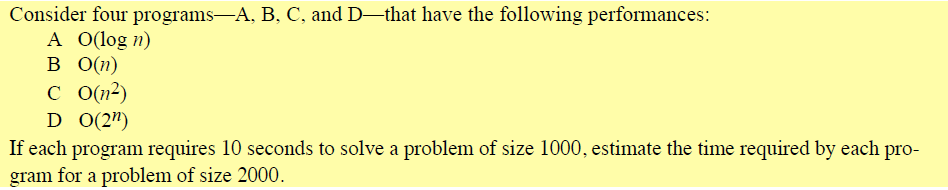
CS146Section8Lab2 Name: Michael Huang Due Date: February 18th, 2016

Ex-1:



A: O(log(n))

c \* g(n) = t. c \* log(1000) = 10s. c = 10/log(1000) = 10/3.

10/3 \* log(2000) = **11.0034333189s** ~ **11s**.

B: O(n)

c \* g(n) = t. c \* 1000 = 10s. c = 0.01.

0.01 \* 2000 = **20s**.

C: O(n2)

c \* g(n) = t. c \* 10002 = 10s. c = 10/10002.

10/10002 \* 20002 = 10 \* 20002 / 10002 = 10 \* 4 = **40s**.

D: O(2n)

c \* g(n) = t. c \* 21000 = 10s. c = 10/21000.

10/21000 \* 22000 = **1.07150860718626732094842504906e+302 seconds**, or an estimated **3.39773150426898561e+292 centuries**.

Ex-2:



Answer:

Using the formula f(n) <= c\*g(n) 🡪 f(n) = O(g(n)) where n >= n0, we can prove this.

Let f(n) = 6n2 + 3, and g(n) = n2. Therefore, 6n2 + 3 <= c\*n2.

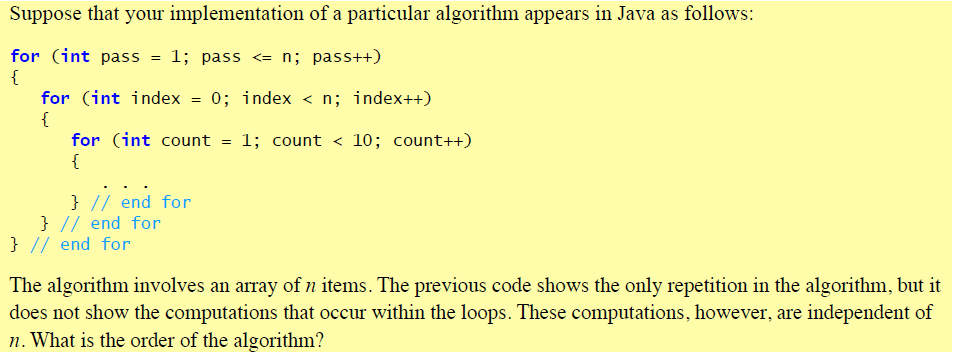
6n2 + 3 <= c\*n2 🡪 3 <= c\*n2 – 6n2 🡪 3 <= (c - 6)n2 🡪 3/(c - 6) <= n2.

If c = 7, then 3/(c - 6) <= n2 is true for all n >= sqrt(3), but mainly n >= 2. n0 = 2.

6n2 + 3 <= 7n2 is true for any n >= 2.

Thus, f(n) = 6n2 + 3 <= 7(n2) 🡪 6n2 + 3 = O(n2) for all n >= 2.

Ex-3:



Answer: The outer two loops are n\*n iterations of loops. Thus starting off, it’s O(n2). The outermost loop has a total of n iterations, the inner loop has a total of n iterations, and the innermost for loop has a total of 9 iterations. Since all of them are nested, multiply each loop’s number of iterations: n \* n \* 9 = 9n2.

Use f(n) <= c\*g(n) 🡪 f(n) = O(g(n)) where n >= n0 to prove 9n2 is O(n2).

Let f(n) = 9n2 and g(n) = n2. Therefore, 9n2 <= c\*n2.

