Paul Winzer
CS 520
Hood College - Spring 2018

1. [COMPLETE]

$$\sum_{i=1}^{n} i^2 = \{1 + 4 + 9 + 16 + 25 + ... + n^2\}$$

Inductive proof that $\forall\ n \geq 1\ :\ \sum_{i=1}^{n} i^2 = \frac{1}{6}n(n+1)(2n+1)$

a. $\frac{1}{6}n(n+1)(2n+1) + (n+1)^2 = \frac{1}{6}(n+1)((n+1)+1)(2(n+1)+1)$
b. $\frac{1}{6}(n^2+n)(2n+1) + (n+1)^2 = \frac{1}{6}(n+1)(n+2)(2n+3)$
c. $\frac{1}{6}(2n^3 + 3n^2 + n) + n^2 + 2n + 1 = \frac{1}{6}(n^2+3n+2)(2n+3)$
d. $\frac{2n^3}{6} + \frac{3n^2}{6} + \frac{n}{6} + \frac{6n^2}{6} + \frac{12n}{6} + \frac{6}{6} = \frac{1}{6}(2n^3 + 9n^2 + 13n + 6)$
e. $\frac{2n^3}{6} + \frac{9n^2}{6} + \frac{13n}{6} + \frac{6}{6} = \frac{1}{6}(2n^3 + 9n^2 + 13n + 6)$
f. $\frac{1}{6}(2n^3 + 9n^2 + 13n + 6) = \frac{1}{6}(2n^3 + 9n^2 + 13n + 6)$ [both sides identical]
g. $... = \frac{2n^3}{6} + \frac{9n^2}{6} + \frac{13n}{6} + \frac{6}{6}$
h. $... = \frac{2n^3}{6} + \frac{3n^2}{6} + \frac{n}{6} + \frac{6n^2}{6} + \frac{12n}{6} + \frac{6}{6}$
i. $... = \frac{1}{6}(2n^3 + 3n^2 + n) + n^2 + 2n + 1$
j. $... = \frac{1}{6}(n^2+n)(2n+1) + (n+1)^2$
k. $... = \frac{1}{6}n(n+1)(2n+1) + (n+1)^2$ QED

2. [COMPLETE]

