Anggota Kelompok :
Christoforus Indra Bagus Pratama (5025231124)

LAPORAN TUGAS 2 PAA

Soal 1 : Best Time to Buy and Sell Stock (Link Soal 1)

a. Deskripsi Soal

Diberikan sebuah array integer *prices*, di mana *prices[i]* adalah harga saham pada hari ke-i. Tujuannya adalah memaksimalkan keuntungan dengan melakukan satu kali transaksi: membeli satu saham pada suatu hari dan menjualnya pada hari yang berbeda di masa depan. User harus membeli saham sebelum menjualnya. Jika tidak ada cara untuk mendapatkan keuntungan, kembalikan 0.

Contoh Kasus 1 :

Input : prices = [7,1,5,3,6,4]

Output : 5 (Beli pada hari ke-2 (harga 1), jual pada hari ke-5 (harga 6). Keuntungan 6−1=5).

Contoh Kasus 2 :

Input : prices = [7,6,4,3,1]

Output : 0 (Tidak ada transaksi yang dapat menghasilkan keuntungan).

b. Abstraksi

Inti dari soal ini adalah **menemukan selisih harga maksimum** antara dua elemen dalam sebuah array, di mana elemen kedua (harga jual) harus berada setelah elemen pertama (harga beli). Secara matematis, yang dicari adalah nilai max(prices[j]−prices[i]) untuk semua j>i.

Masalah ini dapat diselesaikan dengan pendekatan Greedy. Strategi Greedy yang diterapkan di sini adalah membuat pilihan optimal di setiap langkah iterasi tanpa perlu mempertimbangkan dampak jangka panjang. Untuk setiap hari yang dipertimbangkan, kita perlu melakukan dua hal utama:

- **Melacak harga beli terendah** yang pernah ditemui hingga hari saat ini. Ini akan menjadi harga terbaik untuk membeli saham jika keputusan pembelian dibuat.
- **Menghitung potensi keuntungan** jika saham dijual pada harga hari ini, dengan asumsi pembelian dilakukan pada harga terendah yang tercatat. Kemudian, keuntungan maksimum yang telah dicatat akan diperbarui jika potensi keuntungan saat ini lebih besar.

Dengan strategi ini, saat array harga saham diiterasi, kita selalu memiliki harga beli "ideal" (terendah) untuk menghitung potensi keuntungan, sehingga secara kumulatif akan ditemukan keuntungan terbesar.

Model yang digunakan:

Kami menggunakan dua variabel untuk melacak status iterasi:

- *min_price* : Variabel ini menyimpan harga saham terendah yang telah terlihat dari awal iterasi hingga hari saat ini.
- *max_profit* : Variabel ini menyimpan keuntungan tertinggi yang telah dicatat sejauh ini.

Prosesnya melibatkan satu kali iterasi (loop) melalui array prices

c. Pseudocode

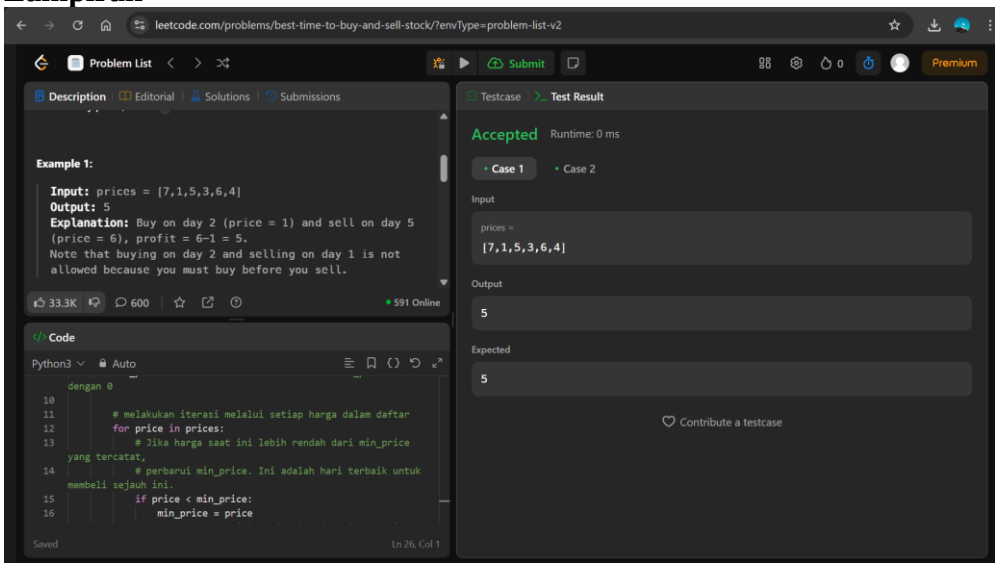
```
function maxProfit(prices):
  if prices is empty:     → Jika tidak ada harga, maka tidak ada transaksi yang dapat dilakukan, sehingga keuntungan adalah 0.
    return 0
  min_price = positive_infinity → harga saham pertama yang ditemui akan selalu lebih kecil dari 'min_price'
  max_profit = 0  → Keuntungan awal dianggap 0, karena tidak ada keuntungan yang bisa didapat jika tidak ada transaksi

  for each price in prices:→ Iterasi melalui setiap 'price' (harga saham) dalam daftar 'prices'
    if price < min_price:
      min_price = price
          → Jika 'price' saat ini TIDAK lebih kecil dari 'min_price' (yaitu, 'price' >= 'min_price'). Artinya, ada potensi untuk menjual
          saham pada harga saat ini, karena harganya lebih tinggi atau sama dengan harga beli terendah yang pernah ditemui
    else:
      current_profit = price - min_price
      max_profit = max(max_profit, current_profit)
          → bertujuan untuk menyimpan keuntungan terbesar, jadi perbarui 'max_profit' jika 'current_profit' lebih besar
  return max_profit  →'max_profit' akan berisi keuntungan maksimum yang bisa dicapai dari satu kali transaksi
```

