CSE 321: Algorithms
Solutions of Homework 3

Fall 2018
Perihan Mirkelam, Scientific Preparation

1. ?

2. n: Number of chips.
   m: Maximum number of taken chips.
   The function “startGame” is calling recursively by getting three parameters as n, m and which player’s turn is.
   startGame(n, m, True)

   Ask user an input number between $1 - m$. Each time the function calls itself by decreasing n according to user input. This recursive call continiuous until $m - 1$ divides remaining chips. Function returns which player’s turn as winner at this base condition.
   startGame(n - input, m, not isFirstUser)

   Best case is to win in first move: This takes constant time.
   Worst case is $n + 1$ divides m and each player takes 1 chip over m moves. Then $n - m$ divides n: This takes m times.
   Time complexity is linear: $\mathcal{O}(m)$

```
In [ ]: n = 19 #Number of chips
        m = 3  #Number of maximum taken chips


        def startGame(n, m, isFirstPlayer):
            print("n ", n, ", m", m)
            if isFirstPlayer :
                player = 'First'
            else:
                player = 'Second'

            if not n % (m+1) == 0:
                ask = str(player) + ' player. How many chips will you get: '
                var = int(input(ask))
                if var > 3 or var < 1 :
                    error = "You can get at least 1 and at most "+ str(m) + " chips."
                    print(error)
                    var = int(input(ask))
                startGame(n-var, m, not isFirstPlayer)
            return player

        print(startGame(n, m, True), "player wins!")
```

3. Method "search" uses binary search algorithm. Since the array is already sorted and has increasing integers, not search for unnecessary nodes by controlling whether $A[i]$ is greater or smaller than i.

   Since we are using binary search, for each call to search the difference between upperBound and lowerBound is halved. Hence, the running time is: $\mathcal{O}(\log n)$

```
In []: import array as arr

         def search(A,lowerBound,upperBound):
             mid = int((upperBound + lowerBound)/2)
             print("mid: ", mid, " A[mid]: ", A[mid], " lowerBound: ", lowerBound, " upperBound
             if lowerBound == upperBound and A[mid] != mid :
                 print("NO. There is no element on A array such that A[i] = i.")
             if A[mid] == mid:
                 print("There exist an element (", mid, "th element) of array A such that A[i]
             if A[mid] > mid :
                 search(A,lowerBound,mid)
             if A[mid] < mid :
                 search(A,mid + 1,upperBound)

         A = arr.array("i", [-4, -2, -1, 0, 4, 6, 7])
         search(A, 0, len(A));
```

```
mid:  3  A[mid]:  0  lowerBound:  0  upperBound:  7
mid:  5  A[mid]:  6  lowerBound:  4  upperBound:  7
mid:  4  A[mid]:  4  lowerBound:  4  upperBound:  5
There exist an element ( 4 th element) of array A such that A[i] = i = 4
```

4. Method "calcMaxSubArraySum" is calling recursively each time divided by 2 for two halves. Therefore, recurrence relation will be,
   $T(n) = 2T(n/2) + f(n)$
   By Master Theorem, time complexity of our algorithm: $\Theta(n \log n)$

```
In []: import sys

         # Calculated max sum is maximum of three cases.
         # 1. Calculate right half and left half recursively.
         # 2. Plus calculate sub array(first time whole array)
         # include middle point and its both side of sum
         def calcMaxSubArraySum(arr, left, right) :

             if (left == right) :
                 return arr[left]

             mid = int((left + right) / 2)

             leftside = calcMaxSubArraySum(arr, left, mid)
             rightside = calcMaxSubArraySum(arr, mid+1, right)

             #calculate max sum of left and right side of mid
             temp_left = 0
             left_sum = -sys.maxsize

             for i in range(mid, left-1, -1) :
                 temp_left = temp_left + arr[i]
                 left_sum = max(left_sum, temp_left)

             temp_right = 0
             right_sum = -sys.maxsize
             for i in range(mid + 1, right + 1) :
                 temp_right = temp_right + arr[i]
                 right_sum = max(right_sum, temp_right)

             centered = left_sum + right_sum;

             return max(leftside, rightside, centered)

         A = [5, -6, 6, 7, -6, 7, -4, 3]
         maxSum = calcMaxSubArraySum(A, 0, len(arr)-1)
         print("Max sum of subset is: ", maxSum)
```

```
Max sum of subset is:  14
```

5. The function “match” is calling recursively by getting two parameters “text” as a string and “pattern” as a string list.

   When the function is called first, the pattern will have a pattern list with at least one element:
   match(’Tobeornottobe’, [A, B, C, D, A, B, C])

   Then the function will be called recursively in a loop so that the "pattern" list with an exact one element and part of the "text":
   match(’be’, [B])

   Let n: Length of “text” string.
   Let p: Number of pattern list items.
   Let m_i: Size of ith string item of pattern list.
   Let M: Total count of “pattern” list items.

$$M = \sum_{i=0}^{p-1} m_i$$

   Time complexity is linear: $\mathcal{O}(M) = \mathcal{O}(n)$

```
In [ ]: text = 'Tobeornottobe'
        A = 'to'
        B = 'be'
        C = 'or'
        D = 'not'
        pattern = [A, B, C, D, A, B]

        def match(text, pattern):
            textLength = len(text)
            patternCharLength = 0

            for pat in pattern:
                patternCharLength += len(pat)

            #text length and total length of pattern items must match
            if len(pattern) == 1 or patternCharLength == textLength:
                startIndex = 0
                endIndex = 0
                if len(pattern) > 1:
                    #this part works when the function is called for the first time
                    #the recursive call for each pattern element
                    for i in range(len(pattern)):
                        partOfPattern = [pattern[i]]
                        endIndex = startIndex + len(pattern[i])
                        if not match(text[startIndex: endIndex], partOfPattern):
```

```
                    print("Not match text part: ", text[startIndex:endIndex], " partOf
                    return False
                startIndex = endIndex

        #this part works for each element of pattern list after the function called fo
        #complexity is m_i
        if len(pattern) == 1:
            patternItem = pattern[0]
            for i in range(len(patternItem)):
                if not patternItem[i].lower() == text[i].lower():
                    return False
           # print(patternItem, " matches to ", text)
        return True
    else:
        print("Not valid sizes. Pattern sizes are greater than text size!")
        return False

print(text, " matches to ", pattern, ": ", match(text, pattern))
```