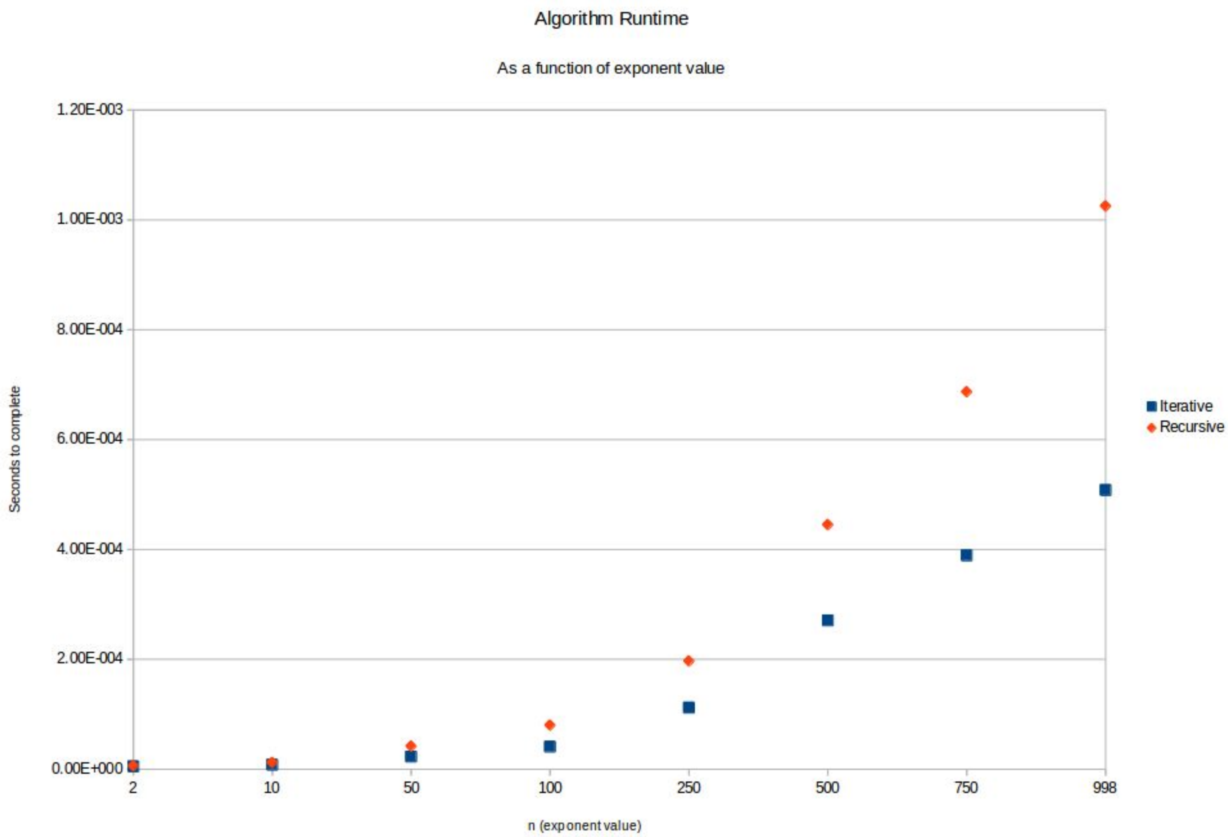
----------------BEGIN CODE----------------------

```
import time

# iterate raising x to the n
def pow_it(x, n):

	print 'Raising {0} to the {1}'.format(x,n)
	start = time.time()
	a = 1
	i = n
	if n == 0:
		return 1
	else:
		while i > 0:
			a *= x
			i -= 1
	stop = time.time()
	run_time = stop - start
	print 'Raised {0} to the {1} in {2}'.format(x,n,run_time)
	return a
```

```
# recursively raising x to the n
def pow_re(x,n,top=True):

      if top:
            print 'Raising {0} to the {1}'.format(x,n)
            start = time.time()

      a = x

      if n == 0:
            return 1
      else:
            a =  pow_re(a, n-1, False)
            if top:
                  stop = time.time()
                  run_time = stop - start
                  print 'Raised {0} to the {1} in {2}'.format(x,n,run_time)
            return a
```

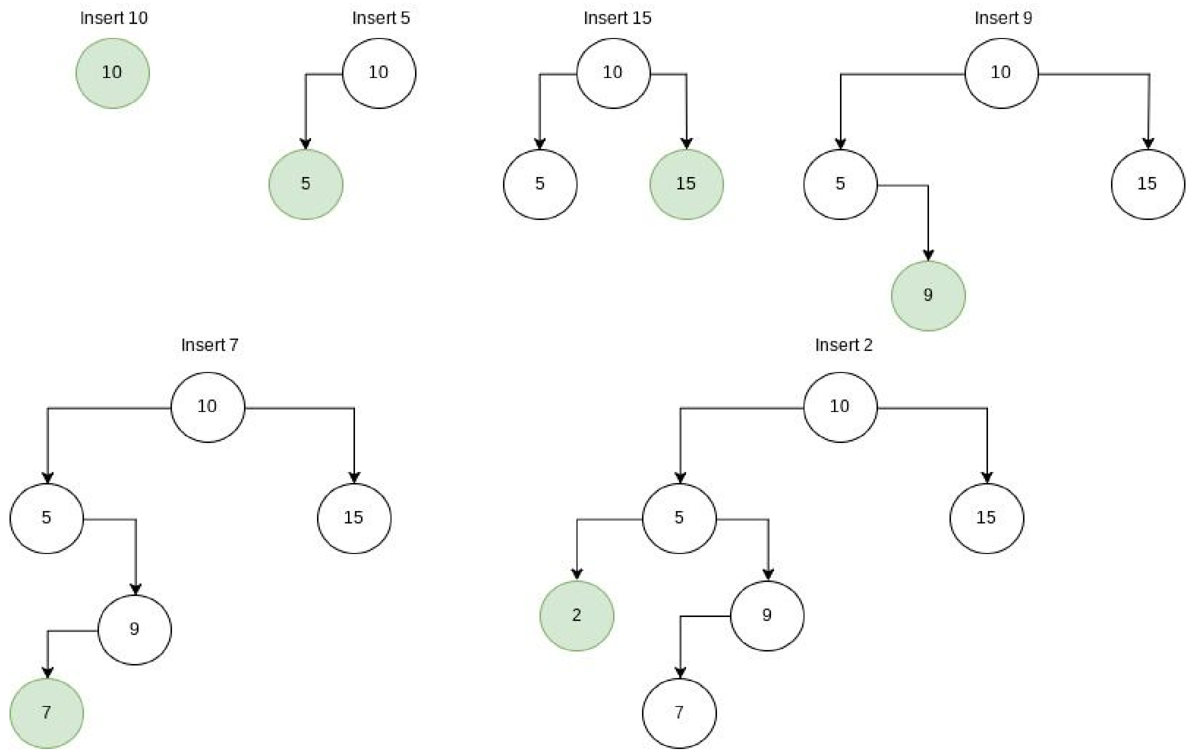
----------------END CODE----------------------
(code also included as pow_pdw.py in .zip)