d. Implementasi

Link Github : [Code Soal 1]

```python
from typing import List

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        if not prices:  # untuk menangani list kosong
            return 0

        min_price = float('inf')  #inisialisasi min_price dengan nilai tak terbatas
        max_profit = 0            #inisialisasi max_profit dengan 0

        # melakukan iterasi melalui setiap harga dalam daftar
        for price in prices:
            # Jika harga saat ini lebih rendah dari min_price yang tercatat,
            # perbarui min_price. Ini adalah hari terbaik untuk membeli sejauh ini.
            if price < min_price:
                min_price = price
            # Jika harga saat ini lebih tinggi dari min_price,
            # hitung potensi keuntungan dan perbarui max_profit jika lebih besar.
            else:
                max_profit = max(max_profit, price - min_price)

        return max_profit

if __name__ == "__main__":
    s = Solution()
```

e. Lampiran



Soal 2 : Climbing Stairs [(Link Soal 2)]

a. Deskripsi Soal

Diberikan sebuah tangga yang memerlukan n langkah untuk mencapai puncaknya. Setiap kali mendaki, kita hanya diperbolehkan mengambil 1 atau 2 langkah. Tugasnya adalah menentukan berapa banyak cara berbeda yang mungkin untuk mencapai puncak tangga.

Contoh Kasus:

Input : n = 2

- Output: 2
- Penjelasan: Terdapat dua cara untuk mencapai puncak:
    1. 1 langkah + 1 langkah
    2. 2 langkah

Input : n = 3

- Output: 3
- Penjelasan: Terdapat tiga cara untuk mencapai puncak:
    1. 1 langkah + 1 langkah + 1 langkah
    2. 1 langkah + 2 langkah
    3. 2 langkah + 1 langkah

b. Abstraksi Soal

Soal "Climbing Stairs" adalah contoh klasik dari masalah Dynamic Programming (DP). Kunci untuk memahami masalah ini terletak pada observasi bahwa jumlah cara untuk mencapai langkah ke-n bergantung pada jumlah cara untuk mencapai langkah-langkah sebelumnya.

Jika kita berada di langkah ke-n (puncak), langkah terakhir yang mungkin kita ambil adalah:

1. Satu langkah dari posisi (n−1).
2. Dua langkah dari posisi (n−2).

Ini berarti, ***total jumlah cara untuk mencapai langkah ke-n adalah jumlah cara untuk mencapai langkah ke-(n−1) ditambah jumlah cara untuk mencapai langkah ke-(n−2)***.

Hubungan rekursif ini dapat ditulis sebaga i:

`ways(n) = ways(n-1) + ways(n-2)`

Ini adalah relasi rekurensi yang sama dengan barisan Fibonacci.

Untuk memulai rekurensi ini, kita perlu mendefinisikan kasus dasar (base cases):

- Untuk n=1: Hanya ada 1 cara (ambil 1 langkah). Jadi, ways(1)=1.
- Untuk n=2: Ada 2 cara (1 langkah + 1 langkah, atau 2 langkah). Jadi, ways(2)=2.

Dari sini, kita bisa menghitung nilai selanjutnya:

- ways(3) = ways(2) + ways(1) = 2+1 = 3
- ways(4) = ways(3) + ways(2)= 3+2 = 5
- dan seterusnya.

Model Dynamic Programming (Bottom-Up / Tabulation):

Karena adanya sub-masalah yang tumpang tindih (misalnya, untuk menghitung ways(5), kita butuh ways(4) dan ways(3); ways(4) juga butuh ways(3) dan ways(2)), serta sifat struktur optimal (solusi untuk n dibangun dari solusi optimal untuk n−1 dan n−2), Dynamic Programming adalah pendekatan yang ideal.

