Assignment 5

1. In a previous life, you worked as a cashier in the lost Antarctican colony of Nadiria, spending the better part of your day giving change to your customers. Because paper is a very rare and valuable resource in Antarctica, cashiers were required by law to use the fewest bills possible whenever
they gave change. Thanks to the numerological predilections of one of its founders, the currency of Nadiria, called Dream-Dollars, was available in the following denominations: $1, $4, $7, $13, $28, $52, $91, and $365.19

(a) The greedy change algorithm repeatedly takes the largest bill that does not exceed the target amount. For example, to make $122 using the greedy algorithm, we first take a $91 bill, then a $28 bill, and finally three $1 bills. Give an example where this greedy algorithm uses more Dream-Dollar bills than the minimum possible. [Hint: It may be easier to write a small program than to work this out by hand.]

Let's take the target amount of $144.
Greedy algo will start with $144 ,the largest bill that does not exceed $144 is $91. Subtract $91 from the total, leaving us with $53.The largest bill that does not exceed $53 is $52. Subtract $52, and we have $1 left.
Now, the algorithm has no choice but to give four $1 bills to make up the remaining $4.

Using the greedy algorithm, we have used a total of 6 bills (1x $91, 1x $52, and 4x $1) to make $144. However, we can do better by using two $52 bills, which sum up to $104, and two $7 bills for the remaining $40. This totals only 4 bills (2x $52 and 2x $7), which is fewer than the greedy algorithm's result.

So, the greedy algorithm uses more Dream-Dollar bills than the minimum possible for the target amount of $144.

(b) Describe and analyze a recursive algorithm that computes, given an integer k, the minimum number of bills needed to make k Dream- Dollars. (Don't worry about making your algorithm fast; just make sure it's correct.)

```
def minBills(k, denominations):
    if k == 0:
        return 0
    if k < 0 or not denominations:
        return float('inf')
    min_bills = float('inf')
    for denom in denominations:
        sub_problem = minBills(k - denom, denominations)

        if sub_problem != float('inf'):
            min_bills = min(min_bills, 1 + sub_problem)
    return min_bills
k = 48
denominations = [365, 91, 52, 28, 13, 7, 4, 1]
```

```
result = minBills(k, denominations)
print(f"The minimum number of bills needed to make ${k} is {result}.")
#this will result as 3
```

(c) Describe a dynamic programming algorithm that computes, given an integer k, the minimum number of bills needed to make k Dream-Dollars.

```
def min_dynamic(k, denominations):
    min_count = [float('inf')] * (k + 1)
    min_count[0] = 0
    for i in range(1, k + 1):
        for denom in denominations:
            if i >= denom:
                min_count[i] = min(min_count[i], 1 + min_count[i - denom])
    return min_count[k]
denominations = [365.19, 91, 52, 28, 13, 7, 4, 1]
target = 48
min_bills = min_dynamic(target, denominations)
print(f"Minimum number of bills needed to make ${target}: {min_bills}")
#This will results as 3
```

**2. Describe efficient algorithms for the following variants of the text segmen-tation problem. Assume that you have a subroutine IsWord that takes an array of characters as input and returns True if and only if that string is a “word”. Analyze your algorithms by bounding the number of calls to
IsWord.**

(a) Given an array A[1 .. n] of characters, compute the number of partitions of A into words. For example, given the string ARTISTOIL, your algorithm should return 2, for the partitions ARTIST·OIL and ART·IS·TOIL.

```
def countWordPartitions(A, is_word_func):
    n = len(A)
    dp = [0] * (n + 1)
    dp[0] = 1
    for i in range(1, n + 1):
        for j in range(i, 0, -1):
            if is_word_func(A[j - 1:i]):
                dp[i] += dp[j - 1]
    return dp[n]

def IsWord(word):
    valid_words = {"ART", "IS", "TOIL", "OIL", "ARTIST"}
    return word in valid_words
A = "ARTISTOIL"
result = countWordPartitions(A, IsWord)
print(result)
```

This will results as 2

(b) Given two arrays A[1 .. n] and B[1 .. n] of characters, decide whether A and B can be partitioned into words at the same indices. For example, the strings BOTHEARTHANDSATURNSPIN and PINSTARTRAPSANDRAGSLAP can be partitioned into words at the same indices as follows:
BOT·HEART·HAND·SAT·URNS·PIN
PIN·START·RAPS·AND·RAGS·LAP

```
def canPartitionWordsAtSameIndices(A, B, is_word_func):
    n = len(A)
    dp = [False] * (n + 1)
    dp[0] = True
    for i in range(1, n + 1):
        for j in range(i, 0, -1):
            if dp[j - 1] and is_word_func(A[j - 1:i]) and is_word_func(B[j - 1:i]):
                    dp[i] = True
                    break

    return dp[n]
def IsWord(word):
    valid_words = {"BOT", "HEART", "HAND", "SAT", "URNS", "PIN", "START", "RAPS","AND", "RAGS",
"LAP"}
    return word in valid_words

A = "BOTHEARTHANDSATURNSPIN"
B = "PINSTARTRAPSANDRAGSLAP"
result = canPartitionWordsAtSameIndices(A, B, IsWord)
print(result)
```

