Problem 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 currence of Nadiria, called Dream-Dollars, was available in the following denominations: $1, $4, $7, $13, $28, $52, $91, and $365.

1. 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 3 $1 bills. Give an example where this greedy algorithm uses more Dream-Dollar bills than the minimum possible.

   When Trying to make the total $455, the two algorithms produce different results

$$\text{Greedy Algorithm} \rightarrow \{365, 28, 7, 1, 1, 1\} = 7 \text{ bills}$$
$$\text{Dynamic Programming} \rightarrow \{91, 91, 91, 91, 91\} = 5 \text{ bills}$$

2. 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.)

   This problem is analogous to the subset sum problem, just with the freedom that we can now "draw" from the pool of values more than once. Therefore, we have the following recursive function:

$$bills = \{1, 4, 7, 13, 28, 52, 91, 365\}$$
$$\text{MinPaper}(\text{bills}, \text{total}) = \begin{cases} 0 & \text{if total} = 0 \\ 1 + \min\limits_{B \in \text{bills}}{}^{1} \{\text{MinPaper}(\text{bills}, \text{total} - B)\} & \text{Otherwise} \end{cases}$$

   In other words, we have a brute force algorithm that "guesses" the bill to take, and then tries to find the minimum number of bills it takes to make up the remaining sum. The implementation, in python notation, is as follows:

```
def MinPaper(bills, total):
    minimum = inf
    # Base case - our total is 0
    if total == 0:
        return 0

    #iterate across each type of bill
    for B in bills:
        if B <= total:
            # Check every possible
            # combination of bills
            possible = MinPaper(bills, total-B)
            # If we've found a smaller
            # amount, then use that
            if possible <= minimum:
                minimum = possible
    return 1 + minimum
```

[1] Note that the following must be true: $B \leq \text{total}$

3. Describe a dynamic programming algorithm that computes, given an integer k, the minimum number of bills needed to make k Dream-Dollars. (This one needs to be fast).

   Notice that when we turn this into a dynamic programming algorithm we need to only store one thing: The minimum number of bills to make the numerical amount from n = \$0 to n = \$total. Therefore we can create a memoization table to store all this, which we will lazily name "minarray". In order to calculate the minimum number of bills for some value we will label "t" (short for "total") we have to move iteratively across the minarray for all t+1 entries.

   At an arbitrary entry n into minarray, we can calculate it's value as being the minimum of all entries before it that can be reached by subtracting the current total n minus the amount of all bills in the set of bills we have. It is explained in the following recurrence relationship (*notation note: $|bills|$ denotes the number of different types of bills we have to work with, and $bills[i]$ denotes the value of the bill at index i, for example, $bills[1] = 4$*):

$$bills = \{1, 4, 7, 13, 28, 52, 91, 365\}$$

$$S = \{S \in \{1, ..., |bills|\} \text{ s.t. } bills[S] \leq n\}$$

$$\text{minarray}[n] = \begin{cases} 0 & \text{if } n = 0 \\ 1 + \min\limits_{B \in S} \{\text{minarray}[n - bills[B]]\} & \text{Otherwise} \end{cases}$$

   The algorithm, when implemented in python, is as follows:

```
inf = float('inf')
def MinPaperDP(bills, total):
    # minarray[n] stores the minimum number of
    # bills to make total=n
    minarray = [inf for i in range(total+1)]

    # set total = 0 to require
    # 0 bills
    for i in range(len(bills)):
        minarray[0] = 0

    # iterate across each possible total n
    # and each possible bill beta
    for n in range(1, total+1):
        for beta in range(len(bills)):
            if bills[beta] > n:
                break
            # check the minimum number of bills if
            # we use bill beta
            possible = 1 + minarray[n - bills[beta]]
            # if it's smaller, update the minarray
            if possible < minarray[n]:
                minarray[n] = possible

    return minarray[n]
```

Space Analysis

Our memoization table, simply named minarray, is of size n where n denotes the total we are trying to make. If we are trying to make \$20, then $n = 20$. Because of this, our space usage is simply $O(n)$.