Kami akan menggunakan pendekatan ***bottom-up (tabulation)***. Ini berarti kita akan mulai dari kasus dasar yang paling kecil dan secara iteratif membangun solusi hingga mencapai n

c. Pseudocode

```
function climbStairs(n):
  if n == 0:   → base case
    return 0   → Jika n adalah 0, tidak ada langkah, tidak ada cara untuk memanjat.
  if n == 1:
    return 1   → Jika n adalah 1, hanya ada 1 cara: (1 langkah).
  if n == 2:
    return 2   → Jika n adalah 2, ada 2 cara: (1 langkah + 1 langkah) atau (2 langkah).

  → Inisialisasi variabel untuk menyimpan jumlah cara mencapai langkah (i-2) dan (i-1).
  two_steps_before = 1 → Merepresentasikan ways(1)
  one_step_before = 2  → Merepresentasikan ways(2)

  for i from 3 to n: → Iterasi dari langkah ke-3 hingga langkah ke-n (inklusif).
    current_ways = one_step_before + two_steps_before
     → Jumlah cara untuk mencapai langkah saat ini (i) adalah penjumlahan dari cara mencapai langkah (i-1) dan langkah (i-2).
    two_steps_before = one_step_before → Nilai dari 'one_step_before' yang lama menjadi 'two_steps_before' baru
    one_step_before = current_ways → Nilai 'current_ways' yang baru dihitung menjadi 'one_step_before' yang baru.
  return one_step_before → Setelah loop selesai, 'one_step_before' akan berisi total jumlah cara untuk mencapai langkah 'n'.
```

d. Implementasi

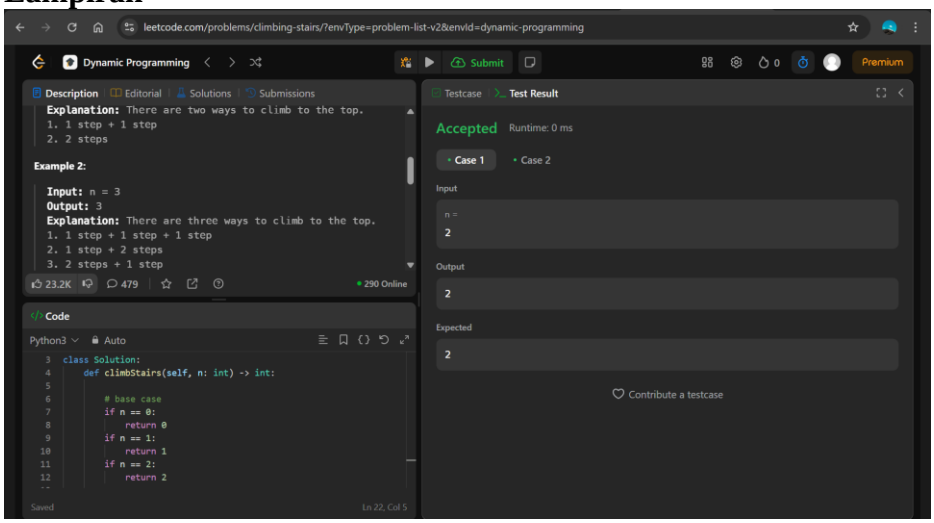
Link Github : Code Soal 2

```python
from typing import List

class Solution:
    def climbStairs(self, n: int) -> int:

        # base case
        if n == 0:
            return 0
        if n == 1:
            return 1
        if n == 2:
            return 2

        two_steps_before = 1
        one_step_before = 2

        for i in range(3, n + 1):
            current_ways = one_step_before + two_steps_before
            two_steps_before = one_step_before
            one_step_before = current_ways
        return one_step_before
```

Kompleksitas :

- Kompleksitas Waktu :
  O(N), di mana N adalah jumlah langkah. Algoritma ini melakukan satu lintasan tunggal (loop) dari 3 hingga N.
- Kompleksitas Ruang :
  O(1), karena hanya menggunakan beberapa variabel tambahan tetap (two_steps_before, one_step_before, current_ways), terlepas dari ukuran input N.

e. Lampiran



Soal 3 : Coin Change (Link Soal 3)

a. Deskripsi Soal

Diberikan sebuah array integer coins yang merepresentasikan koin-koin dengan denominasi yang berbeda, dan sebuah integer amount yang merepresentasikan jumlah uang total. Tugasnya adalah mengembalikan jumlah koin paling sedikit yang dibutuhkan untuk membentuk jumlah uang amount tersebut. Diasumsikan kita memiliki jumlah koin dari setiap jenis yang tidak terbatas. Jika jumlah uang amount tidak dapat dibentuk oleh kombinasi koin apa pun, kembalikan -1.

Contoh Kasus 1 :

Input : coins = [1,2,5], amount = 11
Output : 3
Penjelasan : 11 = 5+5+1 (menggunakan 3 koin)

Contoh Kasus 2 :

Input : coins = [2], amount = 3
Output : -1
Penjelasan : Dengan koin 2, jumlah 3 tidak dapat dibentuk.

