Algorithms & Data Structures II (course 1DL231)
Uppsala University – Autumn 2023
Report for Assignment 1

Yueyuan Huang

23rd November 2023

Part 1
The Weightlifting Problem

A The Recursive Equation

Define $F(P, w, p)$ - in which P is a list of n weights of weightlifting plates at a gym, w is a preferred total weight for a weightlifter, and p is an integer - as whether there exists a list P' of elements among the first p elements of P whose sum is exactly w. In other words,

$$F(P, w, p) = \begin{cases} True & \text{if there exists such list,} \\ False & \text{otherwise.} \end{cases}$$

We can therefore write the recursive equation for $F(P, w, p)$ as (Assume we are using a 0-indexed list, like the one in Python):

$$F(P, w, p) = \begin{cases} w == 0 & \text{if } \ p \ is \ 0, \\ F(P, w, p-1) \ or \ F(P, w - P[p-1], p-1) & \text{otherwise.} \end{cases}$$

From the recursive equation, we can say the weightlifting problem has the optimal substructure property because both $F(P, w, p-1)$ and $F(P, w-P[p-1], p-1)$ are parts of $F(P, w, p)$. It also has overlapping subproblems, which happens when we have elements in P having identical value. For example, we have $P = [1, 2, 2]$ and $w = 3$. In this case, by starting with $F(P, 3, 3)$, we first have to compute $F(P, 3, 2)$ and $F(P, 1, 2)$, which will require us to deal with $F(P, 3, 1)$, $F(P, 1, 1)$, $F(P, 1, 1)$ and $F(P, -1, 1)$. We can see an overlapping occurs.

In fact we may do some pruning by adding some basic cases. Then the recursive equation will look like this:

$$F(P, w, p) = \begin{cases} False & \text{if } w < 0, \\ True & \text{if } w \ is \ 0, \\ w == 0 & \text{if } \ p \ is \ 0, \\ F(P, w, p-1) \ or \ F(P, w - P[p-1], p-1) & \text{otherwise.} \end{cases}$$

B The Recursive Algorithm

As shown in 1.

```python
# recursion invariant: p stays positive and will always decrease by 1 after
# each recursive call, and the recursion will terminate when p reaches 0
def weightlifting_recursive(P: List[int], w: int, p: int) -> bool:
    '''
    Pre: 0 <= p <= len(P)
    Post: p == 0
    Ex: P = [2, 32, 234, 35, 12332, 1, 7, 56]
        weightlifting_recursive(P, 299, 8) returns True
        weightlifting_recursive(P, 11, 8) returns False
    '''
    # 1. Add base case(s)
    if w < 0:
        return False
    if w == 0:
        return True
    if p == 0:
        return False
    # 2. add recursive case(s)
    return weightlifting_recursive(P, w, p-1) \
        or weightlifting_recursive(P, w - P[p-1], p-1)
```

Listing 1: Python function $weightlifting\_recursive(P, w, p)$.

C The Top-down DP Algorithm

As shown in 2.

```python
# recursion invariant (for _wltd()): p stays positive and will always decrease by
# 1 after each recursive call, and the recursion will terminate when p reaches 0
def weightlifting_top_down(P: List[int], w: int,
                          dp_matrix: List[List[Union[None, bool]]]) -> bool:
    '''
    Pre: dp_matrix is a matrix of size len(P)+1 x w+1
    Post:
    Ex: dp_matrix [[None, ..., None], ..., [None, ..., None]]]
        P = [2, 32, 234, 35, 12332, 1, 7, 56]
        weightlifting_top_down(P, 299, dp_matrix) returns True
        weightlifting_top_down(P, 11, dp_matrix) returns False
    '''
    # A recursive function that uses dp_matrix for memoisation
    def _wltd(P: List[int], w: int, p: int) -> bool:
        '''
        Pre: 0 <= p <= len(P)
        Post: p == 0
        '''
        if w < 0:
            return False
        if p == 0:
            dp_matrix[p][w] = False
        if w == 0:
            dp_matrix[p][w] = True
        if dp_matrix[p][w] is None:
            dp_matrix[p][w] = _wltd(P, w, p-1) or _wltd(P, w - P[p-1], p-1)
        return dp_matrix[p][w]
    # Use _wltd() to do a top-down dp
    return _wltd(P, w, len(P))
```

Listing 2: Python function $weightlifting\_top\_down(P, w, dp\_matrix)$.

