Ex. 1

The complete matrix that is filled to compute the maximum possible total value:

```
    |   c   0   1   2   3   4   5
------------------------------
i   |
    |
0   |       0   0   0   0   0   0
1   |       0   0   0   8   8   8
2   |       0   0   3   8   8   11
3   |       0   0   3   8   9   11
4   |       0   6   6   9   14  15
```

The reconstruction algorithm runs as follows:

```
i=4, c=5:
s4 = 1 <= c
    A[i-1, c-s4] + v4 = 9 + 6 > 11 = A[i-1,c]
    So add 4 to S and i-- and c = c - s4 = 4

i = 3, c = 4
s3 = 4 <= c
    A[i-1,c-s3] + v3 = 0+9 > 8 = A[i-1,c]
    So add 3 to S and i-- and c = c - s3 = 4-4 = 0

i = 2, c = 0
    c = 0, so skip

i = 1, c = 0
    c = 0, so skip

return S = {3,4}
```

Ex. 2

I decided to write (and debug...) the algorithm in Python code. For the arrays, I used numpy. **The proof of correctness and complexity analysis is all present! You can find it in the comments.**

You can copy this script and run it if you want. I included an example problem at the bottom (it is from the slides of this week).

```python
import numpy as np

def matMul(p):
    '''
    Given matrices of dimension
```

```
    A_0:    p[0] by p[1]
    A_1:    p[1] by p[2]
    ...
    A_n-2:  p[n-2] by p[n-1]

    where n = len(p)

    define the n-1 by n-1 matrices m and s as follows:
    m[i,j] = minimum number of operations needed to multiply A_i ... A_j
optimally.
    s[i,j] = the k in [i, j-1] such that the top-level bracketing (A_i ...
A_k)(A_k ... A_j) gives the optimal order of operations.

    Note that these numbers only make sense for j >= i. For j < i, we leave
these matrices undefined and don't care about their contents

    They also satisfy the recursion:
    m[i,i] = 0 for all i

    m[i,j] = min {m[i, k] + m[k + 1, j] +  + nrows[i] * ncols[k] * ncols[j]
| i <= k < j}
    s[i,j] = argmin {m[i, k] + m[k + 1, j] +  + nrows[i] * ncols[k] *
ncols[j] | i <= k < j}


    Where, for **clarity** (there was so much trouble with indexing due to
1 based/ 0 based differenced XD),
    ncols and nrows are just views of p[1:], p[:-1], respectively.

    The order of computation is important: we need to iterate over i,j
correctly to make sure that all memoized values are already present.
    This actually means, quite simply, that we can only compute m[i,j] for
longer subsequences A_i ... A_j once we know m[i,j] for all shorter
    subsequences. I.e., we know at first only how many multiplications it
takes to compute A_0, A_1, ... A_n-2 (all take 0 multiplications!).
    Next, we can calculate the number for A_1 A_2, A_2 A_3, ... A_n-3 A_n-
2.
    Next, for A_1 A_2 A_3, A_2, A_3, A_4, .. etc.

    In other words, in the outer loop, iterate over d, the length of
subsequences, that is d:= j - i = 0, 1, ..., n-2 - 0 = n-2:
        Inside, iterate over i = 0, ... until j:= i+d > n-2, then break and
go to the next d.
            Inside, iterate over k. Keep track of the current argmax and
max, and once done iterating over k, write these to
            m[i,j], s[i,j] respectively.
    '''
    n = len(p)

    nrows = p[:-1]
    ncols = p[1:]

    m = np.empty(shape=(n-1,n-1), dtype=np.int64)
    s = np.empty(shape=(n-1,n-1), dtype=np.int64)
```

```
    for i in range(0,n-1):
        m[i,i] = 0

    for d in range (1,n-1):
        for i in range(0,n-1):
            j = i+d
            if j > n-2:
                break
            min = m[i,i] + m[i+1, j] + nrows[i] * ncols[i] * ncols[j]
            argmin = i
            for k in range(i+1, j):
                if m[i, k] + m[k + 1, j] + nrows[i] * ncols[k] * ncols[j] <
min:
                    min = m[i, k] + m[k + 1, j] + nrows[i] * ncols[k] *
ncols[j]
                    argmin = k
            m[i,j] = min
            s[i,j] = argmin

    return m, s

def obtainParenthesization(s, i, j):
    '''
    Obtaining the minimum number of operations is simple: this is stored in
m[0,n-2].

    How do we obtain the corresponding parenthesizing?
    Well, if s[i,j] = k, then this is equal to:
    "( the optimal parenthesizing of A_i...A_k, which can be found
recursively using s[i,k])( the optimal parenthesizing of A_k+1...A_j, which
can be found recursively using s[k,j])

    The base case is encountered when k = i or k = j, in which case one of
the clauses is just A_i or A_j, respectively.

    In other words, we can find this parenthesizing recursively from array
s.

    What is the complexity of this lookup? Well, it is sort of a tree
search, but there are only n-2 A_i, and every time we recurse, one split is
created, and there
    are only n-3 places between A_i's where splits can be created, so the
search will branch n-3 times in total before it can only encounter leaves.
So obtaining
    the parenthization is O(n).
    '''
    if i == j:
        return f"A_{i}"
    else:
        return f"({obtainParenthesization(s,i, s[i,j])})
({obtainParenthesization(s,s[i,j]+1,j)})"

def getMatrixDims():
```

```
    return [int(s) for s in input("give the dimensions p0, .. , pn-1, where
Ai = pi x pi+1 for i = 0, .. n-2:\n").split()]

dims = getMatrixDims()
n = len(dims)
m, s = matMul(dims)
print(m)
print(f"Least number of operations needed: {m[0, n-2]}")
print("optimal parenthesization:")
print(obtainParenthesization(s, 0, n-2))
```