Contoh Kasus 3 :

Input : coins = [1], amount = 0
Output : 0
Penjelasan : Untuk jumlah 0, tidak diperlukan koin.

b. Abstraksi Soal

Permasalahan "Coin Change" merupakan problem optimasi klasik yang mencari nilai minimum, sehingga sangat cocok untuk diselesaikan menggunakan pendekatan Dynamic Programming (DP). Analisis dimulai dengan mendefinisikan struktur sub-masalah.

Misalkan dp[i] merepresentasikan jumlah koin minimum yang dibutuhkan untuk membentuk jumlah uang sebesar i. Tujuan utama dari permasalahan ini adalah untuk menemukan nilai `dp[amount]`.

Untuk menentukan nilai `dp[i]`, pertimbangan dilakukan terhadap setiap denominasi koin c yang tersedia dalam array coins. Jika koin c digunakan untuk mencapai jumlah i, maka sisa jumlah yang perlu dibentuk adalah i - c. Jumlah koin total yang dibutuhkan untuk i melalui jalur ini adalah `1 + dp[i − c]` (1 koin untuk c itu sendiri, ditambah koin minimum untuk sisa jumlah i - c).

Karena objective adalah menemukan jumlah koin paling sedikit, maka untuk setiap i, nilai dp[i] ditentukan dengan memilih nilai minimum dari `1 + dp[i − c]` di antara semua koin c yang memungkinkan (`i − c >= 0`).

Formulasi rekurensi yang terbentuk adalah:
`dp[i] = min(1 + dp[i − c])` untuk setiap `c` di `coins` di mana `i − c >= 0`.

Base Cases :

Titik awal bagi rekurensi ini adalah jumlah 0. Untuk membentuk jumlah 0, diperlukan 0 koin. Oleh karena itu`, dp[0] = 0`.

Model Dynamic Programming (Bottom-Up / Tabulation) :

Strategi yang diterapkan adalah pendekatan bottom-up (tabulation). Ini melibatkan pembangunan solusi secara iteratif dari kasus dasar dp[0] hingga dp[amount].

1. Sebuah array dp dengan ukuran amount + 1 diinisialisasi.
2. Semua elemen dp diinisialisasi dengan nilai yang sangat besar (misalnya, amount + 1 atau `float('inf')`). Ini berfungsi sebagai indikator bahwa jumlah tersebut belum dapat dibentuk atau memerlukan jumlah koin yang sangat besar.
3. Nilai dp[0] diatur menjadi 0.

Proses dilanjutkan dengan iterasi dari i = 1 hingga amount. Pada setiap iterasi i:

- Setiap koin c dari array coins dipertimbangkan.
- Apabila i - c merupakan jumlah yang valid (yaitu, `i − c >= 0`) dan `dp[i − c]` bukan nilai "tidak bisa dibentuk" (yaitu, `dp[i − c]` bukan nilai inisialisasi tak terbatas), maka `1 + dp[i − c]` merupakan kandidat jumlah koin untuk i.
- Nilai `dp[i]` kemudian diperbarui dengan mengambil nilai minimum dari semua kandidat yang valid tersebut.

Setelah seluruh iterasi selesai, nilai `dp[amount]` akan mengandung solusi. Jika `dp[amount]` masih sama dengan nilai inisialisasi tak terbatas, ini mengindikasikan bahwa jumlah target tidak dapat dibentuk, sehingga hasil yang dikembalikan adalah -1.

c. Pseudocode

```
function coinChange(coins, amount):
  dp = array of size (amount + 1)
```
→ array 'dp' berukuran (amount + 1) untuk menyimpan jumlah koin minimum bagi setiap jumlah dari 0 hingga 'amount'
```
  for i from 0 to amount:
    dp[i] = amount + 1
```
→ Inisialisasi setiap elemen dengan 'amount + 1' yang merepresentasikan nilai 'infinity', mengindikasikan bahwa jumlah tersebut belum dapat dibentuk.

`dp[0] = 0` → basecase untuk membentuk jumlah 0, dibutuhkan 0 koin

```
  for i from 1 to amount:
    for each coin in coins:
      if i - coin >= 0:
        dp[i] = min(dp[i], 1 + dp[i - coin])
```
→ Iterasi melalui setiap jumlah target 'i' dari 1 hingga 'amount'

→ Untuk setiap jumlah 'i', coba gunakan setiap 'coin' yang tersedia.

→ Verifikasi apakah 'coin' dapat digunakan, yaitu 'i - coin' tidak negatif

→ Nilai 'dp[i]' diperbarui dengan minimum dari nilai yang sudah ada di 'dp[i]', atau 1 (untuk 'coin' saat ini) ditambah jumlah koin minimum untuk 'i - coin'.

