COMP SCI 577 Homework 03 Problem 3

Dynamic Programming

Ruixuan Tu

rtu7@wisc.edu

University of Wisconsin-Madison

4 October 2022

Algorithm

Explanation

The entry point of this program is the subroutine `main()` to read the input (number of books N, all books $[B_i] = [(W_i, H_i)]$ (for width and height) with $i \in [0, N)$, and maximum of shelf width $W_{\lim}$), the main subroutine is $minHeightShelves(books, shelfWidth)$, and we want to solve the problem recursively using the subroutine $solve(firstBook)$. We assume the subroutine $solve(firstBook)$ for sub-problems returns the minimal height for books $[B_i]$ with $i \in [j, N)$ as the books are on some previous shelves that we do not care. Thus, the solution for all the books is $solve(0)$ as B_0 is the first of all books.

For the $solve(firstBook)$ subroutine, we define the base cases $solve(n) = 0$ (as there is no B_n be counted in height) and $solve(n-1) = H_{n-1}$ (as there is only one shelf $[B_{n-1}]$ for this situation, and its height is H_{n-1}). If some other situations are already solved, we just yield the solution instead of solving again. Then for any situation $solve(j)$ which is not yet solved, we convert it into several sub-problems which are already solved (i.e., strong induction for all m such that $j < m \le n$). For $solve(j)$, we try to build a new shelf starting with B_j whose width does not exceed $W_{\lim}$. If we attempt all possible, consecutive B_m's as long as the width constraint is not violated, then for any B_m we have attempted, the next shelf must be starting with $B_m + 1$, so that the solution for B_j is the sum of current height and the total height in the sub-problem, i.e., $solve(j) = \min_{0 \le i \le M}(solve(j+i+1) + \max_{0 \le k \le i}(H_{j+k}))$ with M be the maximum number of books on any shelf starting with B_j.

Code (Python)

```python
from typing import List
import numpy as np

class Solution:
    solutions: np.ndarray = None  # solution[j] := height for books [j, N) on previous shelves
    n: int = None
    shelfWidth: int = None
    books: List[List[int]] = None  # books [0, N)

    def solve(self, firstBook: int):
        if firstBook == self.n:
            return 0
        if self.solutions[firstBook] != np.inf:
            return self.solutions[firstBook]
        candidates = list()
        shelfW = 0
        shelfH = np.zeros(self.shelfWidth)
        shelfMaxBooks = 0
        for i in range(self.shelfWidth):
            if firstBook + i >= self.n:
                break
            shelfW += self.books[firstBook + i][0]
            if shelfW > self.shelfWidth:
                break
            if i == 0:
                shelfH[i] = self.books[firstBook + i][1]
            else:
                shelfH[i] = max(shelfH[i - 1], self.books[firstBook + i][1])
            shelfMaxBooks += 1
        for i in range(shelfMaxBooks):
            candidates.append(self.solve(firstBook + i + 1) + shelfH[i])
```



```python
        self.solutions[firstBook] = min(candidates)
        return self.solutions[firstBook]

    def minHeightShelves(self, books: List[List[int]], shelfWidth: int) -> int:
        self.n = len(books)
        self.solutions = np.ones(self.n) * np.inf
        self.books = books
        self.solutions[self.n - 1] = self.books[self.n - 1][1]
        self.shelfWidth = shelfWidth
        return(int(self.solve(0)))

    def main(self):
        n, shelfWidth = tuple(int(x) for x in input().split(" "))
        w = list(int(x) for x in input().split(" "))
        h = list(int(x) for x in input().split(" "))
        books = list()
        for i in range(n):
            books.append([w[i], h[i]])
        print(self.minHeightShelves(books, shelfWidth))

if __name__ == "__main__":
    S = Solution()
    S.main()
```

Correctness

Induction

Claim: The algorithm is correct for all sub-problems starting with book B_n with $n \in \mathbb{N}$ and $n \le N$.
Base Case: $n = N$ and $n = N - 1$ as described in Explanation.
Inductive Step: Assume that the algorithm is correct for all m such that $n < m \le N$ by strong induction. For $n = j$, we have the calculation correct, as we have attempted all combinations of possible consecutive book sequences starting from B_j, based on the correct results from the

induction hypothesis, which is stated in Explanation. The **min** and **max** equation for $solve(j)$ holds, as all combinations are enclosed by the outmost **min** with the **max** just for counting the maximum height as the shelf height for any book sequence, and we optimize the **max** to be parallel to **min** to be $O(n)$ instead of $O(n^2)$, as the function **max** is monotonic that we can just compare the last one with the current one.

Termination

The algorithm must terminate, as the attempt on one shelf stops when either the $W_{\lim}$ constraint or the $i + j \ge N$ constraint is reached so that $i + j$ is always in the range $[i, N]$ without violating the base case. The latter constraint does not affect the correctness, as there is no book to count for height on and after B_N.