Ex. 3

Again, here is the Python implementation. The correctness and complexity is discussed in the comments. You can run the code and the example script. It includes an example problem for debugging, of which the DP-array will be printed to the screen.

```
import numpy as np

def coinPick(c, n):
    '''
    Assuming upper-left means: at position (0,0), and lower-right means: at
position (n-1,n-1)
    where grid will be an np.array of shape=(n,n).

    Define T as an (n,n) array where T[i,j] = maximum number of coins that
can be collected from square (i,j)
    Since we can only increment indices (i.e. move up or right), there is
an obvious base case:

    T[0,0] = c[i,j] since we can collect the coins on c[i,j] and then we
are finished

    if (i,j) == (i, n-1), we can only move rightward along the edge of the
grid, so T[i,n-1] = c[i,n-1] + t[i+1,n-1]
    if (i,j) == (n-1, j), we can only move upward, hence T[n-1,i] = c[i,j]
+ T[n-1,j+1]

    Otherwise, we can choose two directions: either up or right. Assuming
that we have memoized T[i',j']
    for {(i',j'): i'>=i and j'>j}, {(i',j'): i'>i and j'>=j}, we can
compute T[i,j] = c[i,j] + max {T[i+1,j],T[i,j+1]}

    The correct order of computation is in diagonal bands, i.e starting at

    (n-1,n-1)
    (n-2,n-1),(n-1,n-2)
    (n-3,n-1),(n-2,n-2),(n-1,n-3)

    ...
```

```
    (0,n-1), (1,n-2)...(n-2,1), (n-1,0)
    ...

    (0,0)

    In other words,

    for manh_dist = n-1 down to including 0:
        for  i,j in {(i,j):i+j = manh_dist}
            compute T[i,j] as above

    The induction argument above shows that T[0,0] indeed contains the
total maximum number
    of coins one can collect if one follows the optimal path.

    I have made the implementation use one big loop with case distinction.
    I think this is more clear (and with the same complexity of O(n^2))
than having separate loops
    for edge cases.

    Complexity: each square in the grid gets visited once. Inside the loop,
there are only constant
    time operations. So the complexity is O(n*n) = O(n^2)
    '''
    T = np.empty(shape=(n,n), dtype=np.int64)

    T[n-1,n-1] = c[n-1,n-1]

    for i in range(n-2,-1,-1):
        T[i,n-1] = c[i,n-1] + T[i+1,n-1]
        T[n-1,i] = c[n-1,i] + T[n-1,i+1]

    for d_from_end in range(1,n):
        for i in range(1,d_from_end+1):
            j = d_from_end - i
            if j <= 0:
                break
            T[i,j] = c[i,j] + max(T[i+1,j],T[i,j+1])

    for d_from_begin in range(n-2,-1,-1):
        for i in range(d_from_begin+1):
            j = d_from_begin - i
            T[i,j] = c[i,j] + max(T[i+1,j],T[i,j+1])

    return T

N = 5
grid = np.random.random_integers(low=0, high=20, size=(N,N))

print(f"Grid:\n{grid}")
dp_grid = coinPick(grid,N)
print(f"Dynamic Programming grid:\n{dp_grid}")
```

Ex. 4

The Python code below implements the algorithm. You can find the complexity analysis and proof of correctness in the comments. Finally, I included an additional algorithm for retrieving the parsing. And at the bottom, an example problem.

