I pledge that I have neither given nor received any unauthorized aid on this assignment.

Problem 1 [PART COMPLETE-TIME]

i. `http://www.proofwiki.org/wiki/Sum_of_Geometric_Progression#Proof_1`

ii. `http://www.proofwiki.org/wiki/Sum_of_Sequence_of_Squares#Proof_1`

Problem 2 [COMPLETE]

Solutions

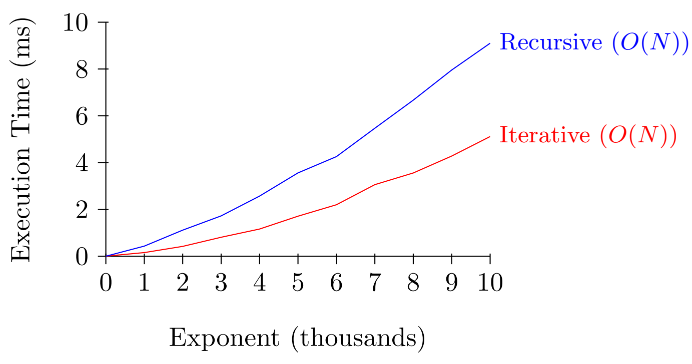
Iterative Solution

```python
#!/usr/bin/env python

import time;

def iterative(base, exp):
  if exp == 0:
    return 1;

  product = base;
  for i in range(1, exp):
    product *= base;
  return product;

if __name__ == "__main__":
  print("Exponent\tIterative");

  for x in range(0, 10001, 1000):
    start = time.time();
    iterative(3, x);
    end = time.time();

    print("%i\t%g" % (x, end - start));
```

Recursive Solution

```python
#!/usr/bin/env python

import time;

def recursive(base, exp):
  if exp == 0:
    return 1;

  if exp > 1:
    return base * recursive(base, exp - 1);
  else:
    return base;

if __name__ == "__main__":
  print("Exponent\tRecursive");

  for x in range(0, 10001, 1000):
    start = time.time();
    recursive(3, x);
    end = time.time();

    print("%i\t%g" % (x, end - start));
```

Results



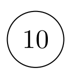
Exponent	Iterative (ms)	Recursive (ms)
0	0.003099	0.000954
1000	0.155926	0.432014
2000	0.421047	1.115080
3000	0.810862	1.724000
4000	1.162050	2.570150
5000	1.710890	3.561020
6000	2.199890	4.256960
7000	3.059150	5.470040
8000	3.557920	6.670000
9000	4.281040	7.956980
10000	5.115030	9.102110

Findings

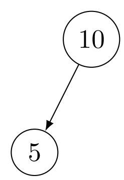
The iterative function has to iterate over the range of the exponent a single time (n), while the number of iterations of the recursive function is calculated by the function $a_1 = 1, a_n = 1 + a_{n-1}$. The iterative function has a growth function of $O(N)$, and the recursive function also has a growth of $O(N)$. The iterative function operates more efficiently than the recursive function because, in the recursive function, the application runtime has to push arguments and addresses to the call stack and invoke itself $n - 1$ times in order to calculate the value. In the iterative solution, this happens 1 time.

Problem 3 [COMPLETE]

