Problem 1 Suppose you are given two sets A and B, each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the i^{th} element of set A, and let b_i be the i^{th} element of set B. You then receive a payoff of $\prod_{i=1}^{n} a_i^{b_i}$. Give an efficient algorithm to find the reorderings that will maximize your payoff. Present the pseudocode. Prove that your algorithm maximizes the payoff, and state its running time.

Solution

Proof of Algorithm:-

Here Payoff is $\prod_{i=1}^{n} a_i^{b_i}$. which we get after reordering of set A and set B however we like. The problem here needs maximization of payoff or say reordering set A and set B in such a way that payoff value is highest between all combinations.

This maximization of payoff can be solved with simpler mathematics. Let A = {1, 2} and B = {4, 5} different combination of set A and set B will be as following:

1. A= {1, 2}
   B={4, 5}
   output = 1^4.2^5 = 2^5 = 32
2. A= {1, 2}
   B={5, 4}
   output = 1^5.2^4 = 16
3. A= {2, 1}
   B={4, 5}
   output = 2^4.1^5 = 16
4. A= {2, 1}
   B={5, 4}
   output = 2^5.1^4 = 2^7 = 32

By the Above observation, we can get the idea that if set A and B are both ordered in the same order i.e. A & B both sorted in increasing or decreasing order then the payoff will be maximum.

Now using this we can solve it for n > 2 where n is number of positive integers in A and B. Let's say ai and bi are the ith element after sorting A and B in decreasing order. Now

a1 > a2 > a3 > a4 . . . . > an
b1 > b2 > b3 > b4 . . . . > bn

So value of a_i^{bi} will be as following:

$a1^{b1} > a2^{b2} > . . . . > an^{bn}$

Hereby above observation we know a_1 is the maximal element in A and to attend maximal element such that $a1^{bi}$ where bi can be any element from B. then $a1^{b1}$ will give maximum value we can get for pay off and this will continue for next maximum value we can get which is lesser than first maximum value $a1^{b1}$.

Hence Sorting of A and B in the same order then calculating payoff will give the maximized value of payoff which we can attain in O(n log n) Time complexity as described below.

"""

1. Sorting in increasing order of each set's will take time of O(n log n) time complexity

2. For loop will take O(n) time as running for n elements

3. Rest of the code will be O(c1) time where c1 is constant

So Total time complexity will be O(n log n) + O(n log n) + O(n) + O(c1) = O(n log n)

```
"""
def maximum_payoff(first: [], second: []):
    if first is None or second is None or len(first) != len(second):
        return Exception("Error in the input please provide correct input :) !")

    first.sort()
    second.sort()

    payoff = 1
    for index in range(0, len(first)):
        payoff = payoff * (first[index] ** second[index])
    return payoff
```

Problem 2 What is the best way to multiply a chain of matrices with dimensions that are 9×6, 6×3, 3×21, 21×11, 11×5, and 5×50? Show your work.
Solution

The best possible solution to multiply a chain of matrices with the above dimension are as follows:-

All matrix multiplication are calculated as below(here multiplication cost and way of multiplication both are added):-

	1	2	3	4	5	6
1	0	162 (AB)	729 ((AB) C)	1152 (AB)(CD)	1155 (AB)((CD)E)	3120 ((AB)(((CD)E)F))
2		0	378 (BC)	891 (B)(CD)	948 (B)((CD)E)	2448 (((B)((CD)E))F)
3			0	693 (CD)	858 ((CD)E)	1608 (((CD)E)F)
4				0	1155 (DE)	6405 ((DE)F)
5					0	2750 (EF)
6						0

Let A B C D E F be respective matrices where dimensions are as following:-
A = 9 * 6
B = 6 * 3
C = 3 * 21
D = 21 * 11
E = 11 * 5
F = 5 * 50

Then optimal solution of matrix multiplication get 3120 multiplication cost where multiplication is done as follows:- ((A B) (((C D) E) F))

Problem 3 Give a pseudopolynomial algorithm for KNAPSACK. Strive for running time of $O(nB)$, but make sure running time is polynomial in n and B. The KNAPSACK problem is defined as follows. An instance consists of n items $1, 2, \ldots, n$ where item i has size s_i and profit p_i, and a knapsack size B with $B \geq s_i$ for all $i = 1, 2, \ldots, n$. All the numbers are integers. A feasible solution consists of a subset Q of $\{1, 2, \ldots, n\}$ such that $\sum_{i \in Q} s_i \leq B$. The objective is to maximize the total profit of Q - that is $\sum_{i \in Q} p_i$.

