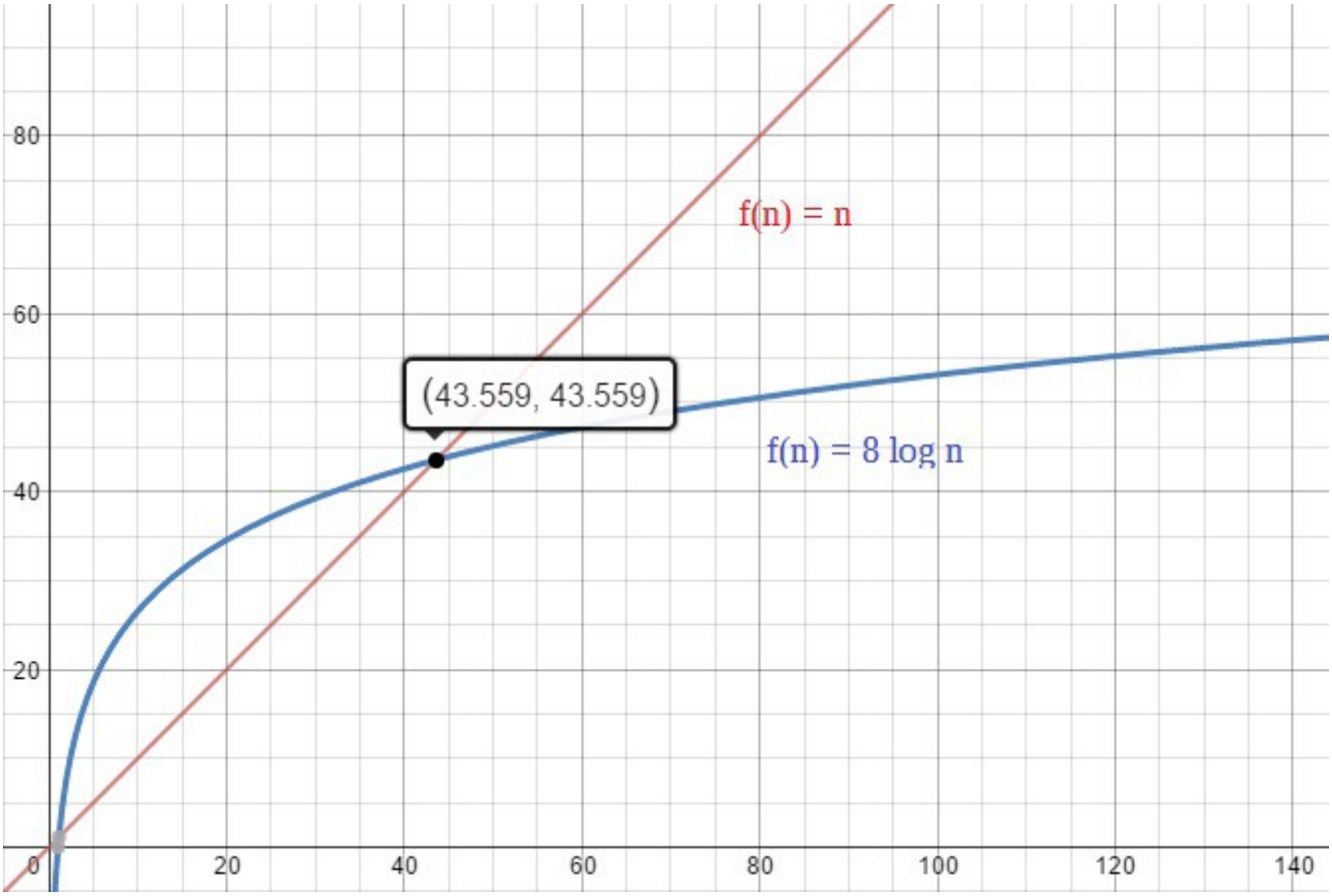
Conrad Lewin
CS325
4/3/2016

1) Insertion sort will beat merge sort as long as $8n^2$ is less than the value of $64n\log_2 n$. To determine which value of *n* will invalidate this inequality, we must find the point of intersection for these two functions.

$$8n^2 \leq 64n\log_2 n$$
$$n \leq 8\log_2 n$$

Plotting both of the resulting functions yields the following:



We can see from the graph, that the point of intersection occurs when n is approximately 43.6. Assuming the number of input items cannot be fractional, the effective input size in which merge sort beats insertion sort can be understood to be the ceiling of 43.6 or $n = 44$. In this case, then, insertion sort will be more effective than merge sort for all $n < 44$.

