Linked Lists

1. You are given the linked sequence: $1 \rightarrow 2 \rightarrow 3$. You may assume that each node has one field called `value` and one called `rest`.

   (a) 1 points to 2 and 2 points to 3. What does 3 point to?
   The implementer may cause the 3 node to point to either `None` or a sentinel node.

   (b) Draw out the linked list structure and add a 5 to the end.

```
+----------+    +----------+    +----------+    +------------+
| value: 1 |    | value: 2 |    | value: 3 |    | value: 5   |
| rest: ------->| rest: ------->| rest: ------->| rest: None |
+----------+    +----------+    +----------+    +------------+
```

   OR

```
+----------+    +----------+    +----------+    +----------+    +-----------------+
| value: 1 |    | value: 2 |    | value: 3 |    | value: 5 |    | value: SENTINEL |
| rest: ------->| rest: ------->| rest: ------->| rest: ------->|                 |
+----------+    +----------+    +----------+    +----------+    +-----------------+
```

   (c) Write pseudocode to add an element onto the end of a linked sequence.

```
def append(linked_list, value):
    newEnd = new node
    newEnd.value = value
    newEnd.rest = None # (or SENTINEL)
    cur = linked_list
    while cur is not end of linked_list:
        cur = cur.rest
    cur.rest = newend
```

   (d) Let's say we want to add a 4 to the list from 1b. What is a procedure for inserting this value at a certain position in the linked sequence? (There are many possibilities.)
   Use some variation of this strategy:
   Find the preceding element's node, link its rest node to the new node containing 4, then link that new node to what was the following node.

Binary Search using Python Lists

2. Given the sorted list [1, 4, 9, 16, 25, 69, 420, 1337], write out the steps that a binary search would take to find the number 69.

```
[1,4,9,16,|25|,69,420,1337]
[1,4,9,16,25,69,|420|,1337]
[1,4,9,16,25,|69|,420,1337]
```

Sorting

3. Below is Python code for a function that performs an insertion sort and prints `data` after each iteration of the `for` loop.

```python
def insertion_sort( data ):
    for marker in range( 1, len( data ) ):
        val = data[marker]
        i = marker
        while i > 0 and data[i-1] > val:
            data[i] = data[i-1]
            i -= 1
        data[i] = val
        print( data )
```

(a) Write out what the function will print for the input list: [3,2,7,1].

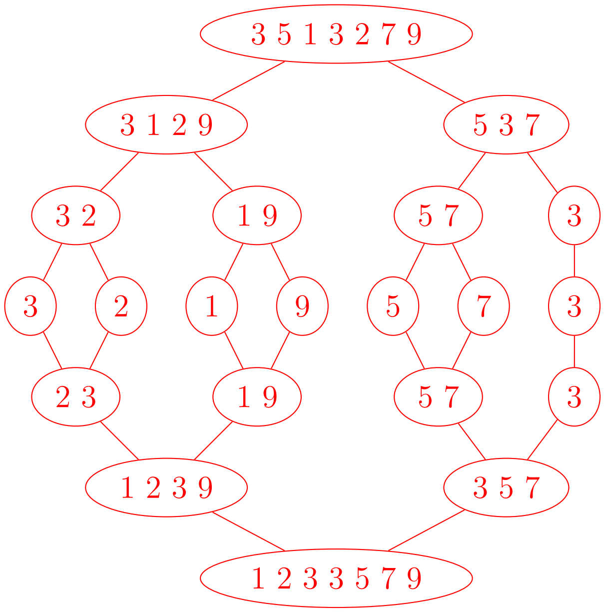
```
[2,3,7,1]
[2,3,7,1]
[1,2,3,7]
```

(b) What is the sort algorithm's time complexity?

