HW06

Jiaxin Li, Sunjun Gu

Octber 26, 2020

1 Question 1

Algorithm 1 JobOrder

Input: An array J[1,...n] of tuples (t_i, w_i).This is a set of n jobs with a processing time t_i and a weight w_i for each job

Output: The jobs order which minimizes the weighted sum of the completion times, $\sum_{i=1}^{n} w_i C_i$ (C_i denote the finishing time of job i)

procedure JOBORDER($J[1, 2, ..., n]$)
S $\leftarrow$ empty array with size of n
O $\leftarrow$ empty array with size of n
for i = 1,2...,n **do**
  $\frac{w_i}{t_i} \leftarrow \frac{J[i][1]}{J[i][0]}$
  S[i] $\leftarrow (\frac{w_i}{t_i}, i)$
end for
Sort S in descending order by $\frac{w_i}{t_i}$
for j = 1,2...,n **do**
  O[j] $\leftarrow S[j][1]$
end for
return O
end procedure=0

Intuition: Using Greedy approach, pick the first job with the highest $\frac{w}{t}$,then the second highest,the third highest,... If meet ties, go with whichever job come first in the input list.
Symbol Description: G is greedy solution. Label G's job schedule is $g_1, g_2, ..., g_j, ..., g_i, ...g_n.g_j, g_i$ are jobs. By greedy approach, if g_j is earlier than g_i, $j < i$,then $\frac{w_j}{t_j} \geq \frac{w_i}{t_i}$.
We assume that there is some other optional solution **S**$\neq$**G**. Label S's job schedule is $s_1, s_2, ..., s_n$.
Optimality and validity: Optimality is minimizing the weighted sum of the completion times; validity is scheduling all jobs and they are non - overlapping.
Claim: Exchange the inverted jobs(if s_j is earlier than s_i, $j < i$,then $\frac{w_j}{t_j} \leq \frac{w_i}{t_i}$) on solution S to be closer to G, the new solution S' is still optimal and valid.

Proof of Correctness By Exchange Argument:
We want to show that we can change the job order made in S to get closer to G without decreasing optimality or affecting validity.
On description of G, every early job must have higher $\frac{w}{t}$ than the subsequent job. There would not existed two consecutive items which are inverted(higher $\frac{w}{t}$ job is after the lower $\frac{w}{t}$).Since S$\neq$G, it would not follow the definition of G, there are must at least two consecutive items s_i,s_j which are

inverted.(if $j > i$,then $\frac{w_j}{t_j} \geq \frac{w_i}{t_i}$). If there is no consecutive inversion pair of jobs, the S will be exactly same as G (Proof of contra-diction). We will **exchange** them and show this retains optimality and validity.
Let $j = i + 1$, so we have S solution of job schedule $s_1, s_2, ...s_x, s_i, s_j, s_y..., s_n$,the finish time of s_x is C_x, the start time of s_y is $C_x + t_i$+t_j. No matter exchange or not, total time of s_i and s_j is t_i+t_j. The finish time of s_x and start time of s_y would not change, so finish time of s_y is not changed too . Therefore, if we swap the job s_i and s_j, it would not affect other jobs' weighted sum of the completion time because it would not affect their finish time. Now, we swap job s_i,s_j,job schedule is $s_1, s_2, ...s_x, s_j, s_i, s_y..., s_n$, because the above proof shows swap does not affect other jobs's weighted sum of the completion, compare the weighted sum of the completion time of s_i,s_j and s_j,s_i.
the weighted sum of the completion time of s_i,s_j is($(C_x + t_i) = C_i$,$(C_x + t_i + t_j) = C_j$):

$$w_i(C_x + t_i) + w_j(C_x + t_i + t_j) \tag{b}$$

the weighted sum of the completion time of s_j,s_i is($(C_x + t_j) = C_j$,$(C_x + t_i + t_j) = C_i$):