Complexity

For one shelf starting with j, the worst case is $n - j = O(n)$ attempts to construct the shelf, ignoring the width constraint, as we retrieve, calculate, and select the values from sub-problems with the retrieval complexity $O(1)$ for one sub-problem as they are memorized by calculation in other shelves with different starting books. Hence, there are $n = O(n)$ sub-problems, as $j \in [0, N)$ for the starting book. Therefore, the time complexity of this algorithm is $n * O(n) = O(n^2)$.

We have one list $solutions$ of length N and two temporary lists $shelfH$ and $candidates$ of length no greater than $N - j = O(n)$. As $N + 2 * O(n) = O(n)$, the space complexity of this algorithm is $O(n)$.

Appendix

Code (Python) of Slower but Correct Version (Draft)

Because we do not care how many shelves we have used, so the first dimension `shelf` should be removed to reduce the complexity, and the second dimension `firstBook` be kept for the final solution.

```python
from typing import List
import numpy as np

```

```python
class Solution:
    solutions: np.ndarray = None
    n: int = None
    shelfWidth: int = None
    books: List[List[int]] = None

    def solve(self, shelf: int, firstBook: int):
        if firstBook == self.n:
            return 0
        if shelf >= self.n or firstBook >= self.n:
            return np.inf
        if self.solutions[shelf][firstBook] != np.inf:
            return self.solutions[shelf][firstBook]
        candidates = list()
        shelfW = 0
        shelfH = np.zeros(self.shelfWidth)
        shelfMaxBooks = 0
        for i in range(self.shelfWidth):
            if firstBook + i >= self.n:
                break
            shelfW += self.books[firstBook + i][0]
            if shelfW > self.shelfWidth:
                break
            if i == 0:
                shelfH[i] = self.books[firstBook + i][1]
            else:
                shelfH[i] = max(shelfH[i - 1], self.books[firstBook + i][1])
            shelfMaxBooks += 1
        for i in range(shelfMaxBooks):
            candidates.append(self.solve(shelf + 1, firstBook + i + 1) + shelfH[i])
        self.solutions[shelf][firstBook] = min(candidates)
        return self.solutions[shelf][firstBook]

```



```python
    def minHeightShelves(self, books: List[List[int]], shelfWidth: int) -> int:
        self.n = len(books)
        self.solutions = np.ones((self.n, self.n)) * np.inf
        self.books = books
        self.solutions[self.n - 1][self.n - 1] = self.books[self.n - 1][1]
        self.shelfWidth = shelfWidth
        return(int(self.solve(0, 0)))

    def main(self):
        n, shelfWidth = tuple(int(x) for x in input().split(" "))
        w = list(int(x) for x in input().split(" "))
        h = list(int(x) for x in input().split(" "))
        books = list()
        for i in range(n):
            books.append([w[i], h[i]])
        print(self.minHeightShelves(books, shelfWidth))

if __name__ == "__main__":
    S = Solution()
    S.main()
```

Code (C++) for Incorrect Greedy Approach (Draft)

```cpp
#include <iostream>
#include <climits>
using namespace std;

#define INF INT_MAX

struct State {
    int currentW;
    int currentH;
    int totalH;
```



```cpp
};

int main() {
    int n, shelfWidth;
    cin >> n >> shelfWidth;
    int w[n], h[n];
    State dp[n][2];
    for (int i = 0; i < n; i++) cin >> w[i];
    for (int i = 0; i < n; i++) cin >> h[i];
    dp[0][0].totalH = INF;
    dp[0][1].currentH = h[0]; dp[0][1].currentW = w[0]; dp[0][1].totalH = h[0];
    for (int i = 1; i < n; i++) {
        State last;
        if (dp[i - 1][0].totalH <= dp[i - 1][1].totalH) last = dp[i - 1][0];
        else last = dp[i - 1][1];
        dp[i][0].currentW = last.currentW + w[i]; dp[i][0].currentH = max(last.currentH, h[i]);
        dp[i][0].totalH = last.totalH - last.currentH + dp[i][0].currentH;
        if (dp[i][0].currentW > shelfWidth) dp[i][0].totalH = INF;
        dp[i][1].currentW = w[i]; dp[i][1].currentH = h[i]; dp[i][1].totalH = last.totalH + h[i];
    }
    if (dp[n - 1][0].totalH <= dp[n - 1][1].totalH) cout << dp[n - 1][0].totalH << endl;
    else cout << dp[n - 1][1].totalH << endl;
}
```

Test Cases

Input: $N \leftarrow 3$; $W_{\lim} \leftarrow 2$; $W \leftarrow [1,1,1]$; $H \leftarrow [1,2,2]$
Output: 3
Input: $N \leftarrow 7$; $W_{\lim} \leftarrow 4$; $W \leftarrow [1,2,2,1,1,1,1]$; $H \leftarrow [1,3,3,1,1,1,2]$
Output: 6