9n2 <= c\*n2 🡪 0 <= c\*n2 – 9n2 🡪 0 <= (c – 9)n2.

If c >= 9, 0 <= c – 9 holds true for any n.

9n2 <= 9n2, which is true for any n.

Thus, f(n) = 9n2 <= 9n2 🡪 9n2 = O(n2).

Ex-4: Write Fibonacci algorithm both recursively and iteratively. Run both algorithm for different values of n and compare the running times. Show the complexity of solving this algorithm recursively and discuss if it is a good solution to write the code recursively.

Code:

public class Fibonacci {

public static long recursive(int n) {

if (n <= 1) {

return 1;

} else {

return recursive(n - 1) + recursive(n - 2);

}

}

public static long iterative(int n) {

if (n <= 1) {

return 1;

}

long sum = 0;

long previous = 0;

long current = 1;

while (n > 0) {

sum = previous + current;

previous = current;

current = sum;

n--;

}

return current;

}

public static void main(String[] args) {

System.out.println("All times are in System.nanoTime().");

long start = System.nanoTime();

System.out.println(recursive(10) + " | recursive(10)");

long end = System.nanoTime();

System.out.println(end - start + " | recursive(10)'s time");

start = System.nanoTime();

System.out.println(iterative(10) + " | iterative(10)");

end = System.nanoTime();

System.out.println(end - start + " | iterative(10)'s time");

start = System.nanoTime();

System.out.println(recursive(30) + " | recursive(30)");

end = System.nanoTime();

System.out.println(end - start + " | recursive(30)'s time");

start = System.nanoTime();

System.out.println(iterative(30) + " | iterative(30)");

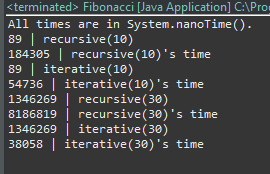
end = System.nanoTime();

System.out.println(end - start + " | iterative(30)'s time");

}

}

Console output:



Through timing the runtimes, it is obvious that the iterative method is more efficient. The iterative method is O(n) because there is only 1 while loop and its number of loops is equivalent to the number of terms. However, the recursive isn’t so.

The recursive method is O(2n). Take the base case n <= 1. Then the runtime would be O(1).

The number of steps when n > 1 grows exponentially as each call will call itself twice. Recursively, those two calls will call themselves twice each, resulting in four calls.

The number of calls are T(n) = T(n – 1) + T(n – 2), plus 1 more when n <= 2.

This recursive call is exponential, making 2 new calls for itself for every call. Therefore, the complexity of the recursive algorithm is O(2n).

Ex-5: Solve factorial recursively and show its order of complexity and prove it.

Code:

public class RecursiveFactorial {

public static long factorial(int n) {

if (n <= 1) {

return 1;

} else {

return n \* factorial(n - 1);

}

}

public static void main(String[] args) {

System.out.println(factorial(5));

}

}

The order of complexity is O(n). For the base case 1, it would be O(1). For n > 1, the number of recursive calls is dependent of n. Multiplying n by factorial(n – 1) is another step. So for every call from n to 1, which is a total of n – 1 calls, there are 2 steps. The total number of steps would be 2(n – 1) and an additional step when n = 1 so the total is 2n – 1.

2n – 1 is O(n), which can be proven with f(n) <= c\*g(n) 🡪 f(n) = O(g(n)) where n >= n0.

Let f(n) = 2n – 1, and g(n) = n. Therefore, 2n – 1 <= c\*n.

2n – 1 <= c\*n 🡪 -1 <= c\*n – 2n 🡪 -1 <= (c – 2)n 🡪 -1/(c – 2) <= n.