$$w_j(C_x + t_j) + w_i(C_x + t_i + t_j) \tag{a}$$

We already know that

$$\frac{w_j}{t_j} \geq \frac{w_i}{t_i} \tag{c}$$

We want to show the optimality that is (a) $\leq$ (b), do calculation if (a) $\leq$ (b):

$$\begin{aligned} w_j(C_x + t_j) + w_i(C_x + t_i + t_j) &\leq w_i(C_x + t_i) + w_j(C_x + t_i + t_j) \\ w_jC_x + w_jt_j + w_iC_x + w_it_i + w_it_j &\leq w_iC_x + w_it_i + w_jC_x + w_jt_i + w_jt_j \\ w_it_j &\leq w_jt_i \\ \frac{w_j}{t_j} &\geq \frac{w_i}{t_i} \end{aligned}$$

By the calculation above and (c), (a) $\leq$ (b) is true. Therefore, the swap retains optimality because it doesn't make the weighted sum of completion time lager. And it retains validity as well because the new solution still schedule all jobs and they are non - overlapping.
Exchange makes S closer to G by fixing the inversion jobs. There are a finite of inversions, so we can fix them and finally change S into G.That is G is the optimal and valid solution to this problem.

Proof of Time Complexity :
From pseudo code line 3-5, we need to compute $\frac{w_i}{t_i}$ for each job, which takes O(n) time. For line 6, we need to sort jobs based on $\frac{w_i}{t_i}$ and by the previous lecture knowledge, sort algorithm takes O($nlogn$) time. For line 7-9, we need to record the job order, which takes O(n) time. Other lines take O(1) constant time work. Therefore, the toal run time is O(n) + O($nlogn$) + O(n) + O(1) = O($nlogn$).

2 Question 3

2.1 Brief introduction to algorithm:

This is only a brief description of the algorithm, detailed implementation is in the pseudocodes
1. **OrderItems:** Sort the items by $c_i - p_i$ on descending order with Quick Sort algorithm
2. **ComputeInitialFunding:** Given the ordered items in the last step, iterate the items from the

end to the **beginning**, let m_i be the minimum remaining money **before** buying the i-th item

$m_n = c_n$

$m_i = max(m_{i+1} + p_i, c_i)$, where $1 \leq i < n$

m_1 is exactly the minimum initial funding the person needs.

Algorithm 2 MinimumInitialFunding

Input: An array C[0,...,n-1], where C[i] is the cost of the i-th item; An array P[0,...,n-1], where P[i] is the price of the i-th item

Output: An array O[0,...,n-1], where O[i] is the initial index of the i-th item in the optimal order

The minimum initial funding, m

procedure MINIMUMINITIALFUNDING(C[0,...,n-1], P[0,...n-1])
O[0,...,n-1], G[0,...,n-1] ← OrderItems(C[0,...,n-1], P[0,...n-1])
m ← ComputeInitialFunding(G[0,...,n-1]))
return O[0,...,n-1], m

Algorithm 3 OrderItems

Input: An array C[0,...,n-1], where C[i] is the cost of the i-th item; An array P[0,...,n-1], where P[i] is the price of the i-th item

Output: An array G[0,...,n-1] of tuples (c_i, p_i), where c_i is the cost of the item and p_i is the price of the item, $c_i - p_i$ is in descending order;

An array O[0,...,n-1], where O[i] is the initial index of the i-th item in the optimal order

procedure ORDERITEMS(C[0,...,n-1], P[0,...,n-1])
Coupon[] ←a new empty array
for i in $0 \rightarrow n-1$ **do**
    append tuple (C[i]-P[i], i) at the end of Coupon[]
end for
Sort Coupon[0,...,n-1] by the first entry of each tuple by descending order with QuickSort algorithm
G[] ← a new empty array
O[] ← a new empty array
for i in $0 \rightarrow n-1$ **do**
    $index \leftarrow Coupon[i][1]$
    append tuple (C[index], P[index]) at the end of G[]
    append index at the end of O[]
end for
return O[0,...,n-1],G[0,...,n-1]
end procedure=0

2.2 Proof of correctness:

Firstly, we want to prove that the second sub-algorithm **ComputeInitialFunding** could return the correct minimum initial funding of a specific order (not necessary to be the optimal one), given a sequence of items with their costs c_i and prices p_i.

