**Q6(a)**

$\mathcal{O}(n)$, because the loop repeats $n$ times and each iteration takes at most $k$ ms for some $k$, regardless of what $n$ is.

**Q6(b)**

Also $\mathcal{O}(n)$. The reasoning is the same.

**Q6(c)** ¶

In the worst case, we keep calling `f` until $j-i$ becomes 0. Initially, $j-i$ is $n(1/4-1/5)=0.05n$, and we keep making $j-i$ smaller by a factor of 2. In total, to get to 0 will take about $\log_2 0.05n$ calls, i.e., we make $\mathcal{O}(\log n)$ calls in total. Each call takes time that doesn't depend on $j-i$, so the runtime is $\mathcal{O}(\log n)$

**Q7**

Here is the outline of the Forward Step of Gaussian Elimination:

```
for each i in 0, 1, 2, ...., m-1
       Find the row at i or below with the smallest leading index (Look through n*m
entries in the worst case,
       which happens if the only non-zero coefficient is in the bottom-right corne
r)
       Swap the rows (Depending on the implementation, could be constant time)
       Eliminate all coeffcients below the leading index in the swapped row (at mos
t O(m))
```

This means the Forward Step is $\mathcal{O}(m^2 \times n)$.

This was sufficient for full marks, but there is a subtlety there. The subtlety is as follows.

This: Find the row at i or below with the smallest leading index (Look through n*m entries in the worst case, which happens if the only non-zero coefficient is in the bottom-right corner)

is a rough approximation: we can actually find the row at i or below with the smallest leading index in at most $n \times (m-i)$ steps. As it turns out, if you do the math, you'll find that $\mathcal{O}(m^2 \times n)$ is a tight bound.

### Q8

The function is an inefficient implementation of MergeSort (implemented iteratively) in disguise.

#### Q8(a)

`mystery_helper()` returns the sorted version of `L1+L2`, in non-increasing order.

#### Q8(b)

`mystery(L)` returns `L`, sorted in non-increasing order. This is the iterative implementation of MergeSort: first, we sort lists of length 2 contained in `L`. Then, we sort lists of length 4 contained in `L`, and so on, until `L` is all sorted.

#### Q8(c)

The complexity of `mystery_helper()` is $\mathcal{O}(n^2)$, so let's say it takes $kn^2$ time

Let's count the runtime by iteration of the `while`-loop:

```
1-st iteration: k * (n/2) * 2^2 = k*2n
2-nd iteration: k * (n/4) * 4^2 = k*4n
3-rd iteration: k * (n/8) * 8^2 = k*8n
...
log(n)-th iter: k *  (2)  * n^2 = k*(2n)*n
```

Summing those up, we get $k \times n \times (2 + 4 + 8 + \ldots + 2n)$

Now $2 + 4 + 8 + \ldots + 2n = 2(1 + 2 + 4 + \ldots + 2^{\log_2 n}) = 2(2n - 1)$.

That means in total the complexity if $\mathcal{O}(n^2)$.

### Q9

In [1]:
```
import santa
import copy

def board_full(board):
    for i in range(len(board)):
        for j in range(len(board[i])):
            if board[i][j] == " ":
                return False
    return True


def invert_board(board):
    new_board = copy.deepcopy(board)
    for i in range(len(board)):
        for j in range(len(board[i])):
            if board[i][j] == "X":
                new_board[i][j] = "O"
            elif board[i][j] == "O":
                new_board[i][j] = "X"
    return new_board

def o_will_lose(board):
    if santa.x_won(board):
        return True

    if santa.x_won(invert_board(board)):
        return False


    if board_full(board):
        return False

    for i in range(len(board)):
        for j in range(len(board[i])):
            if board[i][j] == " ":
                board[i][j] = "O"
                if not x_will_win(board):
                    board[i][j] = " "
                    return False
                board[i][j] = " "
    return True


def x_will_win(board):
    if santa.x_won(board):
        return True

    if santa.x_won(invert_board(board)):
        return False

    if board_full(board):
        return False

    for i in range(len(board)):
        for j in range(len(board[i])):
            if board[i][j] == " ":
```

```
                board[i][j] = "X"
                if o_will_lose(board):
                   board[i][j] = " "
                   return True
                board[i][j] = " "
    return False
```

In [2]:
```
print(x_will_win([[" ", "X", " "],
                  [" ", "X", " "],
                  [" ", "O","O"]]))
```

```
False
```

In [3]:
```
print(x_will_win([[" ", " ", " "],
                  [" ", "X", "O"],
                  [" ", " ", " "]]))
```

```
True
```

In [4]:
```
print(x_will_win([[" ", " ", " "],
                  [" ", " ", " "],
                  [" ", " ", " "]]))
```

```
False
```

The contents of `santa.py`:

In [5]:

```
#1 mark for look-ahead by 1 move
#2 for "don't know" or blank


import copy



def is_row_all_three(board, mark, row_num):
    return board[row_num] == [mark] * 3

def is_col_all_three(board, mark, col_num):
    for i in range(3):
        if board[i][col_num] != mark:
            return False
    return True

def is_left_diag_all_three(board, mark):
    for i in range(3):
        if board[i][i] != mark:
            return False
    return True

def is_right_diag_all_three(board, mark):
    for i in range(3):
        if board[i][2-i] != mark:
            return False
    return True

def is_win(board, mark):
    for i in range(3):
        if is_row_all_three(board, mark, i):
            return True

    for i in range(3):
        if is_col_all_three(board, mark, i):
            return True

    if is_right_diag_all_three(board, mark):
        return True

    if is_left_diag_all_three(board, mark):
        return True

    return False


def make_empty_board():
    board = []
    for i in range(3):
        board.append([" "]*3)
    return board
```

```python
def x_won(board):
    return is_win(board, "X")
```