Anooj Pai
Algo HW 8
04/10/2023

6.1) A *contiguous subsequence* of a list S is a subsequence made up of consecutive elements of S. For instance, if S is

$$5, 15, -30, 10, -5, 40, 10,$$

then 15, −30, 10 is a contiguous subsequence but 5, 15, 40 is not. Give a linear-time algorithm for the following task:

Input: A list of numbers, $a_1, a_2, \ldots, a_n$.

Output: The contiguous subsequence of maximum sum

For the preceding example, the answer would be 10, −5, 40, 10, with a sum of 55.
(*Hint:* For each $j \in \{1, 2, \ldots, n\}$, consider contiguous subsequences ending exactly at position j.)

Answer:

```
def max_Sub_Seq(a):
   S = [ 0 for i in a ]
   I = [ 0 for i in a ]
   for i in range(len(a)):
      if a[i] + S[i - 1] > a[i]:
         S[i] = a[i] + S[i - 1]
         I[i] = I[i - 1]
      else:
         S[i] = a[i]
         I[i] = i

   max = 0
   for i in range(len(a)):
      if S[i] > S[max]:
         max = i

   return a[I[i]:max+1]
```

Runtime: O(n)

6.4)Youaregivenastringofncharacterss[1...n],whichyoubelievetobeacorruptedtextdocument in which all punctuation has vanished (so that it looks something like “itwasthebestoftimes...”). You wish to reconstruct the document using a dictionary, which is available in the form of a Boolean function dict(·): for any strin/g w,
true if w is a valid word false otherwise.
(a) Give a dynamic programming algorithm that determines whether the string s[·] can be reconstituted as a sequence of valid words. The running time should be at most O(n2), assuming calls to dict take unit time.

(b) In the event that the string is valid, make your algorithm output the corresponding sequence of words.

Answer:

```
def valid_Strings(s):
    V = [ False for i in s ]
    w = []
    for i in range(len(s)):
        for j in range(i - 1, -1, -1):
            if ''.join(s[j:i+1]) in dict_ and (j == 0 or V[j-1]):
                V[i] = True
                w.append(''.join(s[j:i+1]))
                break
            else:
                print('i=%d j=%d string=%s V[j-1]=%s V[i]=%s' % (i, j, ''.join(s[j:i+1]), V[j-1], V[i]))

    return (True, w) if V[len(s) - 1] else (False, None)


if __name__ == "__main__":
    # part a
    s = 'itwasthebestoftimes'
    v, w = valid_words(list(s))
    print('s=[%s] Valid words: %s => %s' % (s, v, w))

    # part b
    s = 'aitwasthebestoftimesa'
    v, w = valid_words(list(s))
    print('s=[%s] Valid words: %s => %s' % (s, v, w))
```

6.17)
Given an unlimited supply of coins of denominations x1, x2, . . . , xn, we wish to make change for a value v; that is, we wish to find a set of coins whose total value is v. This might not be possible: for instance, if the denominations are 5 and 10 then we can make change for 15 but not for 12. Give an O(nv) dynamic-programming algorithm for the following problem.
Input: x1,...,xn;v.
Question: Is it possible to make change for v using coins of denominations x1, . . . , xn?

Answer:

```
def get_Change(x, v):
    n = len(x)

    C = [False for i in range(v+1)]
    C[0] = True

    for i in range(1, v+1):
        for j in range(n):
            if i >= x[j]:
                C[i] = C[i-x[j]]
                if C[i]:
                    break

    return C[v]

if __name__ == "__main__":
    x = [5, 10]
    v = 15
    c = coin_change_unlimited(x, v)
    print('x=%s v=%d change: %s' % (x, v, c))

    x = [5, 10]
    v = 12
    c = coin_change_unlimited(x, v)
    print('x=%s v=%d change: %s' % (x, v, c))
```

6.19)
Here is yet another variation on the change-making problem (Exercise 6.17).
Given an unlimited supply of coins of denominations x1, x2, . . . , xn, we wish to make change for a value v using at most k coins; that is, we wish to find a set of ≤ k coins whose total value is v. This might not be possible: for instance, if the denominations are 5 and 10 and k = 6, then we can make change for 55 but not for 65. Give an efficient dynamic-programming algorithm for the following problem.
Input: x1,...,xn; k; v.
Question: Is it possible to make change for v using at most k coins, of denominations x1,...,xn?

Answer:

```
def minCoins(coins,V, k):
    M = [sys.maxint for x in range(V+1)]
    M[0] = 0
    m = len(coins)
    for i in range(1, V+1):
        for j in range(m):
            if(coins[j] <= i):
                sub = M[i - coins[j]]
                if(sub != sys.maxint and sub + 1 < M[i]):
                    M[i] = sub+1

    print M
    return M[V]

if __name__ == '__main__':
    coins =  [5, 6, 9, 1];
    k = 6
    V = 11;
    print "Is possible ?  ", minCoins(coins, V, k);
```

B)Consider the “Weighted Interval Scheduling” problem discussed in the class with the following requests
R={ r1,r2,r3,r4,r5,r6} where
Start time Finish time Value
r1: (1 3 2)
r2: (2 5 4)
r3: (4 5 3)
r4: (2 7 7)
r5: (6 8 2)
r6: (6 9 1)
1. [20 pnts] Implement an iterative DP Algorithm to find the subset of requests that has the total maximum value. (Hint: us an array M to store the values as described in the class)

2. [20 pnts] implement an algorithm and print the list of intervals in the optimal solution above by using
the array M without maintaining another data structure.

1)
```
R = [(1, 3, 2), (2, 5, 4), (4, 5, 3), (2, 7, 7), (6, 8, 2), (6, 9, 1)]
n = len(R)

M = [0] * (n+1)
M[1] = R[0][2]

for i in range(2, n+1):
    j = i - 1
    while j > 0 and R[j-1][1] > R[i-1][0]:
        j -= 1
    M[i] = max(M[i-1], M[j] + R[i-1][2])

print("Maximum value:", M[n])

# Backtracking to find the subset of requests
optimal_subset = []
i = n
while i > 0:
    if M[i] > M[i-1]:
        optimal_subset.append(i-1)
        i -= 2
    else:
        i -= 1

optimal_subset.reverse()
print("Optimal subset of requests:", optimal_subset)
```

2)
```
def print_optimal_intervals(R, M):
    n = len(R)
    optimal_intervals = []
    last_interval_end = -1
    for i in range(n, 0, -1):
        if M[i] != M[i - 1]:
            last_interval_end = R[i - 1][1]
            optimal_intervals.append(R[i - 1])
        elif R[i - 1][1] <= last_interval_end:
            optimal_intervals.pop()
            last_interval_end = optimal_intervals[-1][1] if optimal_intervals else -1
    optimal_intervals.reverse()
    for interval in optimal_intervals:
```

```python
        print(interval)

R = [(1, 3, 2), (2, 5, 4), (4, 5, 3), (2, 7, 7), (6, 8, 2), (6, 9, 1)]
selected_intervals = weighted_interval_scheduling(R)
M = compute_optimal_value(R)
print_optimal_intervals(R, M)
```