2) A function $f(n)$ that can accomplish a task of a given size *n* within a particular time frame *t* in microseconds can be expressed as $f(n) \leq t$.

Consider $f(n) = \log_2 n$ and *t* is 1 second. To determine the largest value of *n* $f(n)$ can solve within *t*, we must find the value of *n* that equals the value of *t*.

$f(n) = 1$ *second*
$\log_2 n = 10^6$ *microseconds*
$n = 2^{(10^6)}$
$n = 2^{1000000}$

Thus, the largest size problem a function $f(n)$ can tackle in 1 second is $2^{1000000}$. The same principle applies for other values of $f(n)$ and *t*:

1 second = 10^6 microseconds
1 minute = 6e+7 microseconds
1 hour = 3.6e+9 microseconds
1 day = 8.64e+10 microseconds
1 month = 2.628e+12 microseconds
1 year = 3.154e+13 microseconds
1 century = 3.154e+15 microseconds

	1 second	**1 minute**	**1 hour**	**1 day**	**1 month**	**1 year**	**1 century**
$\log_2 n$	$2^{1000000}$	2^{6e+7}	$2^{3.6e+9}$	$2^{8.64e+10}$	$2^{2.628e+12}$	$2^{3.154e+13}$	$2^{3.154e+15}$
$\sqrt{n}$	10^{12}	3.6e+15	≈1.3e+19	≈7.4e+21	≈6.9e+24	≈9.9e+26	≈9.9e+30
n	10^6	6e+7	3.6e+9	8.64e+10	2.628e+12	3.154e+13	3.154e+15
$n \log_2 n$	62746	2801417	≈1.3e+8	≈2.7e+9	≈7.1e+10	≈7.9e+11	≈6.8e+13
n^2	1000	7745	60000	293938	≈1.6e+6	≈5.6e+6	≈5.6e+7
n^3	100	391	1532	4420	13799	31595	146651
2^n	19	25	31	36	41	44	51
$n!$	9	11	12	13	15	16	17

3) Use mathematical induction to show that when *n* is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2, \text{ if } n = 2 \\ 2T\left(\frac{n}{2}\right) + n, \text{ if } n = 2^k, \text{ for } k > 1 \end{cases}$$

is $T(n) = n\log_2 n$.

Base Case:
From the definition of $T(n)$, we can see that the base case occurs when $n = 2$ or when $k = 1$. Thus,

$T(n) = n\log_2 n$
$T(2) = 2\log_2 2$
$T(2) = 2(1)$
$T(2) = 2$

In this case when $n = 2$, $T(n) = 2$ and this satisfies the definition of $T(n)$.

Inductive Hypothesis:
The inductive hypothesis is included in the definition of $T(n)$, namely that for all powers of 2 greater than 2, $2T\left(\frac{n}{2}\right) + n = n\log_2 n$.

Inductive Case:
Using substitution, algebra and the rules of logarithms,

$$2T\left(\frac{n}{2}\right) + n = n\log_2 n$$
$$2\left(\frac{n}{2}\right)\log_2\left(\frac{n}{2}\right) + n = n\log_2 n$$
$$n(\log_2 n - \log_2 2) + n = n\log_2 n$$
$$n(\log_2 n - 1) + n = n\log_2 n$$
$$n\log_2 n - n + n = n\log_2 n$$
$$n\log_2 n = n\log_2 n$$

Having proved both the base case and the inductive step, the original supposition holds.

