3. Algorithm modTree [redacted]

Input: a tree
Output: same tree, modified where $g(v) = g(p^{g(v)}(v))$

Begin;
① for i:1 to n, do mark v; unvisited;
② count ← 1;
③ arr[1] := root of T;
④ count := count+1;
⑤ empty the stack
⑥ call modDFS(r)
End.

Procedure modDFS;

[redacted]

begin
① visit vertex v; mark v as visited;
② ptr ← L[v];
③ while (ptr ≠ nil) do
w ← ptr↑.vertex;
if (w is marked visited) then
push {w} onto the stack;
arr[count] := w;
if (arr[count-w]) then
w := arr[count-w];
else then
w := 1;
count := count+1;
modDFS(w);
fi;
ptr ← ptr↑.next; count := count-1; pop stack;

3. Lemma 1.1: ~~algorithm~~ procedure modDFS correctly visits every vertex once

Proof by Lemma 6.1 from the courseware

Lemma 1.2: ~~algorithm~~ procedure modDFS 'correctly' replaces $g(v)$ with $g(p^{g(v)}(v))$ if it exists, and $\perp$ elsewise

Proof: as $p^0(v)$ is the v, $p^1(v)$ is the parent and so on.
two things must be proven:

claim ① the array correctly updates.
Let us ~~assume~~ suppose to the contrary, that the array doesn't correctly update in accordance with the current adjacency list A.
This implies either array $\subsetneq A$ or array $\supsetneq A$.
Let the algorithm modDFS(x) begin. Initially, array is assigned its default values. When modDFS visits the next vertex w, w is appended to array, ensuring all visited vertices are correctly stored. Upon backtracking, when modDFS encounters an unvisited vertex, the index in array is decremented. Consequently, the array tracks the traversal along the adjacency list, ensuring array $\subseteq A$ at all times.
This assumption leads to a contradiction, violating the correctness of array's updates!

claim ② backtracking aligns with adjacency list traversal
During backtracking, modDFS revisits vertices along the path stored in array. If no unvisited vertex exists, modDFS correctly transitions to the parent vertex $p(v)$, ensuring the traversal is consistent with A.
Since the adjacency list A contains all vertices, array's alignment with A ensures the correctness of backtracking.

The assumptions of incorrect array updates or misalignment with A lead to contradictions. Thus, $g(v)$ is updated as stated, and the algorithm proceeds correctly. From my proof, the array consistently mirrors the traversal of the adjacency list A. For example, if $A[count]$ evaluates to true, it implies the existence of an element, and the algorithm correctly updates $g(v)$ to $g(p(v))$, and if $A[count]$ is false or empty, $g(v)$ is set to 1. Thus lemma 1.2 is proven.

3. Collary 1.2: modDFS returns a tree T where all $g(v) = g(p^{g(v)}(v))$ $\forall v$.

By Lemma 1.2 modDFS updates g(v) correctly

Time Complexity Analysis: By Lemma 6.2 in courseware this runs in $O(|V| + |E|) = O(|V| + 2|V|) = O(3|V|) = O(|V|)$, so by the question $O(|V|) = O(n)$ so it ~~seems~~ in $O(n)$ as no new procedures are $= O(1)$.

Key Idea: Create an array alongside DFS and use it to track the adjacency list to keep on $O(1)$ time for ancestor lookup.

Q.E.D. ■

1.

```
algorithm chocolateOrder(prices, coupons);

input: array prices of size n, where prices[i] is the price of the i^th chocolate
       array coupons of size n, where coupons[i] = (coupon_value, brand_for_coupon)

output: order to buy the chocolates, to ensure total price is minimized

begin
  n := length(prices);    // determine # of chocolates

  used := [0, 0, ..., 0];    //initialize arrays
  coupon_given := [1, 1, ..., 1];
  chain := [0, 0, ..., 0];
  order := [1, 1, ..., 0];
  order_index := 1;

  for i from to n do    // populate coupon_given array
    coupon_given[i] := coupons[i].brand_for_coupon;
  eof;

  chocolates_left := n;        //process all chocolates, until one is done
  while chocolates_left > 0 do
    start_chocolate := -1;  // find the starting chocolate for a new chain
    for i from 1 to n do
      if used[i] == 0 then
        start_chocolate := i;
        break;
      end if;
    eof

    chain_size := 0;    // form chain starting from selected chocolate
    current_chocolate := start_chocolate;

    while used[current_chocolate] == 0 do
      chain[chain_size+1] := current chocolate;  //add to the chain
```

```
        chain_size := chain_size + 1;

        current_chocolate := coupon_given[current_chocolate];
    endwhile

    best_chocolate := -1;        //find the best chocolate within the chain
    max_coupon_value := -1;

    for i from 1 to chain_size do
        chocolate := chain[i];

        if used[chocolate] == 0 and
            coupons[chocolate].value > max_coupon_value;
            then max_coupon_value := coupons[chocolate].value;
            best_chocolate := chocolate;
        endif
    eof

    order[order_index+1] := best_chocolate;    //buy best chocolate, record in purchase order

    order_index := order_index + 1;

    used[best_chocolate] := 1;

    chocolates_left := chocolates_left - 1;

    current_chocolate := coupon_given[best_chocolate];    //processing rest of chain

    while used[current_chocolate] == 0 do
        order[order_index + 1] := current_chocolate;

        order_index := order_index + 1;

        used[current_chocolate] := 1

        chocolates_left := chocolates_left - 1;
```

current_chocolate := coupon_given[current_chocolate];
endwhile
endwhile

return order;
end.