→ Periksa : Jika 'dp[amount]' masih bernilai 'amount + 1', ini berarti 'amount' tidak dapat dibentuk

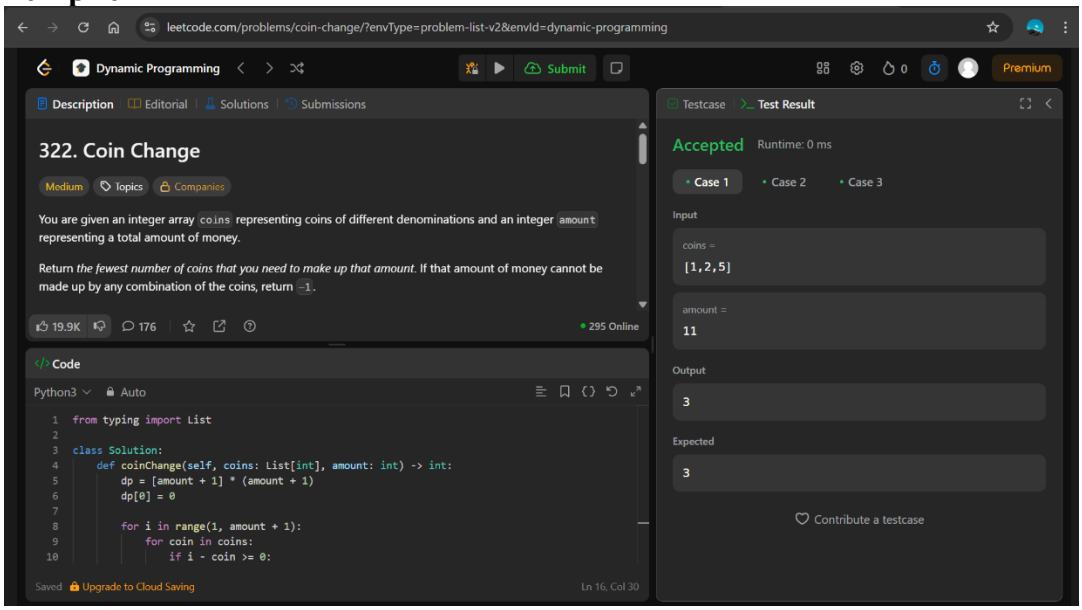
```
  if dp[amount] == amount + 1:
    return -1
  else:
    return dp[amount]
```

d. Implentasi

Link Github : Code Soal 3

```python
from typing import List

class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        dp = [amount + 1] * (amount + 1)
        dp[0] = 0

        for i in range(1, amount + 1):
            for coin in coins:
                if i - coin >= 0:
                    dp[i] = min(dp[i], 1 + dp[i - coin])

        if dp[amount] == amount + 1:
            return -1
        else:
            return dp[amount]
```

e. Lampiran



Soal 4 : Best Time to Buy and Sell Stock with Cooldown (Link Soal 4)

a. Deskripsi Soal

Diberikan sebuah array prices di mana prices[i] adalah harga saham pada hari ke-i. Permasalahan ini bertujuan untuk menemukan keuntungan maksimum yang dapat dicapai dari transaksi saham. Diperbolehkan untuk melakukan transaksi jual-beli berkali-kali dengan batasan berikut:

1. Cooldown : Setelah menjual saham, tidak diperbolehkan membeli saham pada hari berikutnya (ada satu hari "dingin" atau jeda).
2. Tidak Transaksi Bersamaan : Tidak diperbolehkan melakukan banyak transaksi secara bersamaan (saham harus dijual terlebih dahulu sebelum membeli lagi).

Contoh Kasus 1 :

Input : prices = [1,2,3,0,2]
Output : 3
Penjelasan : Urutan transaksi: Beli pada hari 0 (harga 1), Jual pada hari 1 (harga 2) -> Profit 1. Hari 2 adalah cooldown. Beli pada hari 3 (harga 0), Jual pada hari 4 (harga 2) -> Profit 2. Total profit $1 + 2 = 3$.

Contoh Kasus 2 :

Input : prices = [1]
Output : 0
Penjelasan : Hanya ada satu hari, tidak ada transaksi yang bisa dilakukan.

b. Abstraksi Soal

Permasalahan "Best Time to Buy and Sell Stock with Cooldown" adalah kasus yang lebih kompleks dari masalah transaksi saham tunggal karena adanya batasan *cooldown*. Ini adalah tipikal masalah Dynamic Programming karena adanya struktur optimal (keputusan optimal hari ini bergantung pada keputusan optimal hari sebelumnya) dan sub-masalah yang tumpang tindih.