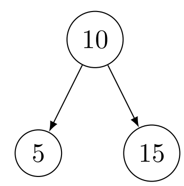
1. *insert(10)*



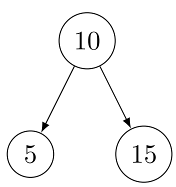
2. *insert(5)*



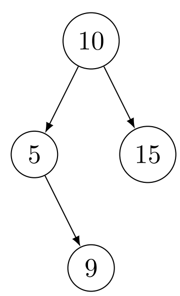
3. *insert(15)*



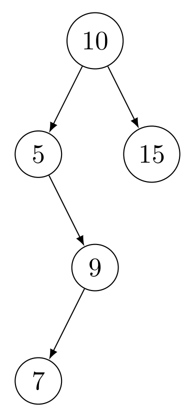
4. *insert(10)*



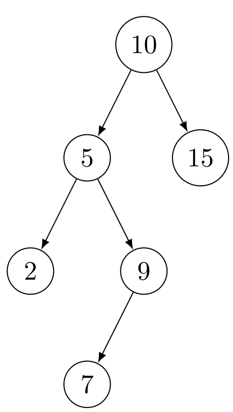
5. *insert(9)*



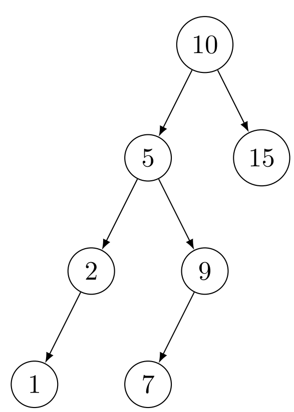
6. *insert(7)*



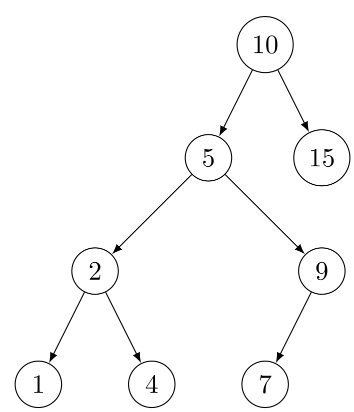
7. *insert(2)*



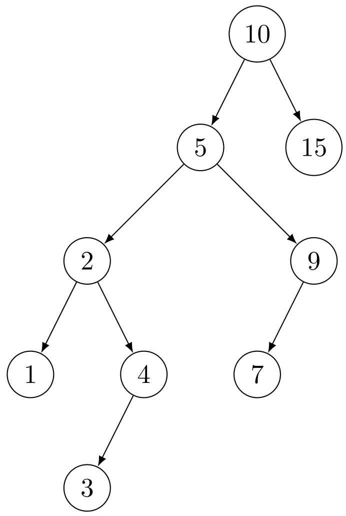
8. *insert(1)*



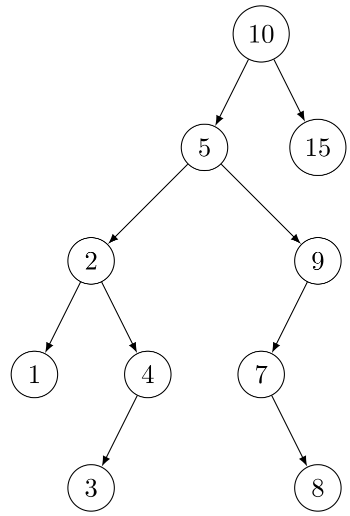
9. *insert(4)*



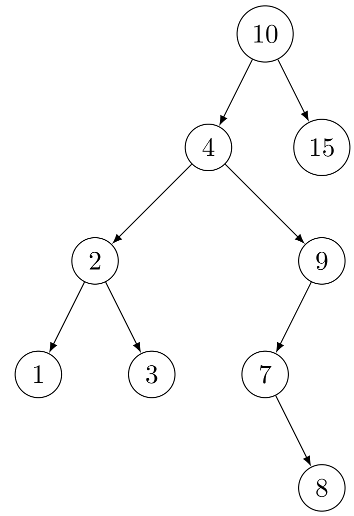
10. *insert(3)*



11. *insert(8)*

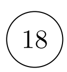


12. *delete(5)*

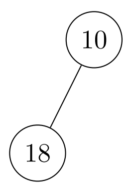


Problem 4 [COMPLETE]