x	n	Iterative time (sec)	Recursive time (sec)
5	1	5.01E-006	5.96E-006
5	10	8.11E-006	1.22E-005
5	50	2.29E-005	4.20E-005
5	100	4.10E-005	8.01E-005
5	250	0.0001120567	0.0001969337
5	500	0.0002708435	0.0004451275
5	750	0.0003890991	0.0006871223
5	998	0.00050807	0.0010251999
5	999	0.0005640984	Fatal, max recursion level reached



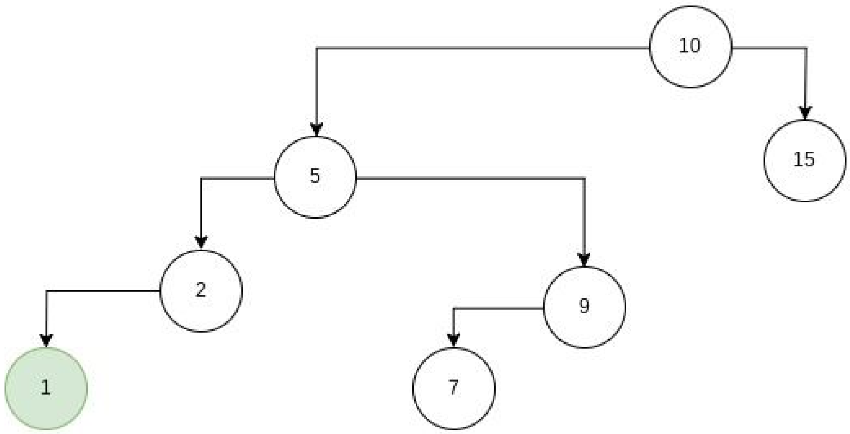
Both iterative and recursive algorithms run in O(n); it looks quadratic but the exponent size (x axis) grows unevenly from 0 to 998, though it grows evenly from 250 to 998, which is the best place to see the linear growth. Though the time complexities are the same, the recursive algorithm takes longer to complete on for all exponents tested. The recursive algorithm also had the disadvantage of topping out at an exponent of 998; it would not recurse more than that.

3. [COMPLETE] Using the BST definition that all element values are unique, the second 10 isn't inserted because there is already a 10 in the tree.

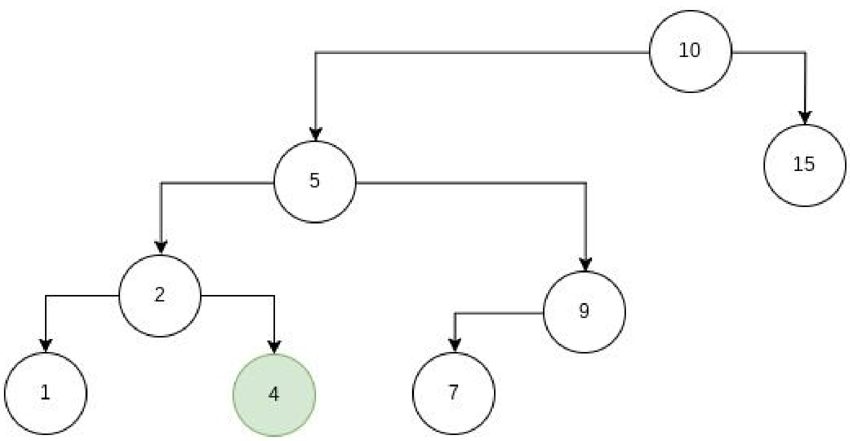


3. (continued)

