3562. Maximum Profit from Trading Stocks with Discounts Solved

Hard Topics Companies Hint

You are given an integer `n`, representing the number of employees in a company. Each employee is assigned a unique ID from 1 to `n`, and employee 1 is the CEO. You are given two **1-based** integer arrays, `present` and `future`, each of length `n`, where:

- `present[i]` represents the **current** price at which the i^{th} employee can buy a stock today.
- `future[i]` represents the **expected** price at which the i^{th} employee can sell the stock tomorrow.

The company's hierarchy is represented by a 2D integer array `hierarchy`, where `hierarchy[i] = [u_i, v_i]` means that employee u_i is the direct boss of employee v_i.

Additionally, you have an integer `budget` representing the total funds available for investment.

However, the company has a discount policy: if an employee's direct boss purchases their own stock, then the employee can buy their stock at **half** the original price (`floor(present[v] / 2)`).

Return the **maximum** profit that can be achieved without exceeding the given budget.

Note:

- You may buy each stock at most **once**.
- You **cannot** use any profit earned from future stock prices to fund additional investments and must buy only from `budget`.

Example 1:

Input: `n = 2, present = [1,2], future = [4,3], hierarchy = [[1,2]], budget = 3`

Output: 5

Explanation:

- Employee 1 buys the stock at price 1 and earns a profit of `4 - 1 = 3`.
- Since Employee 1 is the direct boss of Employee 2, Employee 2 gets a discounted price of `floor(2 / 2) = 1`.
- Employee 2 buys the stock at price 1 and earns a profit of `3 - 1 = 2`.
- The total buying cost is `1 + 1 = 2 <= budget`. Thus, the maximum total profit achieved is `3 + 2 = 5`.

Example 2:

Input: `n = 2, present = [3,4], future = [5,8], hierarchy = [[1,2]], budget = 4`

Output: 4

Explanation:

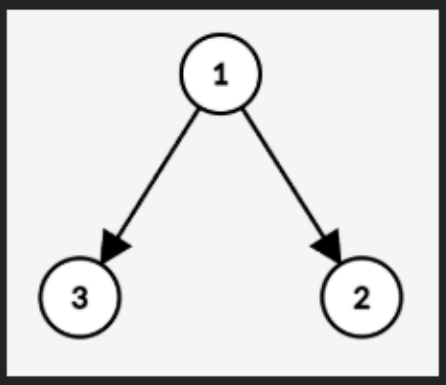
- Employee 2 buys the stock at price 4 and earns a profit of `8 - 4 = 4`.
- Since both employees cannot buy together, the maximum profit is 4.

Example 3:

Input: `n = 3, present = [4,6,8], future = [7,9,11], hierarchy = [[1,2],[1,3]], budget = 10`

Output: 10

Explanation:



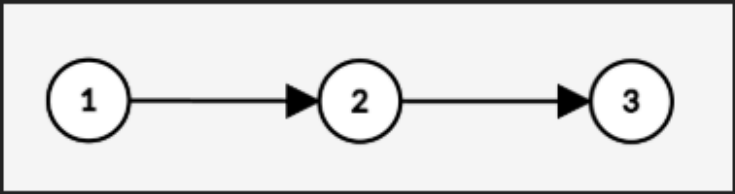
- Employee 1 buys the stock at price 4 and earns a profit of `7 - 4 = 3`.
- Employee 3 would get a discounted price of `floor(8 / 2) = 4` and earns a profit of `11 - 4 = 7`.
- Employee 1 and Employee 3 buy their stocks at a total cost of `4 + 4 = 8 <= budget`. Thus, the maximum total profit achieved is `3 + 7 = 10`.

Example 4:

Input: `n = 3, present = [5,2,3], future = [8,5,6], hierarchy = [[1,2],[2,3]], budget = 7`

Output: 12

Explanation:



- Employee 1 buys the stock at price 5 and earns a profit of `8 - 5 = 3`.
- Employee 2 would get a discounted price of `floor(2 / 2) = 1` and earns a profit of `5 - 1 = 4`.
- Employee 3 would get a discounted price of `floor(3 / 2) = 1` and earns a profit of `6 - 1 = 5`.
- The total cost becomes `5 + 1 + 1 = 7 <= budget`. Thus, the maximum total profit achieved is `3 + 4 + 5 = 12`.