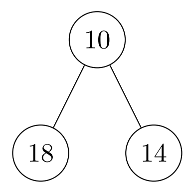
(a) 1. *insert(18)*



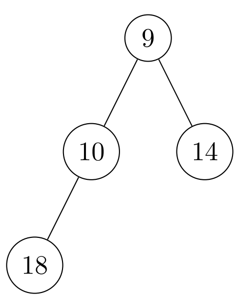
2. *insert(10)*



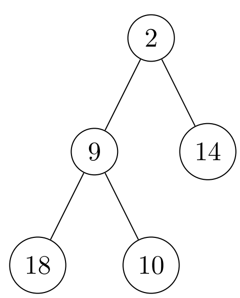
3. *insert(14)*



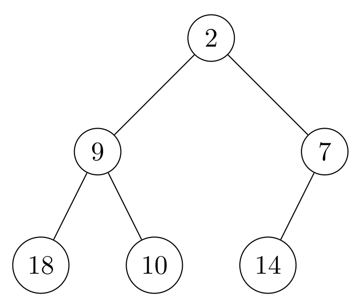
4. *insert(9)*



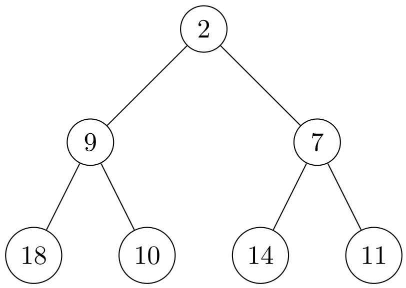
5. *insert(2)*



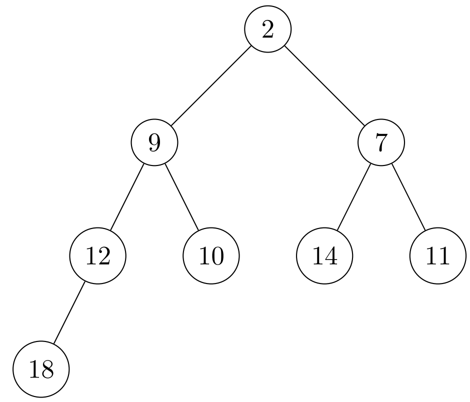
6. *insert(7)*



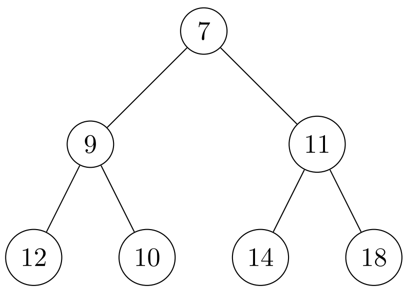
7. *insert(11)*



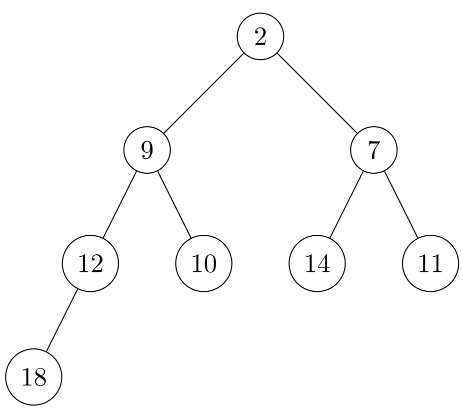
8. *insert(12)*



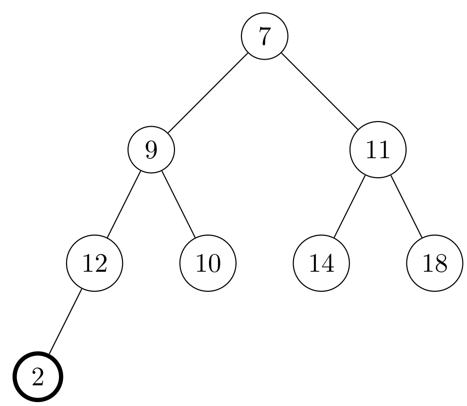
9. *delete-top()*



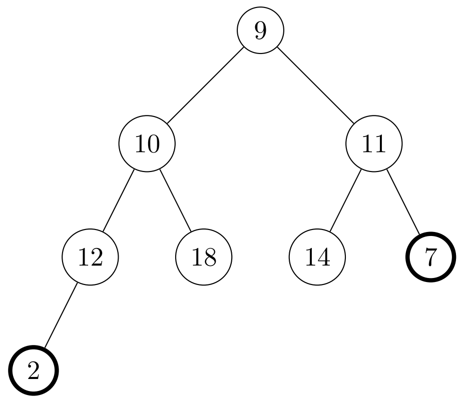
(b) 1. *min-heap*



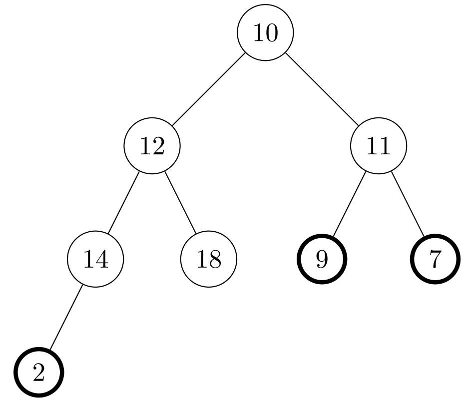
2. *Step 1*



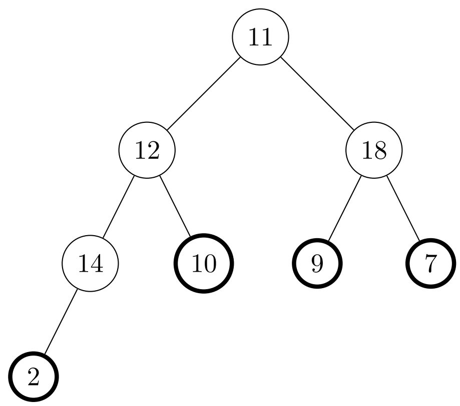
3. *Step 2*



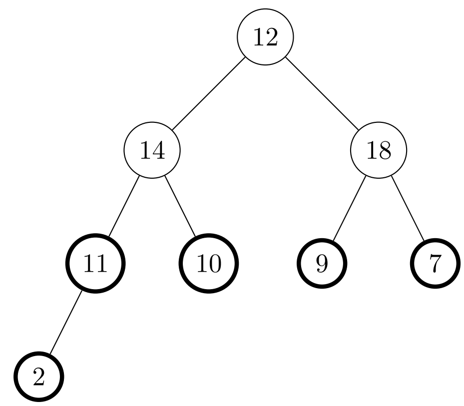
4. *Step 3*



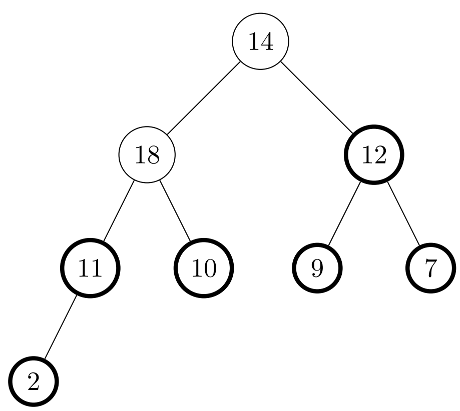
5. *Step 4*



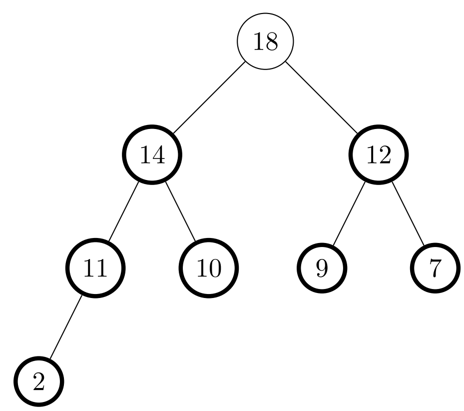
6. *Step 5*



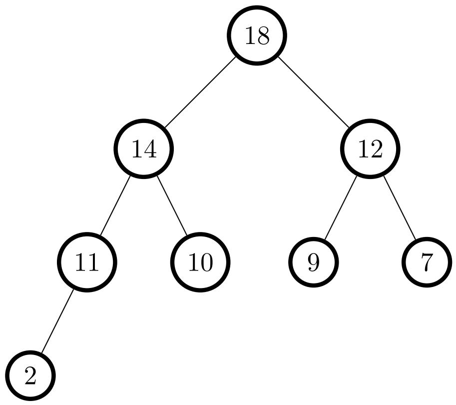
7. *Step 6*



8. *Step 7*



9. *Step 8 (sorted)*



Problem 5 [COMPLETE]

Algorithm 1 Computes the number of leaves in a binary tree

Inputs: T, a binary tree
Outputs: The number of leaf nodes in the initially-provided binary tree
Complexity: $O(N)$. Each node is computed against one time

```
function CountLeaves(T)
    if T = null then
        return 0
    else if T.left = null & T.right = null then
        return 1
    else
        return CountLeaves(T.left) + CountLeaves(T.right)
    end if
end function
```

Algorithm 2 Computes the height of a node in a binary tree

Inputs: T, a binary tree
Outputs: The height of the initally-provided binary tree
Complexity: $O(N)$. Each node is computed against one time

```
function NodeHeight(T)
    if T = null then
        return 0
    end if
    leftHeight := NodeHeight(T.left)
    rightHeight := NodeHeight(T.right)