Present the pseudocode, discuss correctness, and analyze the running time.

Solution

knapsack size is B and the maximum element we can get into knapsack is such that summation of those element sizes is less than knapsack size.

One way to solve this problem is to consider every subset of n items and then computing the total profit condition that subset size < B. For n numbers, we can get 2^n subsets so by doing that we will get the time complexity of $O(2^n)$.

Via Dynamic Programming, we can solve this problem in Pseudo Polynomial-time. Here while calculating every 2^n subsets we will get recurring subproblems, which we can improvise by storing their solution in memory and then again using it for reducing time complexity by taking use of stored memory.

```
"""
   Here For Every combination of elements_remained and remaining_bag_size
we are initializing data where
   n = elements_remained
   B = bag_size
"""


def knapsack_max_profit_using_dynamic_programing(elememnts_size: [],
                                                 elements_profit: [],
                                                 bag_size: int):
    memorization = []
    for i in range(0, len(elememnts_size) + 1):
        memorization.append([-1] * (bag_size + 1))

    return knapsack_max_profit(elememnts_size, elements_profit,
bag_size, len(elememnts_size), memorization)



"""
   Time Complexity Analysis:-
       As we are memorizing every combination of elements_remained and
remaining_bag_size we don't have to see sub problem of that combination again
and again, which will lead us run time where we are calling
```

```
knapsack_max_profit for O(n * B) time to fill the every combination of
elements_remained and remaining_bag_size.
       So time complexity will be O(n * B)
"""
```

```
def knapsack_max_profit(elements_size: [], elements_profit: [],
                        remaining_bag_size: int, elements_remained: int,
                        memorization: []):
    max_profit = 0
    if elements_remained == 0 or remaining_bag_size == 0:
        return max_profit
    if memorization[elements_remained][remaining_bag_size] != -1:
        return memorization[elements_remained][remaining_bag_size]
   # Here because element size is less than remaining bag_size we can include
that in bag
    if remaining_bag_size >= elements_size[elements_remained - 1]:
        # Here we are not including the weight of current element and
going to element before it.
        profit_without_nth_element = knapsack_max_profit(elements_size,
elements_profit, remaining_bag_size, elements_remained - 1, memorization)
        # Here we are including the weight of the current element so
profit of item will also get included in out profit.
        profit_with_nth_element = elements_profit[elements_remained -
             1] + knapsack_max_profit(elements_size, elements_profit,
           remaining_bag_size - elements_size[elements_remained - 1],
             elements_remained - 1, memorization)
        max_profit = max(profit_with_nth_element,
                                        profit_without_nth_element)
    # Here current element size is greater than remaining bag size so
we don't have any option and we have to avoid it
    else:
        max_profit = knapsack_max_profit(elements_size,
                                elements_profit, remaining_bag_size,
                                            elements_remained - 1,
                                                  memorization)
    memorization[elements_remained][remaining_bag_size] = max_profit
    return memorization[elements_remained][remaining_bag_size]
```

Correctness:-

Above code is considering every possibility of including element into knapsack as element can be considered only fully there is only two possibility either to include the element and reduce bag size or not include the element and bag size remains the same. So via that we are comparing best possible case and keeping max profit from every case recursively. It will in end give us the correct output.

Problem 4 Problem 15-4 from the textbook (”Printing neatly”). It is Problem 15-2 from the second edition of Cormen. Present the pseudocode, discuss correctness, and analyze the running time. Polynomial time is required.

15-4 Printing neatly

Consider the problem of neatly printing a paragraph with a monospaced font (all characters having the same width) on a printer. The input text is a sequence of n words of lengths $l_1, l_2, \ldots, l_n$, measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of M characters each. Our criterion of "neatness" is as follows. If a given line contains words i through j, where $i \leq j$, and we leave exactly one space between words, the number of extra space characters at the end of the line is $M - j + i - \sum_{k=i}^{j} l_k$, which must be nonnegative so that the words fit on the line. We wish to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of lines. Give a dynamic-programming algorithm to print a paragraph of n words neatly on a printer. Analyze the running time and space requirements of your algorithm.

