Problem 1. Using the formal definition of big-O, prove that for \(0 < a < b\), \(b^n\) is not \(O(a^n)\).

Problem 2. We can think of big-O as a relation on functions; prove that this relation is transitive. That is, prove that if \(f(n)\) is \(O(g(n))\) and \(g(n)\) is \(O(h(n))\) then \(f(n)\) is \(O(h(n))\). (Use the formal definition of big-O directly; do not appeal to general arguments about which functions must grow faster than which others. If you are stuck, look at worksheet question 3b for a similar problem (and our solution to it).)

Problem 3. Recall that the Fibonacci sequence 0, 1, 1, 2, 3, 5, ... can be defined recursively as follows:

\[
    f_n = \begin{cases} 
    n & \text{if } n \leq 1 \\
    f_{n-1} + f_{n-2} & \text{otherwise}
    \end{cases}
\]

You may use without proof\(^1\) that it has the following closed form:

\[
    f_n = \frac{\phi^n - (1 - \phi)^n}{\sqrt{5}}
\]

where \(\phi\) is the “golden ratio” \(\frac{1 + \sqrt{5}}{2} \approx 1.62\). You may also use this without proof:

\[
    \sum_{i=0}^{n} f_n = f_{n+2} - 1
\]

Consider the following algorithm, presented in pseudocode, which computes the Fibonacci sequence through a naive recursive method:

```plaintext
Naive Fibonacci
1. fib(n): // n >= 0
2.   if n <= 1:
3.     return n
4.   otherwise:
5.     return fib(n-1) + fib(n-2)
```

a) What is the run time of fib in terms of \(n\)? State any assumptions you make about how long different parts of this algorithm take. \(^2\)

b) Come up with a much faster algorithm for this task. What is its run time?

\(^1\) A hint for if you want to try proving this: use induction and note that \(\phi^2 = \phi + 1\)

\(^2\) Hint: come up with a recurrence and then use unrolling to get a closed form. 

\(\phi(2^n)\) is not a correct answer.