otherwise.} \end{cases} \tag{1}$$

and $g(n) = n$.

If $f(n)$ were $O(g(n))$, there would exist constants $c > 0$ and $n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. Letting n be even and substituting, this would mean $0 \leq n^n \leq cn$, for some constants $c > 0$ and $n \geq n_0$.

But this implies a contradiction, as $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^n}{n} = \infty$; so $f(n) \nleq cg(n)$ for some constant $c > 0$ and all $n \geq n_0$, and so $f(n)$ is *not* $O(g(n))$.

Now, if $g(n)$ were $O(f(n))$, there would exist constants $c > 0$ and $n_0 > 0$ such that $0 \leq g(n) \leq cf(n)$ for all $n \geq n_0$. Letting n be odd and substituting, this would mean $0 \leq n \leq c\frac{1}{n}$, for some constants $c > 0$ and $n \geq n_0$.

But this, too, implies a contradiction, as $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{n}{\frac{1}{n}} = \infty$; so $g(n) \nleq cf(n)$ for some constamt $c > 0$ and all $n \geq n_0$, and so $g(n)$ is *not* $O(f(n))$.

Thus, this counterexample proves that it is *not* the case that if f is not $O(g)$ then g is $O(f)$.

) If $f = o(g)$, then for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n), \forall n \geq n_0$.

Let f, g be any functions such that $f = o(g)$. Then, by the definition above, there exist a positive constant $c > 0$ and a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n), \forall n \geq n_0$.

Since $cg(n) \leq cg(n)$ and $n_0 > 0 \geq 0$, by definition, we can tediously rewrite the above as $0 \leq f(n) < cg(n) \leq cg(n), \forall n \geq n_0 \geq 0$. Removing the needless middle term, we get: $0 \leq f(n) \leq cg(n), \forall n \geq n_0$.

So there exist constants $c > 0$ and $n_0 \geq 0$ such that $0 \leq f(n) \leq cg(n), \forall n \geq n_0$, which proves - by definition - that if $f = o(g)$ then $f = O(g)$.

4.

(Assuming StoogeSort() sorts only lists of size 3x in descending order)

Proof of Correctness by Induction

Base Cases (list.size = 1 or list.size = 2):

If the size of the list is 1, StoogeSort() returns the one-valued list, which is clearly correctly sorted.

If the size of the list is 2 (which in our restricted setting will only come about in the recursive calls), StoogeSort() compares the first value in the list to the second and arranges them in descending order, which is to correctly sort the two-valued list.

Case list.size = $3x$:

Assume for our inductive hypothesis that StoogeSort() correctly sorts lists of length $\leq 3x$. Before moving on, we should note one thing about the way StoogeSort() handles lists of length $3x$.

- In the recursive calls, StoogeSort() sorts the sublists indexed at $[1 : 2x]$, $[x : 3x]$, and again $[1 : 2x]$. These sublists do not neccessarily have lengths equal to a multiple of 3. By our assumption, though, StoogeSort() correctly sorts these lists. So StoogeSort() has a method - like an applied floor function - for sorting lists that are not of a length equal to a multiple of 3. Thus, as part of our inductive hypothesis, StoogeSort() correctly sorts all lists with lengths $\leq 3x$, regardless of whether they have lengths equal to a multiple of 3.

Moving on, we will show that our inductive hypothesis implies StoogeSort() correctly sorts lists of size $3(x+1)$. For convenience, we can introduce the variable $z = x + 1$; i.e., we will now show StoogeSort() correctly sorts lists of size $3(z)$. This can be done by walking through the three recursive calls in StoogeSort().

- For convenience, we can call the first, middle, and final thirds of the list to be sorted A, B, and C.
- (1) In the first step, StoogeSort() sorts the sublist indexed at $[1 : 2z]$. By our inductive hypothesis, this sublist is correctly sorted, because it by definition has length $\leq 3x$. This means that every element in the first 2/3 of the list is in the correct descending order. That is, each element in the middle third of the list is strictly smaller than every element in the first third, $A > B$.
- (2) In the second step, StoogeSort() sorts the sublist indexed at $[z : 3z]$. From (1), we know that every element in B is smaller than every element in A. Now, by our inductive hypothesis, we know the sublist $[z : 3z]$ is correctly sorted. Putting the last two sentences together, we know that the last third of the list has been compared to every element in the list and containts strictly the smallest elements, $B > C$ and $A > C$.
- (3) In the final step, StoogeSort() again sorts the sublist indexed at $[1 : 2z]$. From (1) and (2), we know that every element in C is strictly smaller than every element in A and B. Now, by our inductive hypothesis, we know the sublist $[1 : 2z]$ is correctly sorted. Putting this all together, this means that the first third of the list contains the largest elements in descending order, the middle third the next largest elements in descending order, and the final third the smallest elements in descending order. That is, $A > B > C$, and the entire list is correctly sorted.

Thus, by induction, StoogeSort() correctly sorts.

The runtime of StoogeSort() is: $T(n) = 3T(\frac{2n}{3}) + \Theta(1)$.

- There are 3 recursive calls, each on a fraction of $\frac{2}{3}$ the original input, n.
- The other operations (like if-statements and a computation of a floor function) take constant time, $\Theta(1)$.

So $a = 3, b = \frac{3}{2}, c = 1, k = 0$, and $a > b^k$. So StoogeSort() falls into case 1 of the Master Theorem: $T(n) = O(n^{log_b a}) <=> T(n) = O(n^{log_{\frac{3}{2}} 3}) <=> T(n) = O(n^{\approx 2.71})$.

5.

$T(n) = 4T(n/2) + n^3$

$a = 4, b = 2, c = 1, k = 3 ==> 4 < 2^3 ==> a < b^k ==>$ Case 3 of Master Theorem $==> T(n) = O(n^k)$ $==> T(n) = O(n^3)$.

$T(n) = 17T(n/4) + n^2$

$a = 17, b = 4, c = 1, k = 2 ==> 17 > 4^2 ==> a > b^k ==>$ Case 1 $==> T(n) = O(n^{log_b a}) ==>$ $T(n) = O(n^{log_4 17}) ==> T(n) = O(n^{2.0437})$

$T(n) = 9T(n/3) + n^2$

$a = 9, b = 3, c = 1, k = 2 ==> 9 = 3^2 ==> a = b^k ==>$ Case 2 $==> T(n) = O(n^k logn) ==>$ $T(n) = O(n^2 logn)$

$T(n) = T(\sqrt{n}) + 1$

We need to change variables.

Let $n = 10^m$, so $log_{10}(n) = m$.

$T(n) = T(10^m) = T(10^{m/2}) + 1 = T(10^{log_{10} n/2}) + 1 = T(log_{10} n/2 \times log_{10} 10) + 1 = T(m/2) + 1$

$a = 1, b = 2, c = 1, k = 0 ==> 1 = 2^0 ==> a = b^k ==>$ Case 2 $==> T(m) = O(m^k logm) ==>$ $T(m) = O(logm) <==> T(n) = O(log(logn))$.

6.

(see mergesort.py)