Proposition:

Let the proposition to prove, P(i) ($1 \leq i \leq n$), is that Algorithm **ComputeInitialFunding** could return the correct minimum remaining funding before buying the i-th item.

Algorithm 4 ComputeInitialFunding

Input: An array G[0,...n-1] of tuples (c_i, p_i), where c_i is the cost of the item and p_i is the price of the item

Output: m, which is the minimum initial funding given the order of items in the input

procedure ComputeInitialFunding(G[0,...,n-1])
M[0,...,n-1] $\leftarrow$ a new array
M[n-1] $\leftarrow G[n-1][0]$
for i in $n-2 \rightarrow 0$ **do**
  M[i] $\leftarrow Max(M[i+1] + G[i][1], G[i][0])$
end for
return M[0]

Base Case:
When i = n, P(n) is correct because the minimum remaining funding before buying the n-th item should be the cost of the last item, c_n.

Inductive Step:
Induction Hypothesis: Suppose P(k+1) $(1 < (k+1) \leq n)$ is correct, which means the algorithm could get the correct minimum remaining funding before buying the (k+1)-th item.

To prove P(k) is correct:
Given the restrictions, we know after buying the k-th item, the remaining money will be reduced by p_k. We have already knew we should have at least m_{k+1} money to buy the items after the k-th items (By Induction Hypothesis, P(k+1) is correct). So before buying the k-th item, we must have at least $m_{k+1} + p_k$.

In another case, the cost of the k-th item is larger than $m_{k+1} + p_k$, which means we need at least c_k before buying the k-th item.

Combining the above two situations, the larger one between $m_{k+1} + p_k$ and c_k will be the minimum remaining funding before buying the k-th item. So the algorithm's output, $max(m_{k+1} + p_k, c_k)$, is correct. P(k) is correct.
Therefore, P(k+1) $\Rightarrow$ P(k) By mathematical induction principle, P(1) is correct, which means the algorithm could return the correct minimum initial funding given a sequence of items.

Next step, we will illustrate why order the items by $c_i - p_i$ in the descending order will give the optimal order.

Claim:
Given a specific order of n items S, $(s_1, s_2, ..., s_n)$, with their corresponding cost c_i and price p_i, the correct minimum initial funding must be the largest value in the set of values: $\{c_i + \sum_{j=1}^{i-1} p_j | 1 \leq i \leq n\}$

Proof of the claim:
Firstly, we want to prove that the minimum initial funding must be one of the values in the set $\{c_i + \sum_{j=1}^{i-1} p_j | 1 \leq i \leq n\}$. Because we have already proved that **ComputeInitialFunding** can return the correct minimum funding, we can equally prove the value returned by **ComputeInitialFunding** must one value in the set $\{c_i + \sum_{j=1}^{i-1} p_j | 1 \leq i \leq n\}$.
The algorithm is basically making a sequence of decisions to decide take $m_{i+1} + p_i$ or c_i as the value

of m_i.
Case 1: In the procedure, there are at least one times that the algorithm takes c_i:
Let the last time it decides to take c_i is calculating $m_k, 1 \leq k \leq n-1$, so that $m_k = c_k$. Because this is the last decision to take c_i instead of $m_{i+1} + p_i$, the remaining decision must take $m_{i+1} + p_i$. The m_1 returned by the algorithm must be $m_k + \sum_{j=1}^{k-1} p_j = c_k + \sum_{j=1}^{k-1} p_j$, which belongs to $\{c_i + \sum_{j=1}^{i-1} p_j | 1 \leq i \leq n\}$
Case 2: In the procedure, there is no decision to take c_i. In such case, the algorithm just returns $c_n + \sum_{j=1}^{n-1} p_j$, which belongs to $\{c_i + \sum_{j=1}^{i-1} p_i | 1 \leq i \leq n\}$

Therefore, the minimum initial funding must be a value belonging to set $\{c_i + \sum_{j=1}^{i-1} p_i | 1 \leq i \leq n\}$.