Time Analysis

Notice that in our dynamic programming algorithm we move across minarray from left to right, and at each entry we perform at most β comparisons where β denotes the number of different types of bills we have. Because of this, our time usage is then $O(\beta n)$.

Usage Summary

Space Usage	Time Usage
$O(n)$	$O(n\beta)$

Problem 2

Design and analyze an efficient algorithm to break a list of words into lines while minimizing the total slop, i.e. the sum of slop on each line except the last. Your algorithm will be given a list of word lengths and will output the minimum slop achieved.

Whenever working with the input string of words, we can break this problem up into a series of smaller problems recursively. Therefore, if we have a list of words of length n, we can choose somewhere to start a new line, and then with the remaining words choose the way to break those up in a manner that results in the smallest amount of slop

Suppose we have the function $slop(i, j, L, \ell)$ that computes the minimum slop of all words at the index i through the index j. Suppose we also have the function $CalcSlop(i, j, L, \ell)$ which performs the following operation:

$$CalcSlop(i, j, L, \ell) = \begin{cases} 0 & \text{when } j \leq i \\ \left(L - (j - i) - \sum_{k=i}^{j} \ell[k]\right)^3 & \text{otherwise} \end{cases}$$

We can calculate the slop recursively using the following relation:

$$\text{slop}(i, j, L, \ell) = \begin{cases} +\infty & \text{when } CalcSlop(i, j, L, \ell) < 0 \\ 0 & \text{when } (j = n) \wedge (CalcSlop(i, j, L, \ell) \geq 0)^2 \\ \min\limits_{k \in \{i+1, \ldots, j\}} \{CalcSlop(i, k, L, \ell) + slop(k, j, L, \ell)\} & \text{otherwise} \end{cases}$$

Notice that we can memoize this relation and turn it into a dynamic programming algorithm. Although instead of working from the front of the list forwards, we work from the back of the list backwards. We can define an array of length n (remember: n = length of the wordlist we're trying to make a paragraph of), aptly called *sloparray* where we can record the minimum slop for an increasing number of words in ℓ. For our *sloparray*, *sloparray*[0] denotes the minimum slop for all words (i.e. we calculating at the 0th index), and *sloparray*$[n-1]$ denotes the minimum slop for only one word (i.e. the last word in our array). Using this, we find that the slop can be calculated recursively as follows:

In other words, this means that we start from the last word, calculate the slop, add the next word, calculate their minimum slop together, and repeat this process until all words have been added to the paragraph. Assuming we memoize this using the *sloparray*, we find the following recursive relationship (remember we work from $i = n$ to $i = 0$):

$$S(i) = \{S \in \{i + 1, ..., n\} \; s.t. \; CalcSlop(i, S, L, \ell) \geq 0\}^3$$

$$sloparray[i] = \begin{cases} 0 & \text{when } i = n \\ \min\limits_{j \in S(i)} \{CalcSlop(i, j, L, \ell) + sloparray[j + 1]\} & \text{otherwise} \end{cases}$$

The algorithm when implemented in python, is as follows on the next page...

[2] This means that we are at the last line, which should always return 0 slop (assuming that it fits).

[3] In other words, S(i) is the set of ranges over our words where we can select a group of words starting at $\ell[i]$ that does not violate the rule we go over our space limit L.

```
inf = float('inf')

def calcSlop(L, i, j, l):
    # set our initial sum to 0
    s = 0
    # Make sure our bounds are sane
    if j < i:
        return 0
    else:
        # sum up word lengths
        for k in range(i, j):
            s += l[k]
    # calculate & return slop
    s = (L - (j - i) - s)**3
    return s

def minSlopDP(l, L):
    # initialize our slop array
    sloparray = [inf for i in range(len(l))]
    # iterate from the last entry in our
    # word list to the very first
    for i in range(len(l)-1, -1, -1):
        # i through j is the line we are currently looking at
        for j in range(i, len(l)):
            # if j is the final word in our list and it fits
            # into one single line, then we can set our slop
            # as being 0
            if (j >= len(l)-1) and (calcSlop(L, i, len(l)-1, l) >= 0):
                sloparray[i] = 0
                break
            # if we have too many words and we over-fill a line
            # then it is time to stop and move onto the next line
            elif calcSlop(L, i, j, l) < 0:
                break
            # otherwise see the minimum slop if we choose our line
            # to be i through j
            else:
                possible = calcSlop(L, i, j, l) + sloparray[j+1]
                sloparray[i] = min([possible, sloparray[i]])

    # return the minimum slop
    return sloparray[0]
```

Space Analysis

Our space analysis is easy. Because our sloparray is of size n, where n is the number of words we want to put into our paragraph, our space usage is $\Theta(n)$.