Solution

In this solution we want to reduce the number of extra spaces our lines will take. So to minimize the sum of cubes of numbers of extra spaces characters at the end of lines. To do that we need to first get number of spaces will needed for adding every word in the line and after that using those we can use them to get different combinations of lines which will help to decide the minimum cost we can get by comparing different combination of words. To try every combination of words in every line it will take us O(n^2) of time.

```
"""
   Here we are using 4 for loops which have for loops inside them and every
for loops runs
   for n = number_of_words present to print neatly. Plus we have two for
loops to initialize
   values of minimum cost and printing positions. Other then this we have
some assignments
   which will cost constant time.

   So total time complexity =
   O(4 * n ^2) + (2 * n) + constant time which will be equals to O(n ^ 2)
time

   Space complexity
   Here we are using cost list of list which have n columns and n rows and
   minimum cost list and printing position list which have n columns so total
space we require is
   of O(n ^ 2) + some constant space time we require here which result in O(n
^ 2) space.
"""


def print_neatly(words: [], maximum_character: int):
   number_of_words = len(words)
```

```
    cost = []
    for i in range(number_of_words):
        cost_row = []
        for j in range(number_of_words):
            cost_row.append(0)
        cost.append(cost_row)

    for start_word_index in range(0, number_of_words):
        cost[start_word_index][start_word_index] = maximum_character -
len(words[start_word_index])
        for end_word_index in range(start_word_index + 1, number_of_words):
            cost[start_word_index][end_word_index] =
cost[start_word_index][end_word_index - 1] - (
                    len(words[end_word_index]) + 1)

    for start_word_index in range(0, number_of_words):
        for end_word_index in range(start_word_index, number_of_words):
            if cost[start_word_index][end_word_index] >= 0:
                cost[start_word_index][end_word_index] =
cost[start_word_index][end_word_index] ** 3
            else:
                cost[start_word_index][end_word_index] = sys.maxsize

    minimum_cost = []
    for i in range(0, number_of_words):
        minimum_cost.append(0)

    print_positions = []
    for i in range(0, number_of_words):
        print_positions.append(0)

    for start_word_index in range(number_of_words - 1, -1, -1):
        print_positions[start_word_index] = number_of_words
        minimum_cost[start_word_index] =
cost[start_word_index][number_of_words - 1]
        for end_word_index in range(number_of_words - 1, start_word_index,
-1):
            if cost[start_word_index][end_word_index - 1] == sys.maxsize:
                continue
            elif minimum_cost[start_word_index] > minimum_cost[end_word_index]
+ cost[start_word_index][
                end_word_index - 1]:
                print_positions[start_word_index] = end_word_index
                minimum_cost[start_word_index] = minimum_cost[end_word_index]
+ cost[start_word_index][
                    end_word_index - 1]

    i = 0
    j = print_positions[0]
    while True:
        printer = ''
        for k in range(i, j):
```

```
            printer += words[k] + " "
        print(printer)
        i = j
        if i >= number_of_words:
            break
        j = print_positions[i]
```

Correctness:-

In above algorithm we are checking length of every combination of words and then combining then to retrieve cost of line which will result in optimal solution by comparing each other. Also we are storing compared values in memory lists which will result in lesser time complexity.

eg. if we have different words of following length

```
[4, 4, 5, 2, 3, 3, 8]
```

```
then above code will combine words from i to j such that j >= i
and will count spaces at the last position which will be character
limit - sum of words lengths - number of words. by taking i and j
combinations i and j both represent different words we will break
i j combination into different lines which provide minimum cube
cost.

eg.
character limit = 10

Representing spaces
word  1     2     3     4     5     6     7

1     5     0     sys.maxsize

2           5     sys.maxsize

3                 4     1     sys.maxsize

4                       7     3     0     sys.maxsize

5                             and so on

.
.
```

which tells us where we can break the words and which combination can provide minimum cube cost.