    return 1 + Max(leftHeight, rightHeight)
end function
```

Algorithm 3 Computes the maximum depth of a binary tree

Inputs: T, a binary tree
Outputs: The maximum depth of the initally-provided binary tree from the root node to the farthest leaf
Complexity: $O(log(N))$. Each node is computed against its parent one time, reducing the number of computations (compared to N) by half for each iteration

function NodeDepth(T)
    if $T.parent = null$ **then**
        return 0
    else
        return 1 + NodeDepth($T.Parent$)
    end if
end function

Algorithm 4 Indicates whether a binary tree is full (or not)

Inputs: T, a binary tree
Outputs: The state of fullnes of the initially-provided binary tree
Complexity: $O(N)$. Each node is computed against one time

function IsFull(T)
    if $T.left \neq null$ & $T.right \neq null$ **then**
        return IsFull($T.left$) & IsFull($T.right$)
    else if $T.left = null$ & $T.right = null$ **then**
        return $true$
    else
        return $false$
    end if
end function

Problem 6 [COMPLETE]

The worst time complexity to compute the sum of a $n \times n$ 2-dimensional array is $O(N^2)$, because the application would have to iterate over each dimension in order to access every element of the array.

Problem 7 [COMPLETE]

i. The function computes the n-th number in the Fibonacci sequence.

ii. $F_n = F_{n-2} + F_{n-1}$, $n \geq 3$, $F_1 = F_2 = 1$

iii. 12 additions are performed to compute $unknown(6)$.

Problem 8 [COMPLETE]

Yes, this is a valid equality (Nicomachus's theorem), and has been proven by induction (`http://www.proofwiki.org/wiki/Sum_of_Sequence_of_Cubes`). Consider the following table:

i	$\sum_{i=1}^{k} i^3$	$\left(\sum_{i=1}^{k} i\right)^2$
1	1	1
2	9	9
3	25	25
4	100	100
5	225	225