Problem 1 Proof of Asymptotic Growth

(a) Show that the function f(n) = $3n^2 - 2n$ is $\Theta(n^2)$.
Let f(n) = $3n^2 - 2n$ and let g(n) = n^2.
Let $c_1, c_2, n_0 \in \mathbb{Z}^+$ such that using the inequality growth of theta

$$\Theta(n^2) = \{0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

or

$$\Theta(n^2) = \{0 \leq c_1 n^2 \leq 3n^2 - 2n \leq c_2 n^2\}$$

Let $c_1 = 2 \& c_2 = 4 \& n_0 = 1$ such that the inequality can be rewritten

$$\Theta(n^2) = \{0 \leq 2n^2 \leq 3n^2 - 2n \leq 4n^2\}$$

$\forall n \geq n_0 = 1$ this holds to be true.
Therefore, f(n) = $3n^2 - 2n$ is $\Theta(n^2)$ has been proved as desired.

(b) Let f(n) = $3n^2 + n$ and g(n) = n^2 and suppose that we assume that the given proof is true such that it is little o growth.
To begin with the definition of little o growth is

$$\mathrm{o}(g(n)) = \{(0 \leq f(n) < c(g(n))\}$$

or

$$\mathrm{o}(n^2) = \{0 \leq 3n^2 + n < cn^2\}$$

such that $\forall c \in \mathbb{R}^+, \exists n_0 \in \mathbb{R}^+$ s.t. it holds $\forall n \geq n_0$.
Unlike Big O notation, little o notation must hold $\forall c \in \mathbb{R}^+$. Let c = 3, then it leads to a contradiction as

$$o(n^2) \neq \{0 \leq 3n^2 + n \not< 3n^2\}$$

Furthermore, since little o is also known for being asymptotically smaller then using the limit to infinity relation then

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

. But substituting in f(n) and g(n) with the given functions:

$$\lim_{n \to \infty} \frac{3n^2 + n}{n^2} = \frac{3}{1} = 3 \neq 0$$

Here this leads to a contradiction as $3 \neq 0$ which implies that it is not a little o growth.
Therefore, the little o proof given is not a correct proof.

Problem 2 Tree Traversals

(a) Non-recursive method for pre-order From Line 1 to 8 it takes a constant time to operate

Algorithm 1 printBinaryTreeNodes(TreeNode root

```
if x $\neq$ nil
stack = new Stack()
stack.push(root)
while not stack.isEmpty()
    node = stack.pop()
    print(node.key)
    if node.right != NIL
        stack.push(node.right)
    if node.left != NIL
        stack.push(node.left)
else
    return stack.isEmpty()
```

therefore, it runs at O(1) time.

(b) Prove that your solution works and is O(n).
From the CRLS textbook preorder prints from the roots before the subtree. Additionally, by inserting the right child first, the left child leaves following the First in Last Out property of Stacks.
Each call from line 1 to 12 take at O(1) and it takes n items therefore,

$$O(1 \times n) = O(n)$$

Problem 3 Catenable Stack

(a) No because the way to implement catenate stacks has to go through every element in the array there taking O(n) times therefore making it impossible to do it within O(1) times.

(b) Algorithms that implement the original Stack ADT.

Algorithm 2 Stack()

```
1: ListNode top = null
2: size = 0
3: ListNode bottom = null
```

Algorithm 3 push(Object o)

```
1: ListNode newNode = new ListNode(o)
2: newNode.next = top
3: top = newNode
4: size++
5: if size == 1
6:         tail = top
```

Algorithm 4 size()

```
1: return size
```

Algorithm 5 isEmpty()

```
1: return size == 0
```

Algorithm 6 top()

```
1: if top != NIL
2:     return top
```

Algorithm 7 POP()

```
1: if isEmpty()
2:     return NIL
3: else
4:     if size == 1
5:         tail == null
6:     popKey = top.key
7:     top = top.next
8:     size–
9:     return popKey
```

(c) Design an algorithm that implements catenate(Stack s) operation in O(1) time. Write down the algorithm and prove that it runs in O(1) time.