Now we consider if the initial funding is less than an arbitrary value in the set, which means $F < c_i + \sum_{j=1}^{i-1} p_i$.
$\Rightarrow F - \sum_{j=1}^{i-1} p_i < c_i$.
So after buying i-1 items, the remaining funding will be less than the cost of the i-th item, which will cause invalidation. So the initial funding must be not less than any of the elements in the set. Combining with the conclusion that the minimum initial funding must be one of these elements, we know the minimum initial funding must be the largest element of this set $\{c_i + \sum_{j=1}^{i-1} p_i | 1 \leq i \leq n\}$.

Exchange Argument:
Let G be the order given by greedy solution, where $g_1, g_2, ..., g_n$ are items.
Let S ($S \neq G$) be some other solution, where $s_1, s_2, ...s_n$ are items.

In any S, there must be at least one pair of consecutive items that are inverse compared with G.
Because if there is no consecutive inversion pair of items, the S will be exactly G. (Proof of contradiction)

Let s_x and s_y be a pair of consecutive items that are inverse compared with G, which means $y = x + 1$ and $c_x - p_x \leq c_y - p_y$ (By Greed)

According to the claim above, the minimum initial funding of a specific order is the larget value in the set $\{c_i + \sum_{j=1}^{i-1} p_i | 1 \leq i \leq n\}$.
So now we need to examine how swapping s_x and s_y will affect the elements in this set.
1. When $1 \leq i < x$, the value of $c_i + \sum_{j=1}^{i-1} p_i$ doesn't change because the first x-1 items' order is not changed by swapping s_x and s_y.
2. When $y < i \leq n$, the value of $c_i + \sum_{j=1}^{i-1} p_j$ doesn't change, neither. Because though the order of x-th and y-th item is changed, that doesn't affect the sum of the prices of the first y items.

If the largest element (F) in the new set is in above 2 cases, then F is also in the original set, so the largest element of the original set is at least F, which means the swap doesn't make the minimum initial funding larger.

Let $\sum_{j=1}^{x-1} p_j = \alpha$
When $i = x$ and y, the original values are $c_x + \alpha$ and $c_y + \alpha + p_x$. However, after swapping, the values are $c_x + \alpha + p_y$ and $c_y + \alpha$.
$c_x - p_x \leq c_y - p_y \Rightarrow c_x + \alpha + p_y \leq c_y + \alpha + p_x$
$p_x > 0 \Rightarrow c_y + \alpha < c_y + \alpha + p_x$

If the largest element (F) of the new set is $c_y + \alpha$ or $c_x + \alpha + p_y$, $F \leq c_y + \alpha + p_x$, which is an element in the original set. That means the largest element in the original set is at least not less than F.

Above all, the exchange retains optimality, because the swap doesn't make the minimum initial funding larger in any cases.

Exchange makes S closer to G and there are finite exchanges between S and G. So G is the optimal order.

And the validation is proved by the mathematical induction in the proof of the correctness of **ComputeInitialFunding**.

Proof of correctness completed

2.3 Proof of Time Complexity:

1. Calculate $c_i - p_i$ for each item takes O(n) time complexity.
2. Order the items by $c_i - p_i$ takes O(nlog(n)) time complexity.
3. Organizing the input for ComputeInitialFunding in a descending order of c_i 3. Given the optimal order, calculating m_i is a dynamic programming procedure. There are in total n numbers to calculate. If we calculate them from m_n to m_1 and store each m in an array so that m_{i+1} is available while computing m_i , then computing each m will take constant time. The Time complexity of getting m_1 is $c \cdot O(n) = O(n)$

Overall, the time complexity is $O(n) + O(nlog(n)) + O(n) = O(nlog(n))$