Proof of correctness: Proof by Induction

Lemma 1.1 : The algorithm chocolateOrder() successfully returns the order to buy the chocolates, to minimize the final total price.

Basis Case: (n=1)

if there is only one chocolate, then the chain consists soley of this single chocolate.
the algorithm selects and buys it, recording the purchase order as [1].
since there are no other chocolates or coupons, this is proven as correct.

Inductive hypothesis: (n=k)

Assume the algorithm correctly determines the purchase order and minimizes the cost for n=k chocolates.

Inductive Step: (n=k+1)

for n=k+1 chocolates:

① chain formation:
The algorithm forms a valid chain starting from the first unbought chocolate.
Each chocolate in the chain points to another chocolate via its coupon, ensuring all chocolates in the chain are processed.

② Best chocolate selection:
The algorithm selects the chocolate with the largest coupon value, ensuring the largest discount is achived for O subsequent chocolates in the chain.

③ Chain Processing:
After buying the best chocolate, the algorithm processes the remaining chocolate in the chain using their coupons.

④ Iterative Chains:
The algorithm repeats this process for all unbought chocolates, ensuring no chocolate is skipped.

Since, the inductive hypothesis ensures correctness for $n = k$, and the algorithm processes the $(k+1)^{th}$ chocolate using the same logic, the algorithm correctly determines the purchase order for all n such that the total cost is minimum.

Time Complexity Analysis:

Populating coupons_given[] takes $O(n)$ time.

In the main while loop, each chocolate is processed exactly once.

Chain formation involves traversing the chocolates in the chain, which is consistently $O(n)$ across all chains.

Selecting the best chocolates within a chain involves a single pass over the chain, contributing to an $O(n)$ time complexity across all chains.

Adding chocolates to the purchase order is $O(1)$ per chocolate.

Therefor, the total time complexity is $O(n) + O(n) + O(n) = O(n)$

Therefore, the time-complexity for chocolateOrder is $O(n)$. [redacted]

2. algorithm charger

input: $b_1, b_2, \ldots, b_n$
$c_1, c_2, \ldots, c_n$

output: list of hours to reboot the recharger for optimal energy delivery

1. initialize $opt(0, j) := 0$ for all $j := 0$ to n.

2. for $i := 1$ to $n+1$ do:
   a. compute $opt(i, 0)$:
      i.) initialize $opt(i, 0) := 0$.
      ii). for $k := 1$ to $i-1$ do:
         - if $opt(i, 0) < opt(i-1, k)$, update $opt(i, 0) := opt(i-1, k)$.
         - record the reboot time: $reboot(i, 0) := k$

   b. if $i \leq n$, compute $opt(i, j)$ for all $j := 1$ to n:
      for $j := 1$ to n do:
         $opt(i, j) := opt(i-1, j-1) + min(b_i, c_i)$.

3. backtracking to find the reboot hours
   a). initialize $rb0 := n+1$, $k := 0$.
   b). repeat:
      i). increment k: $k := k+1$
      ii). calculate $rbk := rbk-1 - reboot(rbk-1, 0)$.
   until $rbk = 0$

4. output the reboot ~~hours~~ hours:
   ~~a). initialize~~
   for $i := k-1$ down to 1 do:
      output rb_i.

end of algorithm

Lemma 1: $Opt(n) = \max_{1 \leq j \leq n} Opt(n, j)$

Proof: If the recharger is rebooted k hours before hour n, then $opt(n) = opt(n, k)$ for any j, $1 \le j \le n$

$opt(n,k) \ge opt(n,j) \rightarrow opt(n) = \max_{1 \le j \le n} opt(n,j)$

Lemma 2: If the recharger isn't rebooted in hour n, ($j \neq 0$), "optimal substructure"

$$opt(n,j) = opt(n-1, j-1) + \min(b_n, c_n).$$

Proof: Since no reboot occurs in hour n, the kWh delivered in this hour is $\min(b_n, c_n)$. For previous hours, the value is $Opt(n-1, j-1)$, we get:

$$Opt(n,j) = opt(n-1, j-1) + \min(b_n, c_n)$$

Lemma 3: If the recharger is ~~hour~~ rebooted in hour i ($j = 0$):

$$opt(i,0) = \max_{1 \le k \le i-1} opt(i-1, k)$$

Proof: Rebooting at hour i delivers 0 kWh for that hour. Thus the maximum kWh for the first i hours is the maximum value for the first $i-1$ hours:

$$Opt(i,0) = \max_{1 \le k \le i-1} Opt(i-1, k).$$

Recurrence Relation: for $1 \le i \le n+1$:

if $j \neq 0$:

$opt(i,j) = opt(i-1, j-1) + \min(b_i, c_i)$.

if $j = 0$

$opt(i,0) = \max_{1 \le k \le i-1} opt(i-1, k)$.