Untuk menyelesaikan masalah ini, status kita pada setiap hari perlu didefinisikan dengan cermat. Pada hari `i`, kita dapat berada dalam salah satu dari tiga status mutually exclusive yang akan memengaruhi keuntungan akumulatif:

1. `hold_profit[i]` : Keuntungan maksimum jika pada akhir hari `i` kita ***memiliki saham***. Ini berarti kita mempertahankan saham yang sudah dibeli sebelumnya, atau baru membeli saham pada hari `i`.
2. `sold_profit[i]` : Keuntungan maksimum jika pada akhir hari `i` kita ***baru saja menjual saham***. Konsekuensinya, pada hari `i+1` kita akan berada dalam kondisi cooldown dan tidak dapat membeli.
3. `rest_profit[i]` : Keuntungan maksimum jika pada akhir hari `i` kita ***tidak memiliki saham dan tidak baru saja menjual***. Ini berarti kita bisa membeli saham pada hari `i+1`. Status ini mencakup hari-hari setelah cooldown selesai, atau hari-hari di mana kita memang tidak memiliki saham dan tidak melakukan transaksi.

Relasi rekursif untuk setiap status pada hari `i` (berdasarkan status pada hari `i-1`) adalah sebagai berikut:

- `hold_profit[i]` ***(Keuntungan jika memiliki saham pada akhir hari `i`) :***
  - Opsi 1 : Kita sudah memiliki saham sejak hari `i-1` dan memilih untuk mempertahankannya. Keuntungan yang didapat adalah `hold_profit[i-1]`.
  - Opsi 2 : Kita membeli saham pada hari `i`. Pembelian ini hanya bisa dilakukan jika pada hari `i-1` kita berada dalam status `rest_profit` (yaitu, tidak memiliki saham dan sudah melewati masa cooldown). Keuntungan yang didapat adalah `rest_profit[i-1] - prices[i]`.
  - Formulasi : `hold_profit[i] = max(hold_profit[i-1], rest_profit[i-1] - prices[i])`
- `sold_profit[i]` ***(Keuntungan jika baru saja menjual saham pada akhir hari `i`) :***
  - Satu-satunya cara untuk mencapai status ini adalah dengan menjual saham yang kita miliki pada hari `i`. Saham ini pasti kita pegang sejak hari `i-1`.
  - Formulasi : `sold_profit[i] = hold_profit[i-1] + prices[i]`
- `rest_profit[i]` ***(Keuntungan jika tidak memiliki saham dan tidak baru saja menjual pada akhir hari `i`) :***
  - Opsi 1 : Saya baru saja menjual saham pada hari i-1. Karena ada cooldown, hari i saya wajib "istirahat" dan tidak bisa membeli. Keuntungan yang didapat adalah sold_profit[i-1].

- Opsi 2 : Saya sudah dalam status `rest_profit` dari hari `i-1` dan memilih untuk tetap istirahat (tidak membeli dan tidak menjual). Keuntungan yang didapat adalah `rest_profit[i-1]`.
- Formulasi: `rest_profit[i] = max(sold_profit[i-1], rest_profit[i-1])`

***Base Case* :** Untuk memulai proses DP, kita perlu inisialisasi status pada hari pertama (indeks 0) :

- `hold_profit[0] = -prices[0]` (Jika kita membeli saham pada hari 0, keuntungan awal adalah negatif harga beli).
- `sold_profit[0] = -float('inf')` (Tidak mungkin menjual saham pada hari pertama, jadi keuntungan dianggap negatif tak terhingga).
- `rest_profit[0] = 0` (Tidak memiliki saham, belum ada keuntungan, dan kita bisa membeli di hari berikutnya jika ada).

Model Dynamic Programming (Bottom-Up dengan Optimalisasi Ruang) :

Karena perhitungan untuk setiap status pada hari `i` hanya bergantung pada status-status dari hari `i-1`, kita dapat mengoptimalkan ruang memori dari $O(N)$ menjadi $O(1)$. Ini dilakukan dengan hanya menyimpan tiga variabel yang mewakili `hold_profit`, `sold_profit`, dan `rest_profit` dari iterasi sebelumnya.

