Grading Policy for #30

Total points: 7.5

1) Give a worst-case running time of this algorithm (3.5pts)

- **Answer**: Θ(n^2), Theta(n^2), O(n^2)
- **Description**: If Partition generates `1:(n-1)` splits for every iteration, the time complexity would be `T(n) = T(0) + T(n-1) + Theta(n)`. Thus, `T(n)θ(n^2)`.
- Correct answer: 1.5pts
- Reasonable description: 2pts

2) Give a wost-case example (4pts)

- **Answer**: `6,1,2,3,4,5` or any other examples that satisfies the given statement
- **Description**: At every partition, the algorithm will choose pivots that have lowest values with the given list. More specifically, the algorithm will select 1,2,3,4,5 as pivots in given order, which generates `1:(n-1)` partitions for every `n` iterations.
- Correct example: 2pts
- Reversed example: 1pts
    - For example, `1,6,5,4,3,2` is not a worst-case example on the given psuedo-code. But if we try to sort with descending order, it yields 15 iterations, which is a tight-worst case.
- Reasonable description: 2pts
    - If a student provide a **wrong example** with reasonable description, **0** credits for the description are given.
    - If a student provide a **reversed example** with reasonable description, **2** credits for the description are given.
- Code for validating an example

```
def quicksort_iterations(A):
    def quicksort(A, p, r):
        num_iter = 0
        if p < r:
            q, num_iter = partition(A, p, r)
            num_iter += quicksort(A, p, q-1)
            num_iter += quicksort(A, q+1, r)
        return num_iter

    def partition(A, p, r):
        s = min(p+1, r)
        A[r], A[s] = A[s], A[r]
```

```
        num_iter = 0
        x = A[r]
        i = p - 1
        for j in range(p, r):
            num_iter += 1
            if A[j] <= x:
                i += 1
                A[i], A[j] = A[j], A[i]

        A[i+1], A[r] = A[r], A[i+1]
        return i+1, num_iter

    num_iter = quicksort(A[:], 0, len(A)-1)
    return num_iter

def quicksort_iterations_desc(A):
    def quicksort(A, p, r):
        num_iter = 0
        if p < r:
            q, num_iter = partition(A, p, r)
            num_iter += quicksort(A, p, q-1)
            num_iter += quicksort(A, q+1, r)
        return num_iter

    def partition(A, p, r):
        s = min(p+1, r)
        A[r], A[s] = A[s], A[r]

        num_iter = 0
        x = A[r]
        i = p - 1
        for j in range(p, r):
            num_iter += 1
            if A[j] >= x:
                i += 1
                A[i], A[j] = A[j], A[i]

        A[i+1], A[r] = A[r], A[i+1]
        return i+1, num_iter

    num_iter = quicksort(A[:], 0, len(A)-1)
    return num_iter

def is_answer(example):
    N = len(example)
    optimal = N * (N - 1) // 2
    given1 = quicksort_iterations(example)
    given2 = quicksort_iterations_desc(example)

    if optimal == given1:
        return 'Full credit'
    elif optimal == given2:
        return 'Partial credit'
    else:
```

```
        return 'No credit (%d/%d)' % (given1, optimal)

# Usage
print(is_answer([6,1,2,3,4,5])) # Full credit
print(is_answer([1,6,5,4,3,2])) # Partial credit
print(is_answer([1,2,3,4,5,6])) # No credit
```

#31

You are given n boxes $b_1, \ldots, b_n$ of the same size and same cuboid-shape, and they can be stacked perpendicular to the floor, but each box b_i has its own weight $w_i > 0$ and strength $s_i > 0$ so that if the total weight of boxes stacked on the box b_i exceeds its strength s_i, then the box b_i immediately breaks. Your task is to find the maximal number of boxes that can be stacked perpendicular to the floor, without breaking any boxes in the stack. Assuming that the boxes are listed in increasing order of their strengths (i.e., $s_1 \leq \ldots \leq s_n$), answer the following questions.

(1) Let $E(i, j)$ be the minimal total weight of exactly j boxes stacked using $b_1, \ldots, b_i$. Then it is guaranteed to find $E(i, j)$ using solutions of its subproblems, $E(i', j')$ ($i' < i$ or $j' < j$). To find a minimal total weight of boxes stacked using $b_1, \ldots, b_n$, state the recurrence relation of $E(i, j)$ when $i \geq 1$ and $j \geq 1$ and justify your answer with short descriptions. Make sure that it must be well-defined for all $i \geq 1$ and $j \geq 1$ (You can use $E(i, j) = \infty$ if the stacking is impossible).

(2) Using recurrence relation of (1), give a method to find the maximal number of boxes stacked using $b_1, \ldots, b_n$, and justify your answer with short descriptions.