D Time Complexity Comparison

Now we can look at algorithm 1 and 2 and compare their time complexities.

It is obvious that these 2 algorithms have the same best-case complexity (achieved when $P[-1] = w$ or $w = 0$), which is $O(1)$.

The worst-case complexity is got when there doesn't exist such a list P'. For example, P is a list of n 1's and $w = n + 1$. In this case, the recursive algorithm 1 would have to do 2^n function calls, while the top-down algorithm 2 could avoid duplicate calculations and would just do $n * n$ function calls. Therefore the top-down algorithm has the best worst-case time complexity, which is $O(n * w)$, compared with the recursive one's worst-case time complexity $O(2^n)$.

E The Bottom-up DP Algorithm

As shown in 3.

```
def weightlifting_bottom_up(P: List[int], w: int,
                            dp_matrix: List[List[Union[None, bool]]]) -> bool:
    '''
    Pre: dp_matrix is a matrix of size len(P)+1 x w+1
    Post: dp_matrix has signature List[List[bool]]
    Ex: dp_matrix [[None, ..., None], ..., [None, ..., None]]
         P = [2, 32, 234, 35, 12332, 1, 7, 56]
         weightlifting_bottom_up(P, 299, dp_matrix) returns True
         weightlifting_bottom_up(P, 11, dp_matrix) returns False
    '''
    # 1. Fill first column and row of dp_matrix
    for i in range(len(P)+1):
       dp_matrix[i][0] = True
    for j in range(1, w+1):
       dp_matrix[0][j] = False
    # 2. iteratively fill rest of dp_matrix
    for i in range(len(P)):
       for j in range(w, -1, -1):
          dp_matrix[i+1][j] = dp_matrix[i][j]
          if dp_matrix[i][j] and j + P[i] <= w:
             dp_matrix[i+1][j+P[i]] = dp_matrix[i][j]
    # 3. return the result from the dp_matrix
    return dp_matrix[len(P)][w]
```

Listing 3: Python function $weightlifting\_bottom\_up(P, w, dp\_matrix)$.

F Algorithm to find P'

The algorithm shown in 4 is a modification of the one 3 in section E which will instead return a list P' (as described in section A) if one exists.

G Algorithm to return $\hat{w}$

We can modify the weightlifting problem such that the greatest total weight $\hat{w}$ of weightlifting plates that does not exceed the preferred total weight of the weightlifter, $\hat{w} \le w$, is to be returned.

```python
def weightlifting_list(P: List[int], w: int,
                       dp_matrix: List[List[Union[None, bool]]]) -> List[int]:
    '''
    Pre: dp_matrix is a matrix of size len(P)+1 x w+1
    Post: dp_matrix has signature List[List[bool]]
    Ex: P = [2, 32, 234, 35, 12332, 1, 7, 56]
         weightlifting_list(P, 299) returns a permutation of [2, 7, 56, 234]
         weightlifting_list(P, 11) returns []
    '''
    # A matrix storing the choice of plates for each subproblem
    choice_matrix = [[[] for _ in range(w+1)] for _ in range(len(P)+1)]
    # Fill first column and row of dp_matrix
    for i in range(len(P)+1):
       dp_matrix[i][0] = True
    for j in range(1, w+1):
       dp_matrix[0][j] = False
    # Iteratively fill rest of dp_matrix and choice_matrix
    for i in range(len(P)):
       for j in range(w, -1, -1):
          dp_matrix[i+1][j] = dp_matrix[i][j]
          choice_matrix[i+1][j] = choice_matrix[i][j]
          if dp_matrix[i][j] and j + P[i] <= w:
             dp_matrix[i+1][j+P[i]] = dp_matrix[i][j]
             choice_matrix[i+1][j+P[i]] = choice_matrix[i][j] + [P[i]]
    # Return the result from the choice_matrix
    return choice_matrix[len(P)][w] if dp_matrix[len(P)][w] else []
```

Listing 4: Python function $weightlifting\_list(P, w, dp\_matrix)$.