Constraints:

- `1 <= n <= 160`
- `present.length, future.length == n`
- `1 <= present[i], future[i] <= 50`
- `hierarchy.length == n - 1`
- `hierarchy[i] == [ui, vi]`
- `1 <= ui, vi <= n`
- `ui != vi`
- `1 <= budget <= 160`
- There are no duplicate edges.
- Employee 1 is the direct or indirect boss of every employee.
- The input graph `hierarchy` is **guaranteed** to have no cycles.

Python:

```
from typing import List

class Solution:
    def maxProfit(self, n: int, present: List[int], future: List[int], hierarchy: List[List[int]], budget: int)
-> int:
        tree = [[] for _ in range(n)]
        for u, v in hierarchy:
            tree[u - 1].append(v - 1)

        dp = [[[0] * (budget + 1) for _ in range(2)] for _ in range(n)]
        self.dfs(0, present, future, tree, dp, budget)
        return max(dp[0][0])

    def dfs(self, u, present, future, tree, dp, budget):
        children = tree[u]
        child_dps = []
        for v in children:
```

```
            self.dfs(v, present, future, tree, dp, budget)
            child_dps.append((dp[v][0], dp[v][1]))

        for parentBought in range(2):
            price = present[u] // 2 if parentBought else present[u]
            profit = future[u] - price

            # Option 1: not buying u
            base = [0] * (budget + 1)
            for c0, _ in child_dps:
                next_base = [0] * (budget + 1)
                for b in range(budget + 1):
                    if base[b] == 0 and b != 0:
                        continue
                    for k in range(budget - b + 1):
                        next_base[b + k] = max(next_base[b + k], base[b] + c0[k])
                base = next_base

            curr = base[:]

            # Option 2: buying u
            if price <= budget:
                base = [0] * (budget + 1)
                for _, c1 in child_dps:
                    next_base = [0] * (budget + 1)
                    for b in range(budget + 1):
                        if base[b] == 0 and b != 0:
                            continue
                        for k in range(budget - b + 1):
                            next_base[b + k] = max(next_base[b + k], base[b] + c1[k])
                    base = next_base

                for b in range(price, budget + 1):
                    curr[b] = max(curr[b], base[b - price] + profit)

            dp[u][parentBought] = curr
```