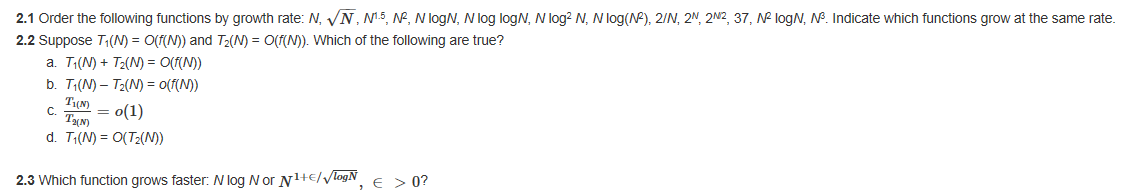
When c >= 3, -1/(c – 2) <= n is true for all n >= 0. n0 = 0.

2n – 1 <= 3n is true for any n >= 0.

Thus, 2n – 1 <= 7n 🡪 2n – 1 = O(n) for any n >= 0, which is practically any input.

In conclusion, a recursive factorial method has the order complexity of O(n).

Ex-6:



2.1: Order:

2/N < 37 < √N < N < Nlog(log(N)) < Nlog(N) <= Nlog(N2) < Nlog2(N) <N1.5 < N2 < N2log(N) < N3 < 2N/2 < 2N.

Nlog(N) and Nlog(N2) should grow at the same rate, because Nlog(N2) = 2Nlog(N), which is c\*Nlog(N) where c = 2. By the definition of Big-Oh, they should grow at the same rate.

2.2: T1(N) = O(f(N)) and T2(N) = O(f(N)).

A: T1(N) + T2(N) = O(f(N))

True, because rule 1 of Big-Oh states that T1(N) + T2(N) = O(max(T1(N), T2(N))), where T1(N) and T2(N) are both O(f(N)).

B: T1(N) – T2(N) = o(f(N))

False. Counter-example: say T1(N) = 3N2 and T2(N) = N2. f(N) in this case is N2 because f(N) is max(T1(N), T2(N)). 3N2 – 2N2 = N2. N2 is obviously O(N2) because N2 <= c\*N2 for any c. N2 is also Ω(N2) because Big-Omega states that if there is a constant c such that:

f(N) >= c\*g(N) 🡪 f(N) = Ω(g(N)). When c = 1, N2 <= 1\*N2 is true, so it is also Ω(N2).

Since 3N2 – 2N2 = N2 is both O(N2) and Ω(N2), it is also Θ(N2).

The rule of o(f(N)) states that T(N) is o(f(N)) if and only if T(N) = O(f(N)) and T(N) ≠ Θ(f(N)). Since T1(N) – T2(N) is Θ(N2), T1(N) – T2(N) is not o(f(N)).

C: T1(N)/T2(N) = o(1)

False. Counter-example: say T1(N) = N3 and T2(N) = N. f(N) in this case is N3 because f(N) is max(T1(N), T2(N)). Therefore T1(N)/T2(N) = N3/N = N2. N2 is always bigger than o(1).

D: T1(N) = O(T2(N))

False. Counter-example: say T1(N) = N and T2(N) = 1. f(N) in this case is Nbecause f(N) is max(T1(N), T2(N)). N in this case is ≠ O(1). N is O(N).

2.3: Which function grows faster: Nlog(N) or N1 + ∈/√log(N) where ∈ > 0?

Nlog(N) grows faster. Assume N1 + ∈/√log(N) <= Nlog(N).

N1 + ∈/√log(N) < Nlog(N) 🡪 N1 + ∈/√log(N) / N < Nlog(N) / N 🡪 N∈/√log(N) < log(N).

N∈/√log(N) < log(N) 🡪 10log(N) \* (∈/√log(N)) < 10log(N) 🡪10√log(N) \* ∈ < N 🡪

log(10√log(N) \* ∈) < log(N) 🡪 ∈\*√log(N) < log(N).

To prove ∈\*√log(N) < log(N), use T(N) = o(f(N)).

∈\*√log(N) < c\*log(N) 🡪 T(N) = o(f(N)). c is any positive constant, which can be set as ∈.

Thus, ∈\*√log(N) < ∈\*log(N) 🡪 ∈\*√log(N) = o(log(N)).

Therefore, N1 + ∈/√log(N) grows slower than Nlog(N).