Runtime Lab

Q1

```
def search_q1(Y, n, x):

    for i in range (0, n):
        if Y[i] == x:
            return i
    return -1


Y = ['apple', 'banana', 'mango', 'grapes', 'pineapple', 'durian']
x = "pineapple"

n = len(Y)
result = search_q1(Y, n, x)
if result == -1:
    print("Element is not present in the list")
else:
    print("Element", x, "is present at index", result)
```

Given that we're ignoring constants, the time complexity will be the len(Y) (which is 6), which in this case in Y.

Q2

```
def search_q2(X, item):
    first = 0
    last = len(X)-1
    found = False
    while first<=last and not found:
        mid = (first + last)//2
        if X[mid] == item:
            found = True
            print("The element item", item, "was found at index ", X.index(60))
        else:
            if item < X[mid]:
                last = mid - 1
            else:
                first = mid + 1
    return found
```

```
print(search_q2([10, 15, 35, 42, 60, 70, 82, 94], 60))
```

In the above script, the sample size is constantly halved. This is binary the time complexity will be O(log(8)), or in the general case O(log(n)).

Q3

```
test = 0
n = 10
for i in range(n):
   test = test + 1

for j in range(n):
   test = test - 1
```

We have 2 loops which are going over the same sample size, and since we're ignoring constants, this time complexity will be O(n) + O(n), or in this case O(10) + O(10).

Q4

```
i = n
while i > 0:
   k = 2 + 2
   i = i // 2
```

Time complexity will be O(log(n)), given that the sample space us constantly halved.

Q5

```
mat = [[1, 2, 3], [1, 1, 1], [5, 7, 8]]
add = 0
for i in range(len(mat)):
    for j in range(len(mat[0])):
        add += mat[i][j]
print(add)
```

```
```

The time complexity for this script is O(n^2), or in this case O(9).

The reason is that we have a main loop and a nested loop. The main loop has 3 elements and the nested loop also has 3 elements (in this script at least). Thus O(n^2).

Q6

```
def fibonacci(n):
    if n<2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)


for n in range(2,12,2):
    print("Series sum for {} is {}".format(n, fibonacci(n)))
```

Due to recursion, the time complexity of the above script will be O(2^n) (we're discounting the for loop since its linear or O(n)).