4a) $f(n) = n^{0.75}$; $g(n) = n^{0.5}$

$$\lim_{n\to\infty} \frac{n^{0.75}}{n^{0.5}} = \frac{n^{0.25}}{1} = \infty$$

The numerator will always be larger than the denominator and so as n grows it will approach infinity, thus $f(n) = \Omega(g(n))$.

4b) $f(n) = n$; $g(n) = (\log_2 n)^2$

$$\lim_{n\to\infty} \frac{n}{(\log_2 n)^2} = \infty$$

The growth factor for $f(n)$ will form the upper bound of $g(n)$ so as n grows the end behavior of $g(n)$ will approach infinity. Thus $f(n) = \Omega(g(n))$.

4c) $f(n) = \log_2 n$; $g(n) = \log_2 n$

$$\lim_{n\to\infty} \frac{\log_2 n}{\log_2 n} = 1$$

As n grows both the numerator and the denominator will be the same number, and so will always approach 1, thus $f(n) = \Theta(g(n))$.

4d) $f(n) = e^n;\ g(n) = 2^n$

$$\lim_{n\to\infty} \frac{e^n}{2^n} = \infty$$

Since *e* is larger than 2, as *n* grows large, the growth factor of the numerator will not be bound by the value of the denominator. Thus, $f(n) = \Omega(g(n))$.

4e) $f(n) = 2^n;\ g(n) = 2^{n-1}$

$$\lim_{n\to\infty} \frac{2^n}{2^{n-1}} = \frac{2^n}{\frac{2^n}{2}} = 2$$

As *n* grows, both functions will grow at a constant rate. Thus, $f(n) = \Theta(g(n))$.

4f) $f(n) = 2^n;\ g(n) = 2^{2n}$

$$\lim_{n\to\infty} \frac{2^n}{2^{2n}} = \frac{2^n}{(2^2)^n} = \frac{2^n}{4^n} = 0$$

$g(n)$ will grow faster then $f(n)$, and defines the bound for f(n). Thus, $f(n) = O(g(n))$.

4g) $f(n) = 2^n;\ g(n) = n!$

$$\lim_{n\to\infty} \frac{2^n}{n!} = 0$$

For $n \geq 4$, $g(n)$ will grow faster than $f(n)$. Thus, $f(n) = O(g(n))$.

4h) $f(n) = n\log_2 n;\ g(n) = n\sqrt{n}$

$$\lim_{n\to\infty} \frac{n\log_2 n}{n\sqrt{n}} = \frac{\log_2 n}{\sqrt{n}} = 0$$

For $n \geq 16$, $g(n)$ will grow faster than $f(n)$. Thus, $f(n) = O(g(n))$.

5) One method for finding a sum *x* of two numbers *p* and *q* in an array *S* that achieves a complexity of Θ(n log n) involves dividing the algorithm into two parts, namely sorting the array and then summing pairs of integers until either *x* is found or all relevant pairings have been exhausted. The complexity of each individual part cannot exceed a complexity of Θ(n log n). Since the most apparent solution for summing pairs involves, at worst, iterating over each number in the array, this part of the algorithm will be bound by O(n). This fact necessitates the need for a sorting algorithm that is tightly bound by Θ(n log n) so that the overall complexity of the given algorithm will be Θ(n log n) + O(n) = Θ(n log n). Some candidates for such a sort include quickSort, mergeSort and heapSort. The following psuedocode uses mergeSort to achieve this end.

Psuedocode:

```
SumPairs(S, x)
    mergeSort(S)

    j ← 0
    k ← S.length – 1

    while j != k
        sum ← S[j] + S[k]
        if sum < x
            j ← j + 1
        else if sum > x
            k ← k – 1
        else
            return S[j], S[k]

    return NIL
```

Explanation:

Once the array is sorted, two index values *j* and *k* are set to point to the beginning and the end of the array respectively. A while loop is then executed to sum the values stored at the current indices pointed to by *j* and *k*, terminating when either S[j] + S[k] = *x* or *j* = *k*. The while loop begins by calculating the sum and then performs a comparison of this sum with *x*. If the sum is less then *x*, then *j* is incremented by 1, if the sum is greater than *x*, then *k* is incremented by 1 and if the sum is equal to *x*, then both S[j] and S[k] are returned. In the event that the loop is allowed to terminate before reaching its return statement, NIL is returned to indicate that there are no pairs in *S* whose sum is *x*.

