Cindy Lee
Net ID: cl3616
N12335562

Question 1:

```
def minandmax (lst, minimum = None, maximum = None, copy = False):
    if len (lst) == 0:
        return minimum, maximum
    if copy == False:
        copy = True
        lst = lst [:]
        minimum = lst [0]
        maximum = lst [0]
    last = lst.pop()
    if last > maximum:
        maximum = last
    elif last < minimum:
        minimum = last
    return minandmax (lst, minimum, maximum, copy)
```

Runtime of this function is $\boldsymbol{\theta(n)}$.

For one function call:

```
if len (lst) == 0:
        return minimum, maximum
```

This runs in constant time as the function checks whether the length is equal to a number. This is then $\theta(1)$

```
    if copy == False:
        copy = True
        lst = lst [:]
        minimum = lst [0]
        maximum = lst [0]
```

For the first function call, this if statement runs in linear time because it copies the original list. Copying a list has a runtime proportional to the size of the list being copied and assigning variables to a certain number is constant time. The runtime of this if statement for the first call would then be $\theta(n)$ because we only care about the biggest term of the runtime, which in this case is n. However, for the later recursive calls, this if statement, does not perform.

```
    last = lst.pop()
```

Popping runs in constant time $\theta(1)$.

```
if last > maximum:
        maximum = last
```
This runs in constant time as it is comparing once, which is constant time, and assigning is constant time: $\theta(1)$.

```
elif last < minimum:
        minimum = last
```
Similar to the code prior to this one, runs in $\theta(1)$.

For the first function call: $\theta(n)$. There are constant times added to $\theta(n)$, but we only care about the largest term, which is n.

For the following function calls, each term in the list compares with the maximum and minimum while skipping over the if statement for copy. This then leads to n-1(1) comparisons for the rest of the recursion calls, which is $\theta(n-1)$

The total run time is $\theta(n) + \theta(n-1) = \theta(2n-1)$, but because we do not care about coefficients and smaller times, the run time is $\theta(n)$.

Question 2

```
def intlog (number, start = 0):
    if number <= 0:
        return None
    if number // 2 == 0:
        return start
    return intlog (number//2, start + 1)
```

Runtime of this function is $\boldsymbol{\theta(log\ (n))}$.

For one function call:

```
    if number <= 0:
        return None
```
This runs in constant time during comparison of a number

```
if number // 2 == 0:
        return start
```
This runs in constant time when the program compares whether the number is divisible by 2

```
return intlog (number//2, start + 1)
```
Inside the return call, each run of dividing and adding runs in constant time.

One function call runs in constant time.

The function runs $\log_2$ (number) times, and because each function call is $\theta(1)$, the runtime for the recursive function is $\theta(\log(n))$.

Question 3

```
def unique (lst, start = 0, end = None):
    if end == None:
        end = start + 1
    if start == len (lst):
        return True
    elif end == len (lst):
        start = start + 1
        end = start + 1
        return unique (lst, start, end)
    if lst [start] == lst [end]:
        return False
    else:
        return unique (lst, start, end + 1)
```

Runtime of this function is $\boldsymbol{\theta(n^2)}$.

For one function call:

```
if end == None:
        end = start + 1
```

Comparing end to None and changing what end is runs in constant time.

```
if start == len (lst):
        return True
```

Comparing start to the length of the list runs in constant time, as len (lst) runs in constant.

```
elif end == len (lst):
        start = start + 1
        end = start + 1
        return unique (lst, start, end)
```

The elif, besides the return call, runs in constant time, and changing the values of start and end run in constant time.

```
    if lst [start] == lst [end]:
        return False
```

This if statement runs in constant time when comparing values of the list.

```
    else:
        return unique (lst, start, end + 1)
```

The end + 1 runs in constant time.

For one call, the function runs in constant time. Only one recursion is called per function call depending on the conditions. Each element checks with n-1, n-2 ….all the way down to 1 element, which is the summation. The runtime is the $\theta(n^2)$.

Question 4

```
def subset (lst, start = 0):
   if start == len (lst):
      return [[]]
   sublist = subset (lst, start + 1)
   return sublist + [[lst[start]] + subby for subby in sublist]
```

```
if start == len (lst):
      return [[]]
```

This if statement runs in constant time as it compares start to the length of the list, in which len () runs in constant time. When the if statement runs, the return statement is also constant time because it creates a list of one element, which itself is empty.

```
   return sublist + [[lst[start]] + subby for subby in sublist]
```

This return statements adds two lists. The list comprehension runs in linear time $\theta(n)$ because we add one element to each of the subsets inside the sublist with n elements. The new list created for each subset in sublist would be of n+1 terms, which runs in $\theta(n)$ for each one. The sublist itself has n elements, so creating a new list for returning is $\theta(n^2)$. To return the addition of the original subset with the sublist would be $(n+1)n + n = n^2 + 2n$, which is also $\theta(n^2)$.

One function call is $\theta(n^2)$ and this function calls itself for n elements in the list, which results in a run time of $\theta(n^3)$.

Question 5

```
def palindrome (string, start = 0):
   if string [start] != string [len(string) - start - 1]:
      return False
   if start > len (string) - start - 1:
      return True
   return palindrome (string, start + 1)
```

Runtime of this function is $\theta(n)$.

For one function call:

```
   if string [start] != string [len(string) - start - 1]:
      return False
```

This if statement runs in constant time because retrieving values of a list is constant time and comparing two values run in constant times.

```
if start > len (string) - start - 1:
    return True
```

This if statement runs in constant time because comparing the value of start to another value runs in constant time. Finding length of a string is constant time.

```
  return palindrome (string, start + 1)
```

The start + 1 of the recursive calls runs in constant time.
The recursive call runs n/2 times because each call compares the first and last element. Once the recursive call compares the middle two or middle elements, the recursion is completed.
Therefore, the runtime of this function is $\theta\left(\frac{n}{2}\right) = \theta(n)$.

Question 6

```
def rearrange (S, k, start = 0, end = None):
    if end == None:
        end = len (S) - 1
    if start == end:
        return S
    if S [start] > k:
        store = S [end]
        S [end] = S [start]
        S [start] = store
        return rearrange (S, k, start, end - 1)
    else:
        return rearrange (S, k, start + 1, end)
```

Runtime is $\boldsymbol{\theta(n)}$.

For one function call:

```
if end == None:
    end = len (S) - 1
```

This compares end to None in constant time, and sets end to one less than the length of the list, in which len (S) runs in constant time.

```
if start == end:
    return S
```

This if statement runs in constant time after comparing start to end, and returning S runs in constant time as S is the original list and not a copy.

```
if S [start] > k:
    store = S [end]
    S [end] = S [start]
    S [start] = store
```

```
        return rearrange (S, k, start, end - 1)
```
The if statement runs in constant time when it compares the value of the list at start to k. If the if statement is true, the following procedures under the if statement each run constant time as it retrieves values of S at certain indexes and stores a value back into an index. In the return function, end – 1 runs in constant time.

```
    else:
            return rearrange (S, k, start + 1, end)
```
Inside the return function, start + 1 runs in constant time.

The function compares the first element to k and if it is larger, it swaps with the last element and calls the function over from the first element to the second to last element. If it is not bigger, then the function calls the function again from the second element to the last.

One function call runs in constant time without regarding the recursive calls. There is a total of n recursive calls, depending on the conditions, so the runtime is $\theta(n)$.