Problem 1. [20 points]
Consider the following basic problem.

You’re given an array $A$ consisting of $n$ integers $A[1], A[2], \ldots, A[n]$. You’d like to output a two-dimensional $n$-by-$n$ array $B$ in which $B[i, j]$ (for $i < j$) contains the sum of array entries $A[i]$ through $A[j]$ — that is, the sum


. (The value of array entry $B[i, j]$ is left unspecified whenever $i \geq j$, so it doesn’t matter what is output for these values.)

Here’s a simple algorithm to solve this problem.

```plaintext
For i = 1, 2, \ldots, n
    For j = i+1, i+2, \ldots, n
        Add up array entries $A[i]$ through $A[j]$
        Store the result in $B[i,j]$
    Endfor
Endfor
```

a) For some function $f$ that you should choose, give a bound of the form $O(f(n))$ on the running time of this algorithm on an input of size $n$ (i.e., a bound on the number of operations performed by the algorithm).

b) Although the algorithm above is the most natural way to solve the problem — after all, it just iterates through the relevant entries of the array $B$, filling in a value for each — it contains some highly unnecessary sources of inefficiency. Give a different algorithm to solve this problem, with an asymptotically better running time.

Problem 2. [20 points]
Consider the searching problem:

Input: A list of $n$ numbers $A = < a_1, a_2, \ldots, a_n >$ and a value $v$.

Output: An index $i$ such that $v = A[i]$, or the special value $NIL$ if $v$ does not appear in $A$.

1. Write pseudocode for the linear search, which scans through the sequence, looking for $v$. Hint: You might need a loop!

2. What is the Big-Oh performance of the algorithm you defined?

3. In your own words, show that your algorithm is correct. Remember, your explanation should be correct, clear, and concise.
Problem 3. [15 points]
Remember the Binary Tree data structure? You have studied this in CS230 before, and you might have also worked with it in other courses.

To remind you, a Binary tree is a tree (acyclic graph with a root node), such that each parent node has at most two children nodes. The layout of the tree depends completely on how it was built and the elements in it. The only rule is on the maximum number of children that each node can have.

Given this definition of a Binary Tree, we define the search algorithm as shown below. The input to this algorithm is the root node of the tree, and the element to be found. The output of this algorithm is a boolean indicating if the element has been found in the tree or not.

Search(root, element) returns boolean
  if(root.value == element)
    return true
  boolean found = false
  if(root.left != null)
    found = Search(root.left, element)
  end if
  if(found == false && root.right != null)
    found = Search(root.right, element)
  end if
  return found
end of function

a) Compute the worst case running time complexity of the algorithm ($T(n)$). Show your work.

b) What is the big-Oh of this algorithm?

Problem 4. [15 points]
Now, another similar data structure to a Binary Tree is the Binary Search tree.

To remind you, a Binary Search tree is a tree (acyclic graph with a root node), such that each parent node has at most two children nodes. Moreover, the values of the children on the left have to be less than that of the parent (or equal), and the values of the children on the right have to be more than that of the parent. The layout of the tree still depends completely on how it was built and the elements in it.

Given this definition of a Binary Search tree, we define the search algorithm as shown below. The input to this algorithm is the root node of the tree, and the element to be found. The output of this algorithm is a boolean indicating if the element has been found in the tree or not.

Search(root, element) returns boolean
  if(root.value == element)
    return true
  if(element < root.value)
    if(root.left != null)
      return Search(root.left, element)
    end if
  else if(root.right != null)
    return Search(root.right, element)
  end if
end of function

a) Compute the worst case running time complexity of the algorithm ($T(n)$). Show your work.
b) What is the big-Oh of this algorithm?

**Problem 5.  [10 points]**  
Analyze the worst case running time complexity of the $HeapifyUp(H, i)$ algorithm as shown in your textbook, page 61.  
Explain your answer and clarify your computation.

**Problem 6.  [20 points]**  
Analyze the best and worst case running time complexities of the $HeapifyDown(H, i)$ algorithm as shown in your textbook, page 63.  
Explain your answer and clarify your computation.