Sample implementation:

Let set $S = \{12, 3, 4, 15, 11, 7\}$ and $x = 20$

step 1) $S = \{3, 4, 7, 11, 12, 15\}$
step 2) $j = 0$, $k = 5$
step 3.1) Enter while loop
- a) 0 != 5
- b) 3 + 15 = 18
- c) 18 < 20
- d) $j = 1$, $k = 5$

3.2) Enter while loop
- a) 1 != 5
- b) 4 + 15 = 19
- c) 19 < 20
- d) $j = 2$, $k = 5$

3.3) Enter while loop
- a) 2 != 5
- b) 7 + 15 = 22
- c) 22 > 20
- d) $j = 2$, $k = 4$

3.2) Enter while loop
- a) 2 != 4

b) 7 + 12 = 19
c) 19 < 20
d) $j = 3$, $k = 4$

3.3) Enter while loop
a) 3 != 4
b) 11 + 12 = 22
c) 22 > 20
d) $j = 3$, $k = 3$

3.4) Loop test fails, 4 != 4 is false. Loop exits.

4) Return NIL

6a) If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$

Proof:
By the definition of $O(g(n))$, $0 \le f_1(n) \le c_1(g_1(n))$ and $0 \le f_2(n) \le O(g_2(n))$ for $n_0 > 0$ and some constant c_j. We must show that $0 \le f_1(n) + f_2(n) \le c_1 g_1(n) + c_2 g_2(n)$.

Assuming $f_1(n) \le c_1(g_1(n))$, we can add $f_2(n)$ to both sides of the inequality without affecting the meaning of the signs. Thus,

$$f_1(n) + f_2(n) \le c_1(g_1(n)) + f_2(n)$$

Since $f_2(n)$ is in the set of $O(g_2(n))$, we can substitute $f_2(n)$ for $c_2 g_2(n)$ and determine whether or not there is a constant that maintains the truth of the inequality

$$f_1(n) + f_2(n) \le c_1(g_1(n)) + c_2(g_2(n))$$

Thus, let $c_3 = max(c_1, c_2)$ so that $f_1(n) + f_2(n) \le c_3[g_1(n) + g_2(n)]$.

By the definition of $O(g(n))$, it follows that since $f_1(n)$ and $f_2(n)$ are in the sets of $c_1(g_1(n))$ and $c_2(g_2(n))$ respectively, multiplying each function by the largest possible constant c_3 will produce a result that is inevitably greater than or equal to the sum of the functions $f_1(n)$ and $f_2(n)$ in which only one of those functions is multiplied by c_3.

6b) If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $\frac{f_1(n)}{f_2(n)} = O\left(\frac{g_1(n)}{g_2(n)}\right)$

Proof:
By the definition of $O(g(n))$, $0 \le f_1(n) \le c_1(g_1(n))$ and $0 \le f_2(n) \le O(g_2(n))$ for $n_0 > 0$ and some constant c_j. We must show that $0 \le \frac{f_1(n)}{f_2(n)} \le \frac{c_1 g_1(n)}{c_2 g_2(n)}$. However, consider this example:

Let $f_1(n) = 6n$ and $f_2(n) = 3n$. Each function is an asymptotically positive polynomial function of degree 1, therefore both functions are $\Theta(n^1)$ and in turn are polynomially bounded by $O(n)$. Thus, by substitution and basic algebra,

$$\frac{f_1(n)}{f_2(n)} \leq \frac{c_1 g_1(n)}{c_2 g_2(n)} = \frac{6n}{3n} \leq \frac{c_1 g_1(n)}{c_2 g_2(n)} = 2 \leq \frac{c_1 g_1(n)}{c_2 g_2(n)}$$

Although c_j is constant, it is not given, and so it is conceivable that c_2 will not always be greater than c_1. Since both $f_1(n)$ and $f_2(n)$ are bound by $O(n)$, the inequality can be further demystified as such:

$$2 \leq \frac{c_1 g_1(n)}{c_2 g_2(n)} = 2 \leq \frac{c_1 n}{c_2 n} = 2 \leq \frac{c_1}{c_2}$$

Now it is clear that any $c_2 \geq c_1$ will invalidate the truth of the inequality. This is a counter example to the original supposition, thus proving it false.

