Week 7: Going over some lab exercises on recursion and greedy algorithms

Question 1

Write a Python function

```
def timesOccursIn(k,A)
```

which which takes an integer and an array of integers and returns the number of times the integer occurs in the array. You must use recursion and no loops for this question.

For example, if its arguments are 5 and [1,2,5,3,6,5,3,5,5,4] the function should return 4.

Hint: Suppose `A` is not empty. If the first element of `A` is in fact `k`, the number of times that `k` occurs in `A` is "1 + the number of times it occurs in `A[1:]`". Otherwise, it is the same as the number of times it occurs in `A[1:]`. On the other hand, if `A` is the empty array `[]` then `k` occurs 0 times in it.

Question 2

Write a Python function

```
def isSubArray(A,B)
```

which takes two arrays and returns `True` if the first array is a (contiguous) subarray of the second array, otherwise it returns `False`. You may solve this problem using recursion or iteration or a mixture of recursion and iteration.

For an array to be a subarray of another, it must occur entirely within the other one without other elements in between. For example:

- [31,7,25] is a subarray of [10,20,26,31,7,25,40,9]
- [26,31,25,40] is not a subarray of [10,20,26,31,7,25,40,9]

Hint: A good way of solving this problem is to make use of an auxiliary function that takes two arrays and returns True if the contents of the first array occur at the front of the second array, otherwise it returns False. Then, `A` is a subarray of `B` if it occurs at the front of `B`, or at the front of `B[1:]`, or at the front of `B[2:]`, etc.

Question 3

Let `coin = [200, 100, 50, 20, 10, 5, 2, 1]`. Write a modified version:

```
def coinSplitGD2(m)
```

of the coin split method `coinSplitGD(m)` which works as follows. It returns an array of integers of the same length as the array `coin` that it uses, where each position gives the number of coins needed of the coin at that position.

So if `m` is 143, it will return [0, 1, 0, 2, 0, 0, 1, 1] meaning no 200-coins, one 100-coin, no 50-coins, two 20-coins, no 10-coins, no 5-coins, one 2-coin and one 1-coin.

Question 4

Write a modified version of the greedy scheduling method so that instead of returning the maximum number of events, it returns an array of the `Event` objects chosen, sorted by `startTime`. So, with the example given in the lecture (week 4), it will return:

```
[Event(9,12), Event(13,15), Event(16,19), Event(20,22)]
```

Question 5

Write a Python function:

```
def closestSubset(s,A)
```

that takes an integer `s` and an array of positive integers `A` and returns an array consisting of elements of `A` which add up to `s`. If there is no subset that adds up to `s`, the function should instead return the subset which adds up to the value closest to `s`. For example:

- if `A` is [12, 79, 99, 91, 81, 47] and `s` is 150, it will return [12, 91, 47] as 12+91+47 is 150
- if `A` is [15, 79, 99, 6, 69, 82, 32] and `s` is 150 it will return [69, 82] as 69+82 is 151, and there is no subset of `A` whose sum is 150.

Test the method with arrays generated by `randomIntArray(s,n)` from Lab 3. Try with:

```
A = randomIntArray(20,1000)
subset = closestSubset(5000,A)
```

Hint: You can use a naive recursive solution for this problem. That is, to find the closest subset adding to `s` starting from position `i`, your algorithm should compare the best solution that includes `A[i]` with the best solution that does not include `A[i]`, and so on. Such an algorithm is $O(2^n)$, with n the length of `A`, so it will take a significant amount of time to process an array of length 20, and get overwhelmed for arrays much longer than that.

Question 6

We next try a simple greedy algorithm for the problem of Question 4: start with an empty subset and go through each element in `A`, adding it to the subset if doing that would result in the subset's sum getting closer to the required sum.

For example for `A = [32, 79, 90, 91, 82, 7, 34, 12]` and `s = 150`, the algorithm will start with subset `[]` and:

- it will pick 32 and add it to the set, making it `[32]` (the sum of `[32]` is `32`, which closer to `150` than the sum of `[]` which is 0)
- it will then pick 79 and add it to the set, making it `[32,79]` (sum of `[32,79]` is `111` which is closer to `150` than `32`)
- it will then not pick 90, as sum of `[32,79,90]` is `211` which is further to `150` than `111`

and so on, until it will eventually return `[32,79,7,34]`.

Write code which implements this algorithm, and test it for how close it reaches the best answers and the time it takes compared to the method of Question 4.

This algorithm is $O(n)$, so it is quick, but it may not give the best solution. For example, for `A = [100, 150]` and `s = 150` the algorithm will return `[100]` instead of `[150]`.