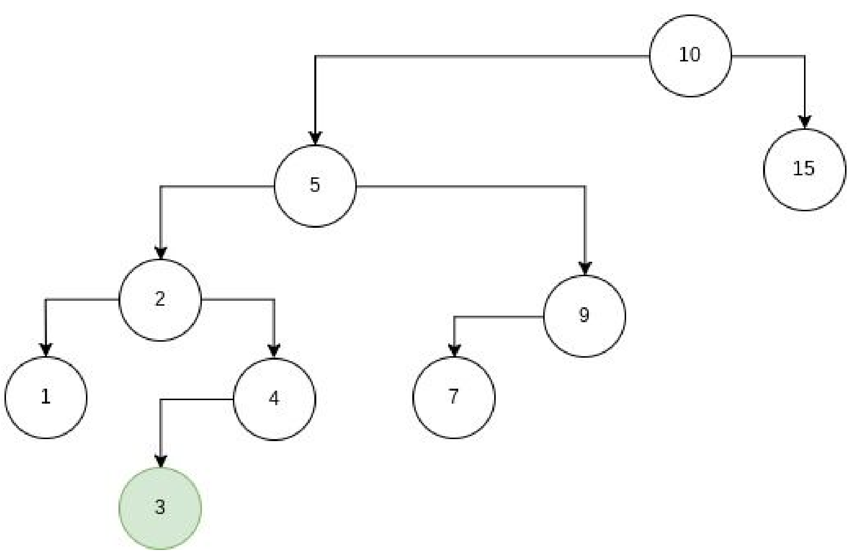
Insert 1



Insert 4

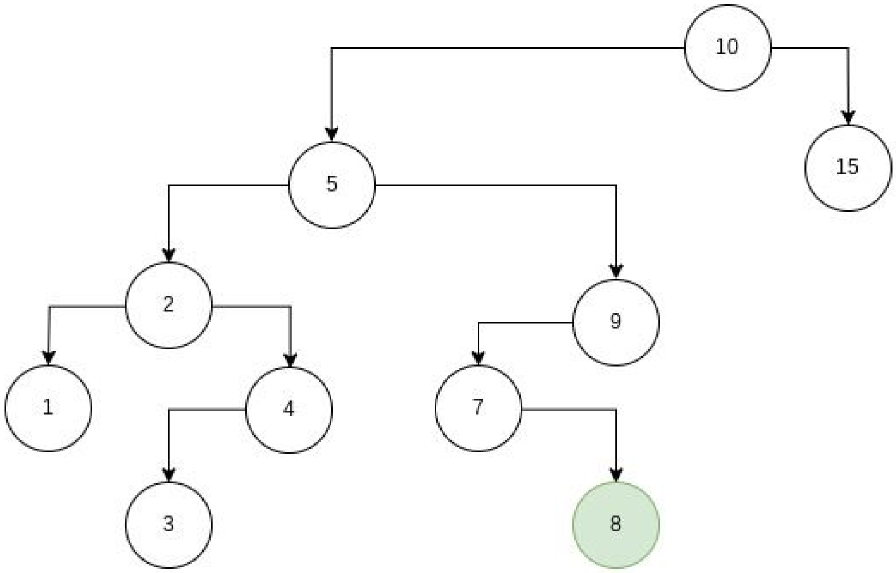


Insert 3

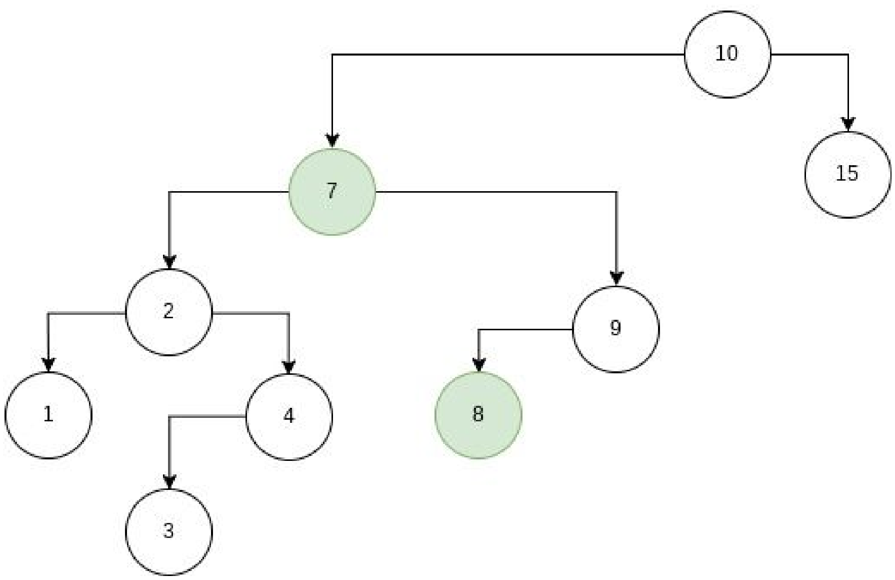


3. (finished)

Insert 8



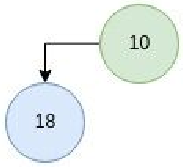
Delete 5



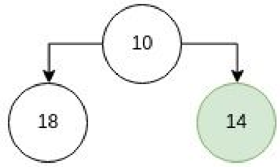
4. [COMPLETE]

Insert 18

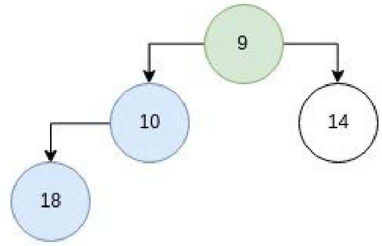