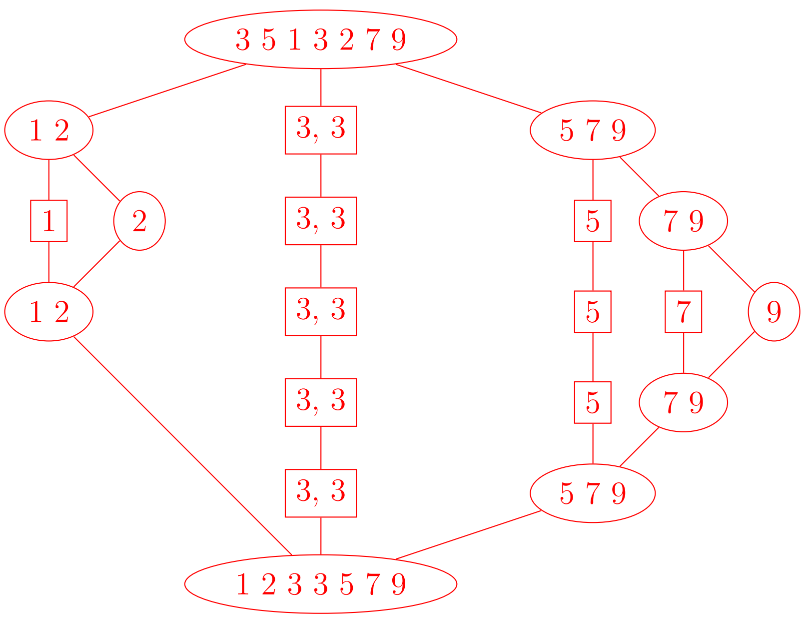
$O(N^2)$

4. Show the stages of a merge sort and a quicksort on the following list: [3,5,1,3,2,7,9]. Be sure to identify your pivot.

Merge sort:



Quicksort (using the first element in the list as a pivot):



Stacks and Queues

5. It is possible to implement both stacks and queues using only simple Python lists.

   (a) Write the following functions that implement stack functionality atop a Python list. Your stack must provide the following functionality:

   `push(lst, elm)` — Push `elm` onto the top of the stack

   `pop(lst)` — Return the top element of the stack and remove it from the stack

   `isEmpty(lst)` — Return whether the stack is empty

   `peek(lst)` — Return the top element of the stack without modifying the stack

```
def push(lst, val):
    lst.append(val)
def pop(lst):
    return lst.pop()
def peek(lst):
    return lst[-1]
def isEmpty(lst):
    return (len(lst) == 0)
```

   (b) Write the following methods to create a queue in similar fashion to previous question (with a Python list as the data structure managing elements "under the hood"):

   `enqueue(lst, val)` - put a value into the queue

   `dequeue(lst)` - take a value out of the queue

```
def enqueue(lst, val):
    lst.append(val)
def dequeue(lst):
    lst.pop(0)
```

   (c) Which of the data structures you implemented is more efficient and why? Give a better way to implement the slower structure, and discuss how this would change the time complexity of its operations.

   Because the queue must be able to modify both ends of the list, it pays an O(n) cost to remove the beginning element during each dequeue operation. This could be reduced to O(1) by using a linked list instead of a Python list.

Structures

6. For each of the following structures with the given attributes, write the code to define the structures:

   - A Hotel
     - `name` (a string)
     - `rooms` (a list of Room structures)
     - `location` (a string)
   - A Room
     - `number` (an integer)
     - `capacity` (an integer)
     - `price` per night (a float)

```
from rit_lib import *

Hotel = struct_type("Hotel",
    (str, "name"), (list, "rooms"), (str, "location"))

Room = struct_type("Room",
    (int, "number"), (int, "capacity"), (float, "price"))
```

Greedy Algorithms

7. Given that an algorithm is *greedy*, is it guaranteed to return an *optimal* solution?

   NO. Greedy algorithms always choose the *current* best solution, which is not necessarily the *overall* best solution!

8. In the game Black and White[1], the player is faced with a row of identical double-sided chips. You can probably guess what colors the two sides of each chip are. The objective is to flip as many chips as necessary so their exposed colors match that of a target pattern.
   The catch? Reordering the chips is said to be Impossible by those who seem to know what they're talking about.
   The *real* catch? Flipping a group of consecutive tiles can be accomplished in a single "action." If every flip takes one "action," write a function `bwMoves` that, given a starting pattern and target pattern as equal-length strings, returns the minimum number of actions required to get them to match. For instance, `bwMoves( 'BBWBBWBBBB', 'WWWWWBBWWB' )` should return 3.

```python
def bwMoves(start, target):
    actions=0
    first=0
    for index in range(len(start)):
        if start[index]==target[index]:
            if first!=index: # Each flip works up to (but not including)
                actions+=1   # the index pointer. If first==index, that's
            first=index+1    # 0 elements, so there is nothing to flip.
                             # (i.e. There were two no-flips in a row.)
    if start[-1]!=target[-1]:
        actions+=1
    return actions
```

[1]Special thanks to Professor Butler for unwittingly allowing us to rip off his problem.