Lemma 4 (Basis case): $opt(0, j) = 0, \forall j, 0 \leq j \leq n$.

Proof: In 0 hours, no energy is delivered, so $opt(0, j) = 0$.

Lemma 5 (Inductive Step): The algorithm correctly computes all $opt(i, j)$ values. I will use induction on I.

Basis case ($i = 0$): From lemma 4, we initialize $opt(0, j) = 0$ for all j. This satisfies the definition of $Opt(0, j)$ as no energy is delivered in 0 hours.

Inductive Hypothesis: Assume $opt(k, j)$ is correctly computed for all j and all $k \leq i$.

Inductive Step: ($i = m$) For $Opt(m, 0)$: By Lemma 3 we compute $Opt(m, 0)$ as the maximum of $opt(m-1, k)$ for $1 \leq k \leq m-1$.

the loop iterates through all such k and correctly updates $Opt(m, 0)$.

2. For $Opt(m, j)$ where $j > 0$:

By Lemma 2, if no reboot occurs in ~~hours~~ hour m, we compute $Opt(m, j)$ as $Opt(m-1, j-1) + \min(bm, cm)$.

Since $Opt(m-1, j-1)$ is computed in previous iterations, this ~~update~~ update is correct.

Thus, by induction, $Opt(i, j)$ is correctly computed for all i and j.

Proof of Correctness for Backtracking: Using Induction, we show that the backtracking step produces the correct reboot hours.

Basis case ($n = 1$): In the first iteration of the repeat loop, $rb1 = (n+1) - reboot(n+1, 1)$ gives the last reboot hour, which is correct as per the definition of $Opt[n]$.

Inductive Hypothesis: Assume for $h = k-1$, rbh, correctly represents the k^{th}-to-last reboot hour.

Inductive Step ($h=k$): During the k^{th} iteration, $rbk = rbk-1 - reboot(rbk-1, 0)$ correctly computes the next reboot time as it subtracts the interval since the previous reboot. Thus, by induction, the backtracking process outputs all reboots hours in reverse order correctly.

Time Complexity Analysis: Initialize $opt(0,5)$:

This requires $O(n)$, as it involves setting $n+1$ values to 0.

Outer loop ($i=1$ to $n+1$):

Computing $opt(i, 0)$: For each i, finding the maximum across $1 \le k \le i-1$ takes $O(i)$. Over all iterations, this contributes:

$$\sum_{i=1}^{n+1} O(i) = O(n^2).$$

Computing $opt(i, j)$ for $j=1$ to n: Each computation is $O(1)$ and there are n such computations per i. This is $O(n^2)$.

Adding up all components: $O(n) + O(n^2) + O(n^2) + O(n) = O(n^2)$

Key Idea: leveraging dynamic prog with optimal substructure to calculate the maximum energy delivered over n hours, while minimizing the number of reboots.

The solution is built around the state $Opt(i, j)$, which represents the maximum energy delivered in the first i hours when the last reboot occured j hours before hour i.

For any given hour i, there are two possibilities:

① ~~No reboot~~
The energy delivered depends on the prior state $Opt(i-1, j-1)$ and the capacity of the recharger in hour i.

② A reboot occurs in hour i
This results in the recharger delivering 0 kWh during i, and the solution depends on the best configuration from earlier hours.

'Optimal Substructure'
By recursively solving for smaller subproblems, can build solution to full problem.

After computing all $Opt(i, j)$ values, the algorithm traces backs to identify the specific hours when reboots should occur. This is optimal and uses recurrence relations to get a TCR of $O(n^2)$.

Q.E.D.

1. General/Key Ideas:

The algorithm starts by getting an array ~~for~~ used[] to track whether a chocolate has already been bought. coupon_given identifies which brand each chocolate provides a coupon for.

A variable total numbers of chocolates left to buy is initialized to n. An order[] to store the order of purchased chocolates.

It works by iterating through all chocolates until all have been bought. Each iteration finds a new chain of chocolates to process.

So a chain starts with the first unbought chocolate found while iterating over the array of chocolates.
all

Then the algorithm will trace through the coupons to identify all chocolates in the chain.

The chain is stored in an array, chain[] to process later.

Chains are formed by following the coupon_given[] mapping, ensuring every chocolate in the chain is considered.

From the chocolate in ~~the~~ a chain, the algorithm selects the 'best chocolate' meaning the chocolate that offers the <u>largest coupon value for another brand.</u>

This will ensure the maximum discount for subsequent purchases, leading to overall cost minimization.

So, this best chocolate is bought first at its full price (as no coupon can be used for it since it hasn't been processed yet).

The algorithm updates order[] and makes this chocolate as bought in the used[] array.

Now starting from the best chocolate the algorithm processes the remaining chocolates.