Insert 10



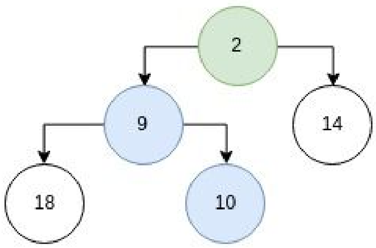
Insert 14



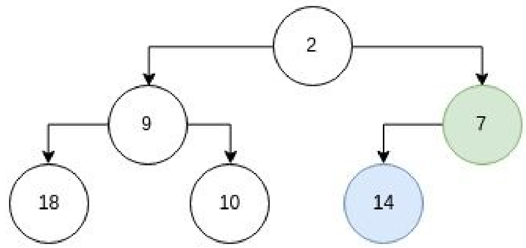
Insert 9



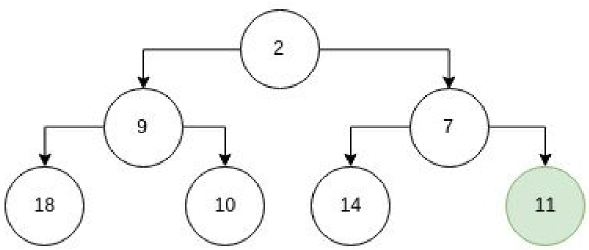
Insert 2



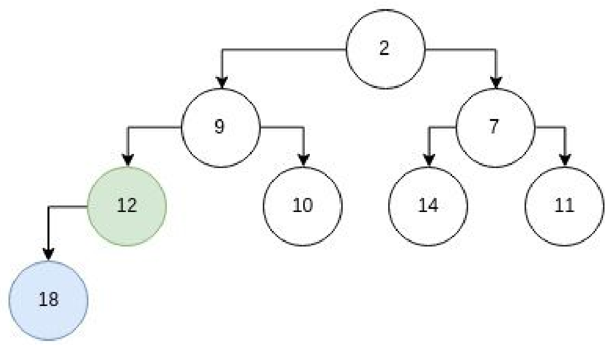
Insert 7



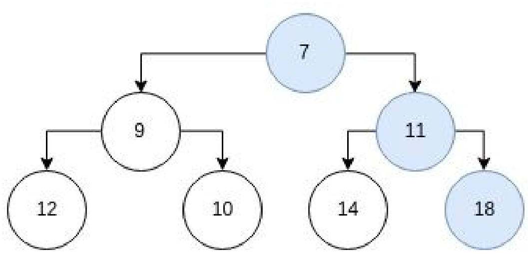
Insert 11



Insert 12



DelMin



5. [COMPLETE]