JavaScript:

```
/**
 * @param {number} n
 * @param {number[]} present
 * @param {number[]} future
 * @param {number[][]} hierarchy
 * @param {number} budget
```

```
 * @return {number}
 */
function maxProfit(n, present, future, hierarchy, budget) {
  const employeeCount = n;
  const budgetLimit = budget;
  const edgeCount = employeeCount - 1;
  const budgetStride = budgetLimit + 1;

  // A very small value used to mark unreachable states in DP
  const MIN_PROFIT = -1_000_000_000;

  // Store prices in typed arrays for faster indexed access (1-indexed by employee ID)
  const presentPrice = new Int16Array(employeeCount + 1);
  const futurePrice = new Int16Array(employeeCount + 1);
  const discountedPrice = new Int16Array(employeeCount + 1);

  // Initialize price arrays
  for (let employeeId = 1; employeeId <= employeeCount; employeeId++) {
    const presentValue = present[employeeId - 1] | 0;
    const futureValue = future[employeeId - 1] | 0;

    presentPrice[employeeId] = presentValue;
    futurePrice[employeeId] = futureValue;

    // Discounted price is floor(present / 2), bit shift is safe for positive integers
    discountedPrice[employeeId] = presentValue >> 1;
  }

  // Adjacency list head pointer for each boss
  const headEdgeIndex = new Int16Array(employeeCount + 1);
  headEdgeIndex.fill(-1);

  // Edge storage: toEmployee[i] is the child of edge i
  const toEmployee = new Int16Array(edgeCount);

  // nextEdgeIndex forms a linked list of edges for each boss
  const nextEdgeIndex = new Int16Array(edgeCount);

  // Build adjacency list from hierarchy input
  for (let index = 0; index < edgeCount; index++) {
    const bossId = hierarchy[index][0] | 0;

    toEmployee[index] = hierarchy[index][1] | 0;
    nextEdgeIndex[index] = headEdgeIndex[bossId];
```

```
  headEdgeIndex[bossId] = index;
}

// Stack for iterative DFS traversal to avoid recursion overhead
const traversalStack = new Int16Array(employeeCount);
let traversalStackSize = 0;

// Stores nodes in preorder, later processed in reverse for postorder effect
const preorderList = new Int16Array(employeeCount);
let preorderSize = 0;

// Start traversal from CEO (employee 1)
traversalStack[traversalStackSize++] = 1;

// Perform DFS to generate preorder traversal
while (traversalStackSize > 0) {
  traversalStackSize--;
  const currentNode = traversalStack[traversalStackSize];

  // Record visit order
  preorderList[preorderSize++] = currentNode;

  // Push all direct children into stack
  for (let edge = headEdgeIndex[currentNode]; edge !== -1; edge = nextEdgeIndex[edge]) {
    traversalStack[traversalStackSize++] = toEmployee[edge];
  }
}

// dp[(employeeId * 2 + parentBoughtFlag) * budgetStride + spent] = max profit
const dp = new Int32Array((employeeCount + 1) * 2 * budgetStride);
dp.fill(MIN_PROFIT);

// Accumulates child contributions when current employee is NOT bought
const childrenProfitWhenNotBought = new Int32Array(budgetStride);

// Accumulates child contributions when current employee IS bought
const childrenProfitWhenBought = new Int32Array(budgetStride);

// Temporary buffer used during knapsack merging
const mergeBuffer = new Int32Array(budgetStride);

// Process nodes bottom-up so children are fully computed first
for (let postIndex = preorderSize - 1; postIndex >= 0; postIndex--) {
  const employeeId = preorderList[postIndex];
```

```
// Reset child aggregation arrays before merging children
childrenProfitWhenNotBought.fill(MIN_PROFIT);
childrenProfitWhenBought.fill(MIN_PROFIT);
childrenProfitWhenNotBought[0] = 0;
childrenProfitWhenBought[0] = 0;

// Iterate through each direct child of the current employee
for (let edge = headEdgeIndex[employeeId]; edge !== -1; edge = nextEdgeIndex[edge]) {
  const childId = toEmployee[edge];

  // Merge DP assuming current employee is NOT bought
  mergeBuffer.fill(MIN_PROFIT);
  const childBaseNotBought = (childId << 1) * budgetStride;

  // Try all existing budget states from previous children
  for (let spentSoFar = 0; spentSoFar <= budgetLimit; spentSoFar++) {
    const currentProfit = childrenProfitWhenNotBought[spentSoFar];
    if (currentProfit === MIN_PROFIT) {
      continue;
    }

    // Remaining budget that can be allocated to this child
    const remainingBudget = budgetLimit - spentSoFar;

    // Combine child DP states with current aggregation
    for (let childSpent = 0; childSpent <= remainingBudget; childSpent++) {
      const childProfit = dp[childBaseNotBought + childSpent];
      if (childProfit === MIN_PROFIT) {
        continue;
      }

      const totalSpent = spentSoFar + childSpent;
      const totalProfit = currentProfit + childProfit;

      if (totalProfit > mergeBuffer[totalSpent]) {
        mergeBuffer[totalSpent] = totalProfit;
      }
    }
  }

  // Commit merged result for NOT bought case
  childrenProfitWhenNotBought.set(mergeBuffer);
```

```
  // Merge DP assuming current employee IS bought
  mergeBuffer.fill(MIN_PROFIT);
  const childBaseBought = childBaseNotBought + budgetStride;

  // Same merge logic but child sees parentBought = true
  for (let spentSoFar = 0; spentSoFar <= budgetLimit; spentSoFar++) {
    const currentProfit = childrenProfitWhenBought[spentSoFar];
    if (currentProfit === MIN_PROFIT) {
      continue;
    }

    const remainingBudget = budgetLimit - spentSoFar;

    for (let childSpent = 0; childSpent <= remainingBudget; childSpent++) {
      const childProfit = dp[childBaseBought + childSpent];
      if (childProfit === MIN_PROFIT) {
        continue;
      }

      const totalSpent = spentSoFar + childSpent;
      const totalProfit = currentProfit + childProfit;

      if (totalProfit > mergeBuffer[totalSpent]) {
        mergeBuffer[totalSpent] = totalProfit;
      }
    }
  }

  // Commit merged result for bought case
  childrenProfitWhenBought.set(mergeBuffer);
}

const nodeBaseParentNotBought = (employeeId << 1) * budgetStride;
const nodeBaseParentBought = nodeBaseParentNotBought + budgetStride;

// Case: do not buy current employee, profit comes only from children
for (let spent = 0; spent <= budgetLimit; spent++) {
  const inheritedProfit = childrenProfitWhenNotBought[spent];
  dp[nodeBaseParentNotBought + spent] = inheritedProfit;
  dp[nodeBaseParentBought + spent] = inheritedProfit;
}

const normalCost = presentPrice[employeeId];
const discountedCostValue = discountedPrice[employeeId];
```

```
  const normalProfit = (futurePrice[employeeId] - normalCost) | 0;
  const discountedProfit = (futurePrice[employeeId] - discountedCostValue) | 0;

  // Case: parent NOT bought, current employee pays full price
  for (let spent = normalCost; spent <= budgetLimit; spent++) {
    const childrenSpent = spent - normalCost;
    const childrenProfit = childrenProfitWhenBought[childrenSpent];
    if (childrenProfit === MIN_PROFIT) {
      continue;
    }

    const candidateProfit = childrenProfit + normalProfit;
    if (candidateProfit > dp[nodeBaseParentNotBought + spent]) {
      dp[nodeBaseParentNotBought + spent] = candidateProfit;
    }
  }

  // Case: parent bought, current employee uses discounted price
  for (let spent = discountedCostValue; spent <= budgetLimit; spent++) {
    const childrenSpent = spent - discountedCostValue;
    const childrenProfit = childrenProfitWhenBought[childrenSpent];
    if (childrenProfit === MIN_PROFIT) {
      continue;
    }

    const candidateProfit = childrenProfit + discountedProfit;
    if (candidateProfit > dp[nodeBaseParentBought + spent]) {
      dp[nodeBaseParentBought + spent] = candidateProfit;
    }
  }
}

// CEO has no parent, so parentBoughtFlag must be 0
const rootBase = (1 << 1) * budgetStride;
let bestProfit = 0;

// Find best achievable profit under budget constraint
for (let spent = 0; spent <= budgetLimit; spent++) {
  const profitValue = dp[rootBase + spent];
  if (profitValue > bestProfit) {
    bestProfit = profitValue;
  }
}
```

```
  return bestProfit;
}
```