Time Analysis

Our time analysis is a little more involved but still fairly straightforward. We work from the back of the array to the front of the array. This takes n steps. At each index k in *sloparray* we perform *at most* $n - i$ comparisons. We start at index $n - 1$ which takes 0 comparisons, $n - 2$ which takes 1 comparison, all the way up to index 0 which takes at most $n - 1$ comparisons. Therefore, our number of comparisons is then bounded by:

$$\begin{aligned} &\sum_{i=0}^{n-1} n \\ &= \frac{(n-1)((n-1)+1)}{2} \\ &= \frac{n^2 - n}{2} \\ &\in o(n^2) \end{aligned}$$

One important thing to note is that this is simply the number of comparisons, calculating the slop at each comparison takes time itself. My *CalcSlop*() function takes *at most* n steps (as most n elements must be summed together). Therefore, our high bound on the runtime is then:

$$\begin{aligned} &\frac{n^2 - n}{2} \times n \\ &= \frac{n^3 - n^2}{2} \\ &\in o(n^3) \end{aligned}$$

And this is assuming our *CalcSlop*() function is fairly inefficient. Assuming we can push this limit down we can get a faster runtime. Again, remember that this is a very high, extreme case bound for our runtime.

Time Usage	Space Usage
$o(n^3)$	$\Theta(n)$

Problem 3

1. Modify this implementation to compute the bounded edit distance. Your modification should be minor; in particular you should not use dynamic programming or memoization. But your modified algorithm should run faster than the original one, at least in some cases. Do not analyze the algorithm's runtime complexity.

   We can imagine the way this recursive algorithm runs as if it were a tree. As we work our way down a single branch of the tree we can imagine this path as the unique sequence of recursive calls to our own function.

   When running this recursive algorithm notice that each time we recurse we add one to our edit distance (assuming the last two elements aren't identical). Because we have a bound on our allowable edit distance, once this sum surpasses our bound there is no use continuing to recursively call our function - as we know that our edit distance can only get bigger the longer we recurse

   Therefore, if we can keep track of our edit distance as we move down the "recursion tree" we can know when to call it quits and when to keep on going. This will save a significant amount of processing time because when we know that our edit distance has surpassed our bound (when traversing down a single, unique path in the recursion tree) there is no more use in continuing down that path, we simply return infinity and move on.

   We can define the modified function as $edit\_recursive\_bound(S, T, rs, B)$ (shortened $erb(S, T, rs, B)$) where S and T are the strings we are comparing, rs is the "rolling sum" that we keep track of at each level, and B is our bound. Our new recursive formula is defined as follows:

$$NEQ(S,T) = \begin{cases} 0 & \text{when } T[-1] \neq S[-1] \\ 1 & \text{when } T[-1] = S[-1] \end{cases}$$

   As shorthand notation to allow for more space, define $N = NEQ(S, T)$

$$erb(S,T,rs,B) = \begin{cases} +\infty & \text{when } rs > B \\ len(T) & \text{when } len(S) = 0 \ \wedge \ len(T) \leq B \\ +\infty & \text{when } len(S) = 0 \ \wedge \ len(T) > B \\ len(S) & \text{when } len(T) = 0 \ \wedge \ len(S) \leq B \\ +\infty & \text{when } len(T) = 0 \ \wedge \ len(S) > B \\ min \begin{cases} erb(S[:-1], T, rs+1, B) + 1 \\ erb(S, T[:-1], rs+1, B) + 1 \\ erb(S[:-1], T[:-1], rs+N, B) + N \end{cases} & \text{otherwise} \end{cases}$$