6c) $max(f_1(n), f_2(n)) = \Theta(f_1(n) + f_2(n))$

It follows from the definition of Big Theta that

$$0 \leq c_1(f_1(n) + f_2(n)) \leq max(f_1(n), f_2(n)) \leq c_2(f_1(n) + f_2(n)) \text{ for all } n \geq n_0$$

It is obvious from this inequality that no matter which $f_p(n)$ is the max, if $c_2 > 0$, then $f_p(n)$ will be less than or equal to the quantity $c_2(f_1(n) + f_2(n))$. The left side of the inequality, however, requires a constant that is less than 1 but greater than or equal to 0. Thus, if $c_1 = 0.5$, then

$$0 \leq 0.5(f_1(n) + f_2(n)) \leq max(f_1(n), f_2(n)) \leq c_2(f_1(n) + f_2(n)) \text{ for all } n \geq n_0$$

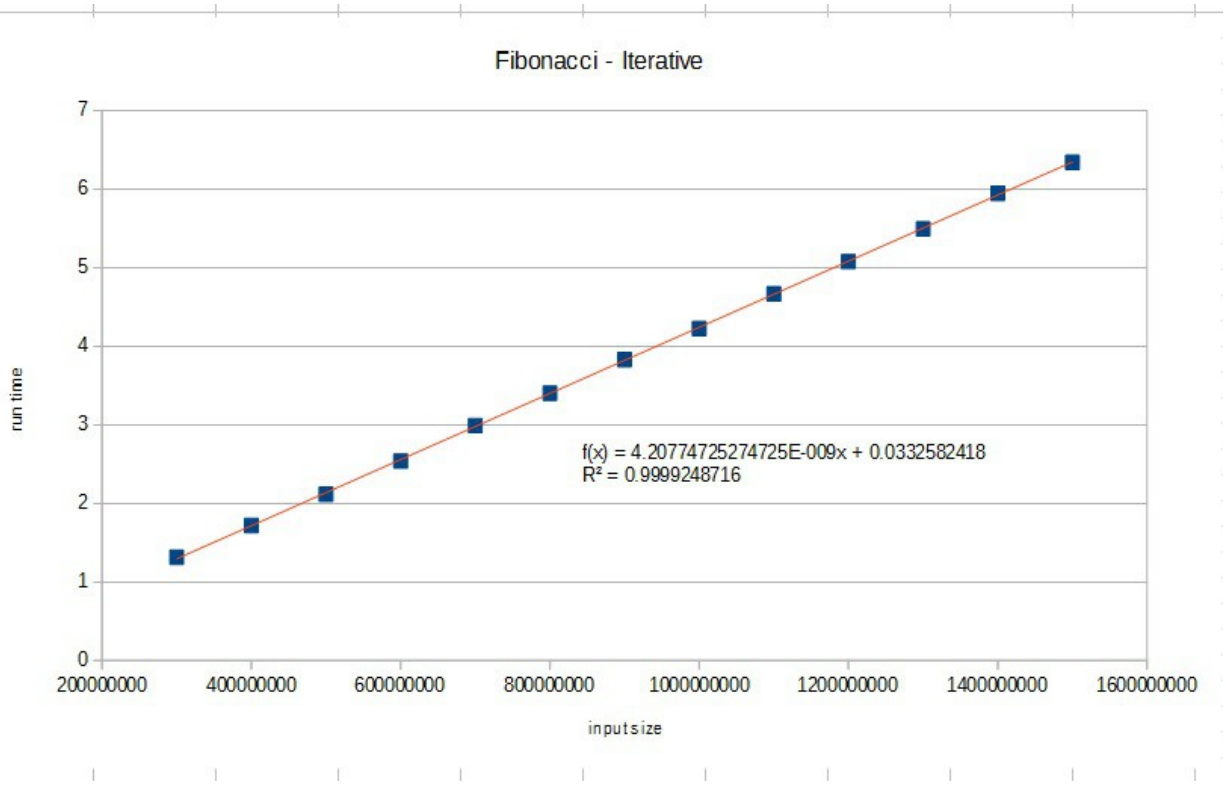
holds true. Therefore, the original supposition is true for all $n \geq n_0$, $0 \leq c_1 < 1$ and $c_2 > 0$. Otherwise, if $c_1 > 1$, then $max(f_1(n), f_2(n)) = O(f_1(n) + f_2(n))$.