Algorithm 8 CATENTATE(Stack s)

```
if isEmpty()
    head = s.head
    tail = s.tail
else
    tail.next = s.head
    if !s.isEmpty()
        tail = s.tail
size+= s.size
```

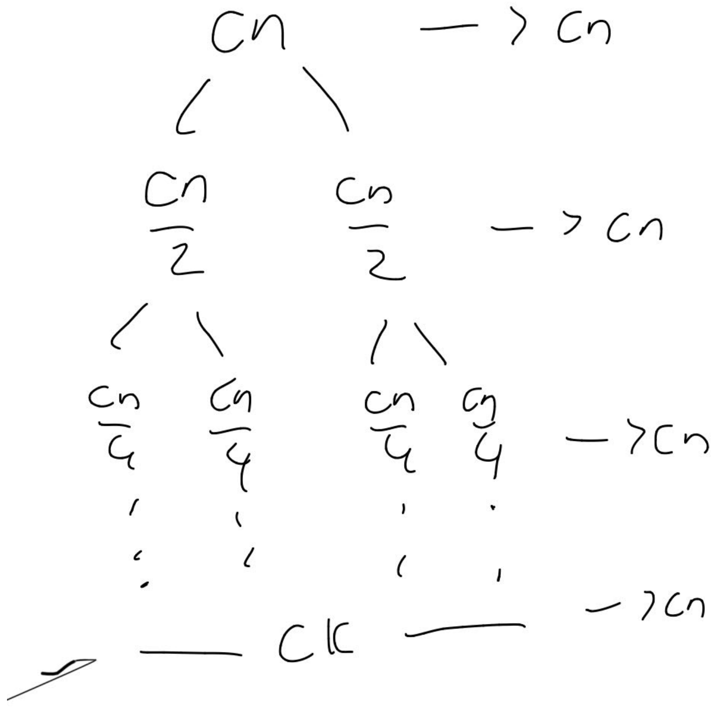
Problem 4: Hybrid Merge/ Insertion Sort Algorithm

(a) Show that InsertionSort can sort the n/k sublists, each of length k, in $\Theta(nk)$ worst-case time.
Recall that the worst case runtime for length of k is $\Theta(k^2)$. Given that we are sorting $\frac{n}{k}$ sublists then

$$\Theta(k^2 \times \frac{n}{k}) = \Theta(nk)$$

Therefore, Insertion-Sort has a worst case runtime of $\Theta(nk)$ as desired.

(b) The diagram of the merge tree is the following:



At each level a total of cn elements are merged at each level. Since there are $\frac{n}{k}$ sublists then it will take $\Theta(\lg(\frac{n}{k}))$ to merge. Then as a result,

$$\Theta(n \times \lg(\frac{n}{k})) = \Theta(n\lg(\frac{n}{k}))$$

Hence, the run-time is $\Theta(n\lg(\frac{n}{k}))$ as desired.

(c) The bigger we make k the bigger lists InsertionSort has to sort. At some point, its $\Theta(n^2)$ growth will overcome the advantage it has over Merge-Sort in lower constant overhead. How big can k get before InsertionSort starts slowing things down? Derive a theoretical answer by proving the largest value of k for which the hybrid sort has the same Θ runtime as a standard $\Theta(nlgn)$ MergeSort. This will be an upper bound on k.

1. Suppose that k = f(n) = lg(n). Then the expression will have the form of $\Theta(nlgn)$ which will run slower than Merge sort. Therefore, the upper bound for k is lg(n).

2. Let k = lg(n) then in the equation

$$\Theta(nk + nlg(n/k))$$

or

$$\Theta(nk + n[lg(n) - lg(k)]$$

when k is substituted in then

$$\Theta(nlg(n) + n(lg(n) - lg(lg(n))$$

or

$$\Theta(2nlg(n) - nlg(lg(n))$$

Since we can ignore the constant 2 and from a mathematical standpoint $n\lg(n) > n\lg(\lg(n))$ then the run-time is $\Theta(n\lg(n))$

(d) Now suppose we have two specific implementations of InsertionSort and MergeSort. How should we choose the optimal value of k to use for these given implementations in practice?
The value of k should depend on the largest value of k where Insertion-Sort is faster than the Merge-Sort.