Common mistakes:
The main difference between 0-1 knapsack problem and this one is that you need to consider both weights w_i and strengths s_i of each element b_i. If you do not consider s_i when defining recurrence relation of $E(i, j)$, each one is generally the value derived from the impossible stack (including *broken* boxes). Also if you try to insert any box into the *middle* of the previous stack, you need to consider all the boxes below not to break them.

Answer of (1). The recurrence relation is the following:

$$\begin{aligned} E(i,0) &= 0 \quad && \text{if } i \geq 0 \\ E(0,j) &= \infty && \text{if } j > 0 \end{aligned}$$

for $i < 1$ or $j < 1$, and

$$E(i,j) = \begin{cases} \min\{E(i-1,j-1) + w_i, E(i-1,j)\} & \text{if } E(i-1,j-1) \leq s_i \\ E(i-1,j) & \text{Otherwise} \end{cases}$$

for $i \geq 1$ and $j \geq 1$.

Description: Any stack of j boxes using $b_1, \ldots, b_i$ can be divided into two cases: having b_i or not. If it has b_i, we can make the stack by making a sub-stack of j boxes using $b_1, ..., b_{i-1}$ and adding b_i into it (assuming we add b_i into the bottom of the sub-stack not to break any boxes in it). If it hasn't b_i, then we can make a sub-stack using j boxes among $b_1, ..., b_{i-1}$. The minimal weight of the two sub-stacks can be represented by $E(i-1, j-1) + w_i$ and $E(i-1, j)$, respectively, and therefore the minimal weight of the original stack $E(i,j)$ is minimum of the two. However, we should not consider the first value if b_i cannot be added to the bottom of the sub-stack (in the case of $E(i-1,j-1) > s_i$) so in this case, $E(i,j) = E(i-1,j)$. For simplicity, we define $E(i,0) = 0$ for $i \geq 0$ and $E(0,j) = \infty$ for $j > 0$ as our base cases, each of which means 0 weight if we *don't need* to make a stack and the largest weight if we *cannot* make a stack.

Grading criteria [5pt]: Recurrence relation of $E(i,j)$ (without base cases) [3pt] + Base cases well-defined for all $i, j \geq 1$ [1pt] + Justification [1pt]. No partial grade for each criterion.

Answer of (2). The maximal number of boxes stacked using $b_1, \ldots, b_n$ is as below:

$$\operatorname*{arg\,max}_{0 \leq j \leq n} \{E(n,j) | E(n,j) \neq \infty\}$$

Description: By the recurrence relation in (1), $E(n,j) = \infty$ when we cannot make a stack of j boxes using $b_1, \ldots, b_n$. Except the case, we can find the largest j satisfying $0 < E(n,j) < \infty$ and it means that we can also find a stack of j boxes. Suppose there exists another stack of $j' > j$ boxes. Then by definition of E, we have $0 < E(n,j') < \infty$ and it makes contradiction to the largest value j.

Grading criteria [2.5pt]: Method [2pt] + Justification [0.5]. No partial grade for each criterion.

problem #3-SAT is a decision problem such that when a 3CNF formula ϕ and a natural number k is given, it is required to decide if ϕ admits k different satisfying assignments. Prove that #3-SAT is in ***NP-hard***.

In order to check if #3-SAT is in ***NP***, consider the following candidate for a certificate: if ϕ admits k different assignments, let the k different assignments be a certificate for ϕ. Does this approach work? Explain why it works or fails.

solution Let us consider an input ϕ for 3-SAT. See that ϕ is a YES instance of 3-SAT if and only if ϕ admits a satisfying assignment. Hence, we can run #3-SAT on $(\phi, 1)$ and can expect that it returns YES if and only if ϕ is a YES instance of 3-SAT. Therefore, #3-SAT is in ***NP-hard***.

However, the candidate for certificates fails. Let $\phi \equiv (x_1 \vee x_2 \vee x_3) \wedge \cdots \wedge (x_{n-2} \vee x_{n-1} \vee x_n)$ where x_is are distinct variables. Then, ϕ admits 2^n variables. The size of the input $(\phi, 2^n)$ is $\Theta(n)$. (Since, the size of the natural number 2^n is $\log 2^n = n$.) Therefore, checking all the 2^n assignments takes exponential run-time to the size of the input. Thus, with the candidate, we cannot conclude that 3-SAT is in ***NP-complete***.

Note: those who read ϕ *admits* k *different satisfying assignments* as ϕ *admits exactly* k *different assignments* and who regarded the length of a natural number k being k not $\log k$ will still get their solution graded properly. For a correct solution, the first part of proving the problem being in ***NP-hard*** deserves four points and the latter part of checking the problem being in ***NP*** deserves three points.