c. Pseudocode

```
function maxProfit(prices):
  if prices is empty or prices.length < 2:
    return 0
```
→ Inisialisasi variabel untuk melacak keuntungan maksimum pada setiap status di akhir hari.
```
  hold_profit = -prices[0]
```
→ Keuntungan jika memiliki saham pada akhir Hari 0
```
  sold_profit = -infinity
```
→ Keuntungan jika baru saja menjual pada akhir Hari 0 (tidak mungkin)
```
  rest_profit = 0
```
→ Keuntungan jika istirahat pada akhir Hari 0 (tidak ada transaksi, bisa beli besok)
```
  for i from 1 to prices.length - 1:
```
→ Iterasi dari Hari 1 hingga hari terakhir (indeks `prices.length - 1`)
```
    current_price = prices[i]
```
→ Simpan nilai-nilai status dari hari sebelumnya (Hari i-1) sebelum diupdate untuk Hari i. Ini krusial karena perhitungan untuk Hari i masih bergantung pada nilai dari Hari i-1.
```
    prev_hold_profit = hold_profit
    prev_sold_profit = sold_profit
    prev_rest_profit = rest_profit

    hold_profit = max(prev_hold_profit, prev_rest_profit - current_price)
```
→ Hitung `hold_profit` untuk hari ini:
Pilihan 1: Lanjutkan memegang saham yang sudah ada. Keuntungan = `prev_hold_profit`.
Pilihan 2: Beli saham pada hari ini. Ini hanya mungkin jika kemarin saya berada dalam status 'rest_profit' (tidak memiliki saham dan cooldown sudah lewat).
```
    sold_profit = prev_hold_profit + current_price
```
→ Satu-satunya cara untuk mencapai status 'sold' adalah dengan menjual saham yang dipegang dari hari sebelumnya.
```
    rest_profit = max(prev_sold_profit, prev_rest_profit)
```
→ Hitung `rest_profit` untuk hari ini:
Pilihan 1: Saya baru saja menjual pada hari sebelumnya (`prev_sold_profit`), sehingga hari ini saya dalam masa cooldown.
Pilihan 2: Saya sudah dalam status 'rest_profit' dari hari sebelumnya (`prev_rest_profit`) dan memilih untuk tetap istirahat.
```
  return max(sold_profit, rest_profit)
```
→ Keuntungan maksimum yang dapat dicapai adalah keuntungan ketika saya tidak memegang saham pada akhir periode (yaitu, status `sold_profit` atau `rest_profit` dari hari terakhir). Saya memilih yang terbesar dari kedua status akhir ini.

d. Implementasi

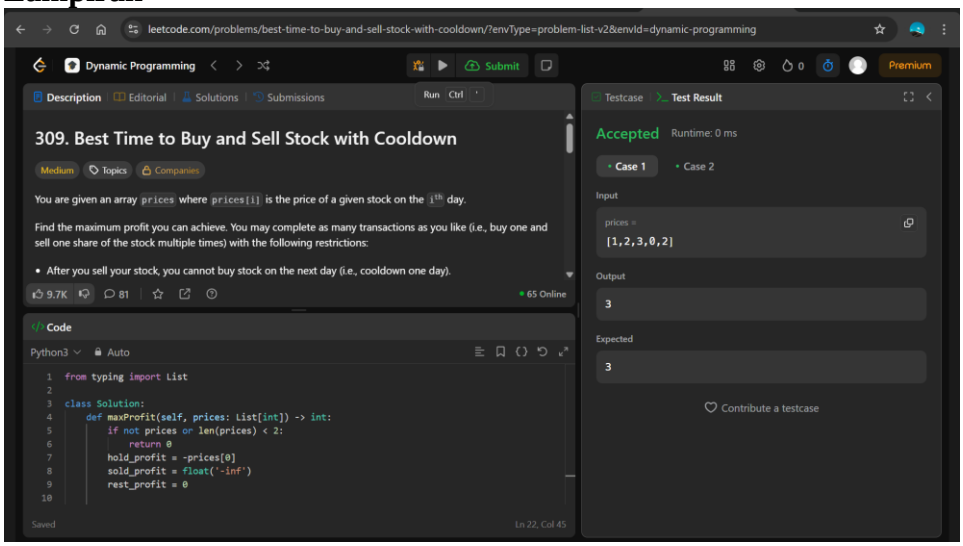
Link Github : Code Soal 4

```python
from typing import List

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        if not prices or len(prices) < 2:
            return 0
        hold_profit = -prices[0]
        sold_profit = float('-inf')
        rest_profit = 0

        for i in range(1, len(prices)):
            current_price = prices[i]

            prev_hold_profit = hold_profit
            prev_sold_profit = sold_profit
            prev_rest_profit = rest_profit

            hold_profit = max(prev_hold_profit, prev_rest_profit - current_price)
            sold_profit = prev_hold_profit + current_price
            rest_profit = max(prev_sold_profit, prev_rest_profit)

        return max(sold_profit, rest_profit)
```

Kompleksitas :

- Kompleksitas Waktu : O(N), di mana N adalah jumlah hari (panjang array `prices`). Algoritma ini melakukan satu lintasan tunggal melalui array `prices`.
- Kompleksitas Ruang **:** O(1), karena hanya menggunakan beberapa variabel tambahan yang jumlahnya tetap (`hold_profit`, `sold_profit`, `rest_profit`, dan variabel `prev_` sementara), terlepas dari ukuran input N.

e. Lampiran



Soal 4 : Best Time to Buy and Sell Stock IV (Link Soal 5)

a. Deskripsi Soal

Diberikan sebuah array `prices` di mana `prices[i]` adalah harga saham pada hari ke-i, dan sebuah integer `k` yang merepresentasikan jumlah transaksi maksimum yang diizinkan. Tugas utama adalah menemukan keuntungan maksimum yang dapat dicapai. Transaksi yang diperbolehkan adalah paling banyak k kali pembelian dan k kali penjualan. Penting untuk diingat bahwa tidak diperbolehkan melakukan banyak transaksi secara bersamaan (saham harus dijual sebelum membeli lagi).