I use a `set()` for the "dictionary" (which has O(1) lookup indeed) and the data type `list()` for the string `s[0..n-1]`

```
import numpy as np # for the array, again

def matchDict(d, s):
    '''
    A dynamic programming algorithm that determines whether the string
    s[i]..s[j] can be reconstituted as a sequence of words in d.

    The recursion relation we can use is:
    s[i]..s[j] can be reconstituted if and only if one of the following
holds:

    * if i == j, s[i] must be in d, otherwise it cannot be reconstituted (a
single character cannot be broken up any further)
    * d(s[i]..s[j]), i.e. s[i]..s[j] is in d.
    * if s[i]..s[j] is not in d, it can only be reconstituted if it can be
split in two
      subwords s[i]..s[k], s[k+1]..s[j], where i <= k < j.

    So if we denote "s[i]..s[j] can be reconstituted from d" as R[i,j], the
above gives a recurrence for R[i,j],
    where just like in the case of matrix multiplications, we have
expressed R[i,j] in a base case or R[i,k], R[k+1,j] for
    k between i and j, i.e. for smaller substrings.

    We can solve this efficiently by computing R[i,j] for increasing sizes
of dist := j-i (= 0,1,..,n-1-0 ) and memoizing
    the intermediate values in an array R, of which we will only filla and
consider the upper triangular (R[i,j] for i<=j) values.

    The complexity of this algorithm is recognized easily in this
implementation, because it is very clear what
    this algorithm does: it fills the upper triangle of an array,
    which are 1/2 * n * (n-1) entries. The computation of each entry (i,j)
is an O(j-i) computation in the worst case.

    Although the summation sum_i sum_j (j-i) is still a bit tedious, we
know it depends on n in third order due to the indices
    appearing as terms in a double summation: this is all we need; the
algorithm is O(n^3).
    '''
    n = len(s)
    R = np.empty(shape=(n,n), dtype=bool)
```

```
    '''
    Although not specified in the assignment, I would also like to
reconstruct the way this algorithm splits the string.
    So in a separate array Sp, we keep track of which k is used in the
split at each stage. We set Sp[i,j] = -1
    if the word s[i]..s[j] was not split but found directly in the
dictionary. If the parsing of s[i]..s[j] failed, we
    can set Sp[i,j] = -2, for example, although a fail to split can also be
deduced from R[i,j]
    '''
    Sp = np.empty(shape=(n,n),dtype=np.int64)

    '''
    The base cases lie on the main diagonal: i==j
    '''
    for i in range(n):
        if s[i:i+1] in d:
            R[i,i] = True
            Sp[i,i] = -1
        else:
            R[i,i] = False
            Sp[i,i] = -2



    '''
    The inductive cases are solved for dist := j-i in 1,2,..,n-2
        Then iterate over i = 0, .. n-1 and set j = i + dist, and stop
        Once j (the column index) goes out of bounds, break and go to the
next diagonal band.
    '''



    for dist in range(1,n):
        for i in range(n-1):
            j = i+dist
            if j > n-1:
                break
            if s[i:j+1] in d:
                R[i,j] = True
                Sp[i,j] = -1
            else:
                R[i,j] = False
                Sp[i,j] = -2
                for k in range(i,j):
                    if R[i,k] and R[k+1,j]:
                        R[i,j] = True
                        Sp[i,j] = k

    return R[0,n-1], Sp

def reconstructParse(s, Sp, i,j):
```

```
    if Sp[i,j] == -1:
        return f"D({s[i:j+1]})"
    if Sp[i,j] == -2:
        return f"FAIL({s[i:j+1]})"
    else:
        return f"{reconstructParse(s, Sp, i,Sp[i,j])} |
{reconstructParse(s, Sp, Sp[i,j]+1,j)}"



# Two example strings and a dictionary d
d = set(["d", "gelei", "kaas", "ik","hebe","gesmeer", "met", "boterham",
"en"])
string1 = "ikhebeenboterhamgesmeerdmetgeleienkaas"
string2 = "ikhebeenboterhamgesmeerdmetgelienkaas"

# Run on example 1
FoundSplit1, Split1 = matchDict(d,string1)
print(FoundSplit1)
print(reconstructParse(string1, Split1, 0,len(string1)-1))

# Run on example 2
FoundSplit2, Split2 = matchDict(d,string2)
print(FoundSplit2)
print(reconstructParse(string2, Split2, 0,len(string2)-1))


# A small example I used for debugging (indices...)
d2 = set(["a", "aa"])
small = "aaa"
FoundSplitSmall, SplitSmall = matchDict(d2,small)
print(reconstructParse(small, SplitSmall, 0,len(small)-1))
```

** A final disclaimer: in exercise 2 and 4, only the upper triangular part of the printed arrays contain computed information. The strictly lower triangular part is uninitialized and will in general contain garbage values (I used `np.empty()`, which does not initialize entries).