a. For height and depth, I print the values as I go, instead of returning them all in one one structure at the end. They also use 0-based indexing; leaf height = 0, root depth = 0.

```
NumLeaves(T):
  // no children means it's a leaf
  if (T → lchild is null) and (T → rchild is null):
    return 1
  // if it has only rchild, search that
  else if (T → lchild is null):
    return NumLeaves(T → rchild)
  // if it has only lchild, search that
  else if (T → rchild is null):
    return NumLeaves(T → lchild)
  // if it has two children, search both
  else:
    return NumLeaves(T → lchild) + NumLeaves(T → rchild)

HeightEveryNode(T):
  // no children means height == 0
  if (T → lchild is null) and (T → rchild is null):
    print 0
    return 0
  // if it has only rchild, it's 1 more than the height of that
  else if (T → lchild is null):
    h = 1 + HeightEveryNode(T → rchild)
    print h
    return h
  // if it has only lchild, it's 1 more than the height of that
  else if (T → rchild is null):
    h = 1 + HeightEveryNode(T → lchild)
    print h
    return h
  // if it has two children, it's 1 more than the height of the taller
  else:
    h = 1 + Max(HeightEveryNode(T → lchild), HeightEveryNode(T → rchild))
    print h
    return h
```

```
DepthEveryNode(T,D=0):
  // no children means max depth
  if (T → lchild is null) and (T → rchild is null):
    print D
    return D
  // if it has only rchild, it's 1 less than the depth of that
  else if (T → lchild is null):
    d = -1 + DepthEveryNode(T → rchild, D+1)
    print d
    return d
  // if it has only lchild, it's 1 less than the depth of that
  else if (T → rchild is null):
    d = -1 + DepthEveryNode(T → lchild, D+1)
    print d
    return d
  // if it has two children, it's 1 less than the depth of either
  // depth of both children are the same, so we only need one
  // but we must call both to find the depth of every node in tree
  else:
    d = -1 + DepthEveryNode(T → lchild, D+1)
    DepthEveryNode(T → rchild, D+1)
    print d
    return d

IsFull(T):
  // a node with no children does not rule out fullness
  if (T → lchild is null) and (T → rchild is null):
    return true
  // a node with exactly one child does rule out fullness
  else if (T → lchild is null) or (T → rchild is null):
    return false
  // if node has two children, see if subtrees are full
  else:
    return IsFull(T → lchild) and IsFull(T → rchild)
```

b. All these routines compute in O(n) time complexity. They all require at least one operation and at most two operations on every node representing a subtree. Returning the base case or returning the recursive call to a single subtree would be one operation, whereas calling down both subtrees would be two operations. Because all these routines require traversing every subtree and visiting every node, we cannot reduce the complexity to O(log n) like we can with BST search or inserts. And because the length of each basic operation does not depend on the number of nodes in the tree, we don't see complexity of O(n^2).

6. [COMPLETE] The asymptotic time complexity of a good algorithm to sum all the elements of an ‘n by n’ 2 dimensional matrix grows linearly with the number of inputs in the matrix. The actual number of inputs to the routine is some number m where $m = n^2$. Since the run time of the algorithm is a function of the number of inputs, and n is the square root of the number of inputs, we would say that the algorithm grows in O(m) time. If we were to say that the growth of runtime was O(n), that n would not be equal to the n in the ‘n by n’ size of the matrix, and that would be confusing.

7. [COMPLETE]

a) This function computes the n^{th} number in the Fibonacci series.

b) $$Fibonacci(n) = \{ \; 1 : n \leq 1 \\ Fibonacci(n-1) + Fibonacci(n-2) : n > 1 \; \}$$

c) There are 12 additions in $unknown(6)$.

The recurrence relation for the number of additions is:

$$FibAdds(n) = \{ \; 0 : n \leq 1 \\ FibAdds(n-1) + FibAdds(n-2) + 1 : n > 1 \; \}$$