This will result as true

(c) Given two arrays A[1 .. n] and B[1 .. n] of characters, compute the number of different ways that A and B can be partitioned into words at the same Indices.

```
def IsWord(word, word_set):
    return word in word_set
def countWordPartitions(A, B, word_set):
    n = len(A)
    DP = [[0] * (n+1) for _ in range(n+1)]
    DP[0][0] = 1
    for i in range(1, n+1):
        for j in range(1, n+1):
            if IsWord(A[i-1], word_set) and IsWord(B[j-1], word_set): DP[i][j] = DP[i-1][j-1] + DP[i-1][j] +
DP[i][j-1]
            else:
                DP[i][j] = 0
A = "BOTHEARTHANDSATURNSPIN"
B = "PINSTARTRAPSANDRAGSLAP"
```

```
dictionary = {"BOT", "HEART", "AND", "SAT", "URNS", "PIN", "START", "RAPS", "RAGS", "LAP"}
result = countWordPartitions(A, B, dictionary)
print("Number of ways to partition A and B into words:", result)
```

3. **Suppose you are given an array A[1 .. n] of numbers, which may be positive,negative, or zero, and which are not necessarily integers.**
(a) Describe and analyze an algorithm that finds the largest sum of elements in a contiguous subarray A[i .. j].
(b) Describe and analyze an algorithm that finds the largest product of elements in a contiguous subarray A[i .. j].

For example, given the array [−6, 12,−7, 0, 14,−7, 5] as input, your first algorithm should return 19, and your second algorithm should return 504.

Given the one-element array [−374] as input, your first algorithm should return 0, and your second algorithm should return 1. (The empty interval is still an interval!) For the sake of analysis, assume that comparing, adding, or multiplying any pair of numbers takes O(1) time.

[Hint: Part (a) has been a standard computer science interview question since at least the mid-1980s. You can find many correct solutions on the web; the problem even has its own Wikipedia page! But at least in 2016, a significant fraction of the solutions I found on the web for part (b) were either slower than necessary or actually incorrect.]

A) Kadane's Algorithm is a widely used dynamic programming technique for finding the maximum sum of elements in a contiguous subarray within an array of numbers.

1. Initialize two variables, `max_sum` and `current_sum`, to the value of the first element in the array. `max_sum` will keep track of the maximum subarray sum found so far, and `current_sum` will keep track of the current subarray sum.

2. Iterate through the array.

3. For each element, calculate the running sum by adding the current element to `current_sum`.

4. Update `current_sum` by taking the maximum of the current element and the sum of the current element and `current_sum`. This step essentially determines whether it's better to start a new subarray at the current element or extend the existing subarray.

5. Update `max_sum` with the maximum value between `max_sum` and `current_sum`. This step keeps track of the largest subarray sum encountered during the iteration.

6. Repeat steps 3-5 for all elements in the array.

7. Finally, the value of `max_sum` will represent the largest sum of elements in a contiguous subarray within the input array.

Kadane's Algorithm has a time complexity of O(n), where n is the number of elements in the input array.

Code:-
```
def max_arr(arr):
    m_sum = arr[0]
    current_sum = arr[0]
    for num in arr[1:]:
        current_sum = max(num, current_sum + num)
        m_sum = max(m_sum, current_sum)

    return m_sum

arr = [-6, 12, -7, 0, 14, -7, 5]
result = max_arr(arr)
print("Maximum subarray sum:", result)
```

So, the largest sum of elements in a contiguous subarray of [−6, 12, −7, 0, 14, −7, 5] is 19, which corresponds to the subarray [12, -7, 0, 14].

B)
To find the largest product of elements in a contiguous subarray A[i .. j], we can use an algorithm that keeps track of both the maximum and minimum products ending at each position in the array. This approach is necessary because multiplying a negative number by a negative number results in a positive product, and we need to consider the possibility of switching between maximum and minimum products.

Code:-

```
def max_arr(arr):
    if not arr:
        return 0
    n = len(arr)
    max_end = arr[0]
    min_end = arr[0]
    max_product = arr[0]

    for i in range(1, n):
        temp = max_end
        max_end = max(arr[i], max_end * arr[i], min_end * arr[i])
        min_end = min(arr[i], temp * arr[i], min_end * arr[i])

        max_product = max(max_product, max_end)
    return max_product
```

```
arr = [-6, 12, -7, 0, 14, -7, 5]
result = max_arr(arr)
print("Largest product of contiguous subarray:", result)
```

So, the largest product of elements in a contiguous subarray of [−6, 12, −7, 0, 14, −7, 5] is 504, which corresponds to the subarray [12, -7, 0, 14].

Since the algorithm has a linear relationship with the size of the input array and performs a constant amount of work for each element, its time complexity remains O(n), making it an efficient solution for finding the largest product of a contiguous subarray within the array.

202311031