7a) Code for iterative and recursive algorithms for calculating the Fibonacci Sequence:

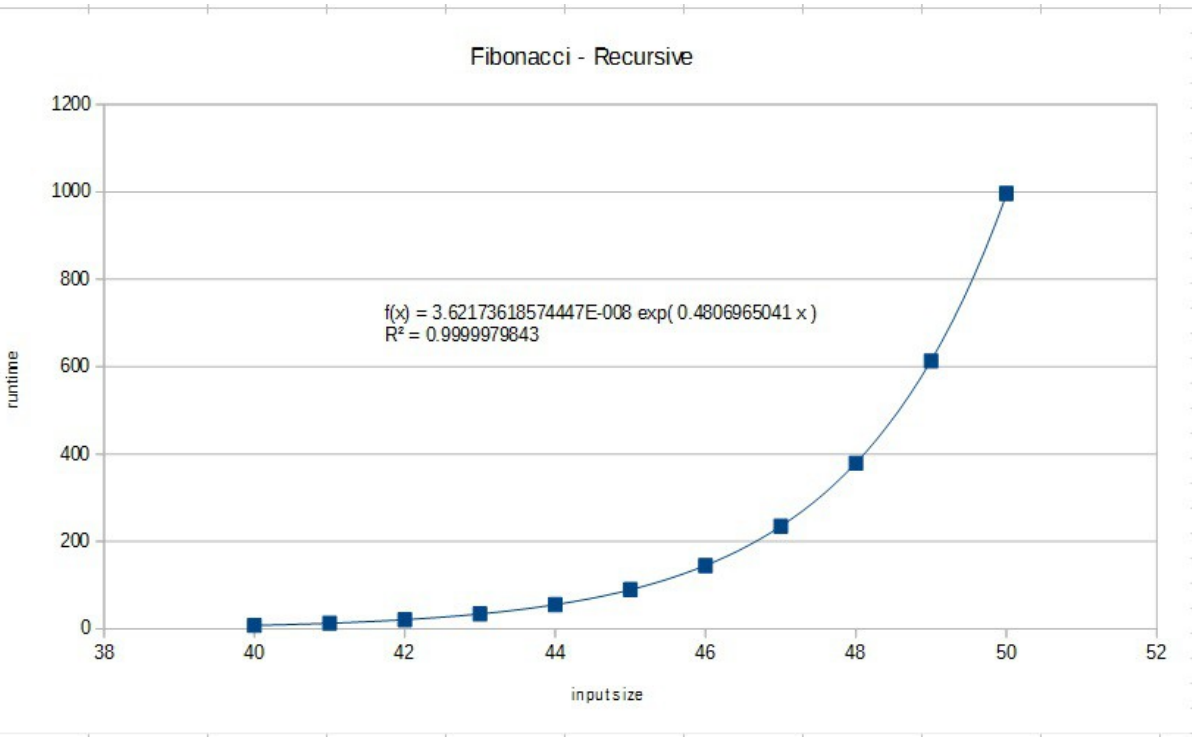
```
1
2   unsigned long long itrFib(long long n)
3   {
4       unsigned long long fib = 0;
5       unsigned long long a = 1;
6       unsigned long long t = 0;
7
8       for (long long i = 0; i < n; i++)
9       {
10          t = fib + a;
11          a = fib;
12          fib = t;
13      }
14
15      return fib;
16  }
17
18  unsigned long long recFib(int n)
19  {
20      if (n == 0)
21          return 0;
22      else if (n == 1)
23          return 1;
24      else
25          return recFib(n - 1) + recFib(n - 2);
26  }
```