Given the algorithm 3, before returning in line 119, we can traverse the $len(P)$-th row of dp_matrix to find the maximum j which satisfies $dp\_matrix[len(P)][j] = True$. That j will be the $\hat{w}$ we are looking for.

Part 2
Augmenting Path Detection in Network Graphs

A Finding the Existence of Augmenting Path

The algorithm 5 returns $True$ if and only if there exists an augmenting path from the source s to the sink t in a flow network G.

B Finding the Augmenting Path

The algorithm 6 returns also an augmenting path if one exists.

C Time Complexity Analysis

The extended algorithm 6 performs a BFS which discovers all nodes reachable from the source s. The BFS takes at most $O(|E|)$ time because each edge is travelled at most once, and when

```python
def augmenting(G: Graph, s: str, t: str) -> bool:
   '''
   Pre: G is a flow network with source s and sink t
       (so s != t and s is in G and t is in G)
   Post: None of the parameters are modified
   Ex: >>> G = Graph(is_directed=True)
      >>> G.add_edge('a', 'b', capacity=1, flow=0)
      >>> augmenting(G, 'a', 'b')
      True
      >>> G = Graph(is_directed=True)
      >>> G.add_edge('a', 'b', capacity=1, flow=2)
      >>> augmenting(G, 'a', 'b')
      False
   '''
   q = deque()
   visited = [False] * 256 # Assume that nodes of G are ASCII characters
   q.append(s)
   visited[ord(s)] = True
   while q:
      u = q.popleft()
      for v in G.neighbors(u):
         if visited[ord(v)]: continue
         if G.flow(u, v) < G.capacity(u, v):
            if v == t: return True
            q.append(v)
            visited[ord(v)] = True
   return False
```

Listing 5: Python function $augmenting(G, s, t)$.

examining if we have visited a node (thus determining whether to add the edge from current node to it), we simply look it up in our *visited* array, which takes $O(1)$ time. If the sink t is reached, we get the augmented path by going backwards it with the help of the hash table *prev* we maintained, which also takes at most $O(|E|)$ time, because only the edges along the augmented path is visited. We can conclude that the overall time complexity of the extended algorithm is $O(|E|)$.

```python
def augmenting_extended(G: Graph,
                  s: str, t: str) -> Tuple[bool, List[Tuple[str, str]]]:
   '''
   Pre: G is a flow network with source s and sink t
       (so s != t and s is in G and t is in G)
   Post: None of the parameters are modified
   Ex: >>> G = Graph(is_directed=True)
      >>> G.add_edge('a', 'b', capacity=1, flow=0)
      >>> G.add_edge('b', 'c', capacity=1, flow=0)
      >>> augmenting_extended(G, 'a', 'c')
      True, [('a', 'b'), ('b', 'c')]
      >>> G = Graph(is_directed=True)
      >>> G.add_edge('a', 'b', capacity=1, flow=2)
      >>> augmenting_extended(G, 'a', 'b')
      False, []
   '''
   prev = {}
   q = deque()
   visited = [False] * 256 # Assume that nodes of G are ASCII characters
   q.append(s)
   visited[ord(s)] = True
   while q:
      u = q.popleft()
      for v in G.neighbors(u):
         if visited[ord(v)]: continue
         if G.flow(u, v) < G.capacity(u, v):
            prev[v] = u
            if v == t:
               path = []
               while v != s:
                  path.append((prev[v], v))
                  v = prev[v]
               path.reverse()
               return True, path
            q.append(v)
            visited[ord(v)] = True
   return False, []
```

Listing 6: Python function *augmenting_extended*(G, s, t).