The recursive algorithm is as follows:

```
# S = first string
# T = second string
# rs = "rolling sum" to keep track of
#      the edit distance so far
# B = The bound we have on our edit
#     distance
def edit_recursive_bound(S, T, rs, B):
    # Check if we have surpassed our
    # bound and decide whether or not
    # we should continue
    if rs > B:
        return float('inf')

    if len(S) == 0:
        # insert all characters in T
        if len(T) > B:
            return float('inf')
        else:
            return len(T)
    if len(T) == 0:
        # delete all characters in S
        if len(S) > B:
            return float('inf')
        else:
            return len(S)
    # cost to delete one char of S
    del_cost = edit_recursive_bound(S[:-1], T, rs+1, B) + 1
    # cost to insert one char of T
    ins_cost = edit_recursive_bound(S, T[:-1], rs+1, B) + 1
    if S[-1] == T[-1]:
        # zero cost to match chars
        match_cost = edit_recursive_bound(S[:-1], T[:-1], rs, B)
    else:
        match_cost = edit_recursive_bound(S[:-1], T[:-1], rs+1, B) + 1
    return min(del_cost, ins_cost, match_cost)
```

2. Design and analyze a dynamic programming algorithm for bounded edit distance. Your algorithm should be $o(n^2)$ when run on two strings of length n, assuming that the bound is $o(n)$.

   From the problem description we know that both strings (labeled S and T) are both of length n. Therefore, we can create a memoization table of size $(n+1) \times (n+1)$ and simply name it *memo* where $memo[i][j]$ means the edit distance for strings $S[:i]$ and $T[:j]$. When thinking about it in table format this means that S takes up the y-axis and T takes up the x-axis. (*notation note:* $S[:0]$ *means the empty string and* $S[:n+1]$ *means the entire string of S).*

   Our recursive formula is as follows:

$$memo[i][j] = \begin{cases} +\infty & \text{when } i = 0 \ \wedge \ j > B \\ j & \text{when } i = 0 \ \wedge \ j \leq B \\ +\infty & \text{when } j = 0 \ \wedge \ i > B \\ i & \text{when } j = 0 \ \wedge \ i \leq B \\ min \begin{cases} memo[i-1][j] + 1 \\ memo[i][j-1] + 1 \\ memo[i-1][j-1] + [S[i] \neq T[j]] \end{cases} & \text{otherwise} \end{cases}$$

   We can use a lazy cop-out algorithm and calculate the edit as usual and then if the final edit distance is greater than our given bound we simply return $+\infty$ and otherwise we return the edit distance. The algorithm is as follows:

```
# S = first string
# T = second string
# B = edit distance bound
def ed_bound_DP(S, T, B):
    inf = float('inf')
    # initialize the memo array
    memo = [[inf for i in range(len(T)+1)] for j in range(len(S)+1)]
    for i in range(len(S)+1):
        for j in range(len(T)+1):
            # edge case on the top edge
            if i == 0:
                memo[i][j] = j
                continue
            # edge case on the left edge
            elif j == 0:
                memo[i][j] = i
                continue
            # otherwise iterate through the array and
            # get the edit distance
            else:
                diff = 1
                if S[i - 1] == T[j - 1]:
                    diff = 0

                memo[i][j] = min([memo[i - 1][j] + 1,
                                  memo[i][j - 1] + 1,
```

```
                                    memo[i - 1][j - 1] + diff])

    # return the edit distance or
    # infinity if it is above the
    # bound
    if memo[-1][-1] > B:
        return inf
    else:
        return memo[-1][-1]
```

Space Analysis

If S is of length m and T is of length n, then our memo array is of size $(m+1) \times (n+1) = mn + n + m + 1 \in \Theta(mn)$. Therefore, our space usage is $\Theta(mn)$.

Time analysis

So we have a memoization table of size $mn+m+n+1$ which we iterate through each element once. Each computation at each element in the array takes $\emptyset(1)$ time, and we have one final comparison at the end that takes $O(1)$ time, therefore our total runtime is $(mn+m+n+1) \times O(1) + O(1) \in \Theta(mn)$. Therefore, our runtime is of $\Theta(mn)$.