7b - c) Run times for each algorithm:

Iterative	
n	time
300000000	1.316
400000000	1.72
500000000	2.116
600000000	2.539
700000000	2.99
800000000	3.403
900000000	3.83
1000000000	4.225
1100000000	4.666
1200000000	5.077
1300000000	5.493
1400000000	5.947
1500000000	6.341

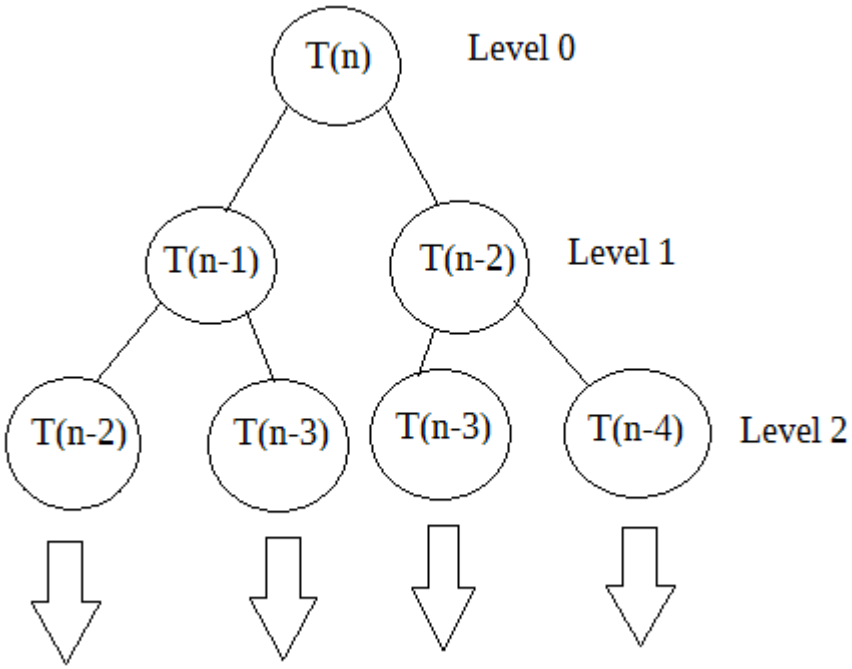


Recursive	
n	time
40	8.094
41	13.188
42	21.243999
43	34.34
44	55.437
45	89.847
46	145.074005
47	235.035995
48	379.138
49	613.052002
50	996.517029



7d) The function that best fits the iterative solution is linear in nature, that is $f(n) = O(n)$, while the recursive algorithm is best defined by an exponential function or $f(n) = O(2^n)$. The iterative algorithm is the most efficient solution with the linear equation tied to the given input sizes equal to $f(n) \approx 4.2\text{E-}9x + 0.03$.

The difference in run times for each algorithm lies in the manner in which each procedure calculates the sequence. The iterative solution uses a for loop that performs three constant time operations *n* number of times, that is $T(n) = O(1) + O(1) + O(1) + O(n) = O(n)$. On the other hand, the recursive solution must perform several redundant recursive calls in order to achieve the same result. This can most easily be seen by examining a section of the recursion tree:



Each node represents the recursive call in the algorithm's final return statement, and, since two calls are made with each return, the nodes in each level of the tree indicates how many calls must be resolved before the procedure exits. There are 2^i nodes at each level of the tree where *i* is the value of the relevant level, thus 2^k, where *k* is the height of the tree, recursive calls must be made to complete the algorithm, including a constant addition operation. In this way, the recursive algorithm to compute *n* Fibonacci numbers is

$$T(n) = 2T(2^n) + O(1)$$
$$T(n) = c_1(2^n) + c_1(2^n) + O(1)$$
$$T(n) = O(2^n) + O(2^n) + O(1)$$
$$T(n) = O(2^n)$$