Java:

```
class Solution {

    public int maxProfit(int n, int[] present, int[] future, int[][] hierarchy, int budget) {
        // Build tree
        List<Integer>[] tree = new List[n];
        for (int i = 0; i < n; i++) tree[i] = new ArrayList<>();
        for (int[] edge : hierarchy) {
            tree[edge[0] - 1].add(edge[1] - 1);
        }

        int[][][] dp = new int[n][2][budget + 1];  // [node][parentBought][budget]
        dfs(0, present, future, tree, dp, budget);

        // Answer is the max profit in dp[0][0][b] for any b <= budget
        int ans = 0;
        for (int b = 0; b <= budget; b++) {
            ans = Math.max(ans, dp[0][0][b]);
        }
        return ans;
    }

    private void dfs(int u, int[] present, int[] future, List<Integer>[] tree,
                     int[][][] dp, int budget) {
        // Base case: no children, init to 0
        for (int b = 0; b <= budget; b++) dp[u][0][b] = dp[u][1][b] = 0;

        // For each child, process recursively
        List<int[][]> childDPs = new ArrayList<>();
        for (int v : tree[u]) {
            dfs(v, present, future, tree, dp, budget);
            childDPs.add(new int[][]{dp[v][0], dp[v][1]});
        }

        // For parentNotBought and parentBought
        for (int parentBought = 0; parentBought <= 1; parentBought++) {
            int price = parentBought == 1 ? present[u] / 2 : present[u];
            int profit = future[u] - price;

            // Create DP array to fill for this u
```

```
            int[] curr = new int[budget + 1];

            // Option 1: don't buy u
            int[] base = new int[budget + 1];
            base[0] = 0;
            for (int[][] child : childDPs) {
                int[] next = new int[budget + 1];
                for (int b1 = 0; b1 <= budget; b1++) {
                    for (int b2 = 0; b1 + b2 <= budget; b2++) {
                        next[b1 + b2] = Math.max(next[b1 + b2], base[b1] + child[0][b2]);
                    }
                }
                base = next;
            }

            for (int b = 0; b <= budget; b++) {
                curr[b] = Math.max(curr[b], base[b]); // not buying u
            }

            // Option 2: buy u
            if (price <= budget) {
                int[] baseBuy = new int[budget + 1];
                baseBuy[0] = 0;
                for (int[][] child : childDPs) {
                    int[] next = new int[budget + 1];
                    for (int b1 = 0; b1 <= budget; b1++) {
                        for (int b2 = 0; b1 + b2 <= budget; b2++) {
                            next[b1 + b2] = Math.max(next[b1 + b2], baseBuy[b1] + child[1][b2]);
                        }
                    }
                    baseBuy = next;
                }

                for (int b = price; b <= budget; b++) {
                    curr[b] = Math.max(curr[b], baseBuy[b - price] + profit);
                }
            }

            dp[u][parentBought] = curr;
        }
    }
}
```