Contoh Kasus 1 :

- Input : `k = 2, prices = [2,4,1]`
- Output : 2
- Penjelasan : Transaksi: Beli pada hari 1 (harga = 2), Jual pada hari 2 (harga = 4). Keuntungan 4−2=2.

Contoh Kasus 2 :

- Input : `k = 2, prices = [3,2,6,5,0,3]`
- Output : 7

- Penjelasan : Transaksi: Beli pada hari 2 (harga = 2), Jual pada hari 3 (harga = 6). Keuntungan 6−2=4. Kemudian beli pada hari 5 (harga = 0), Jual pada hari 6 (harga = 3). Keuntungan 3−0=3. Total keuntungan 4+3=7.

b. **Abstraksi Soal**

"Best Time to Buy and Sell Stock IV" adalah masalah Dynamic Programming (DP) yang kompleks karena batasan `k` transaksi.

***Kasus Khusus* :** Jika `k` sangat besar (lebih dari atau sama dengan separuh jumlah hari), saya bisa melakukan transaksi tak terbatas. Dalam situasi ini, saya cukup menjumlahkan setiap kenaikan harga yang ada. Ini adalah optimasi penting untuk menghindari TLE (Time Limit Exceeded).

Model DP Umum : Untuk kasus `k` yang tidak terlalu besar, saya menggunakan pendekatan DP bottom-up. Saya melacak keuntungan maksimum untuk setiap hari `i` dan setiap jumlah transaksi `j` yang telah dilakukan, dalam dua status:

1. `dp_buy[j]`: Keuntungan maksimum jika saya ***memiliki saham*** setelah menyelesaikan `j-1` transaksi jual-beli dan melakukan pembelian ke-`j`.
2. `dp_sell[j]`: Keuntungan maksimum jika saya ***tidak memiliki saham*** setelah menyelesaikan tepat `j` transaksi jual-beli.

Saya menginisialisasi `dp_buy[j]` dengan `-infinity` (karena membeli berarti keuntungan berkurang) dan `dp_sell[j]` dengan `0`. `dp_sell[0]` juga saya set 0, yang mewakili keuntungan tanpa transaksi.

Kemudian, saya mengiterasi setiap hari (`price`) dan setiap jumlah transaksi `j` (dari `k` turun ke `1`). Perhitungan saya adalah:

- `dp_sell[j]` (baru) : `max(dp_sell[j]` (lama), `dp_buy[j]` (lama) `+ price)`
  Artinya, keuntungan hari ini setelah `j` transaksi jual adalah maksimum dari mempertahankan status `dp_sell[j]` sebelumnya, atau menjual saham yang saya miliki (`dp_buy[j]`) hari ini.
- `dp_buy[j]` (baru) : `max(dp_buy[j]` (lama), `dp_sell[j-1]` (lama) `- price)`
  Artinya, keuntungan hari ini setelah `j` transaksi beli adalah maksimum dari mempertahankan status `dp_buy[j]` sebelumnya, atau membeli saham (setelah `j-1` transaksi selesai dengan keuntungan `dp_sell[j-1]`) hari ini.

Setelah semua hari diproses, keuntungan maksimum akan menjadi nilai terbesar di antara semua elemen `dp_sell`.

c. **Pseudocode**

```
function maxProfit(k, prices):
  n = prices.length

  if k >= n / 2: → transaksi tak terbatas jika k sangat besar
    max_profit = 0
    for i from 1 to n - 1:
      if prices[i] > prices[i-1]:
        max_profit += prices[i] - prices[i-1]
    return max_profit

  dp_buy = array of size (k + 1) → Inisialisasi array DP profit jika memiliki saham setelah transaksi ke-j
  dp_sell = array of size (k + 1) → Inisialisasi array DP profit jika tidak memiliki saham setelah transaksi ke-j

  for j from 0 to k:       → Set nilai awal
    dp_buy[j] = -infinity
    dp_sell[j] = 0

  for price in prices:       → Iterasi setiap harga saham
    for j from k down to 1: → Iterasi jumlah transaksi mundur
      dp_sell[j] = max(dp_sell[j], dp_buy[j] + price)
      → Update dp_sell[j] : max profit jika saya jual saham ke-j hari ini
      dp_buy[j] = max(dp_buy[j], dp_sell[j-1] - price)
      → Update dp_buy[j] : max profit jika saya beli saham ke-j hari ini

  return max(dp_sell) // max dari dp_sell[0] sampai dp_sell[k]
  Hasil akhir adalah keuntungan terbesar dari semua kemungkinan jumlah transaksi yang telah diselesaikan
```

d. **Implementasi**

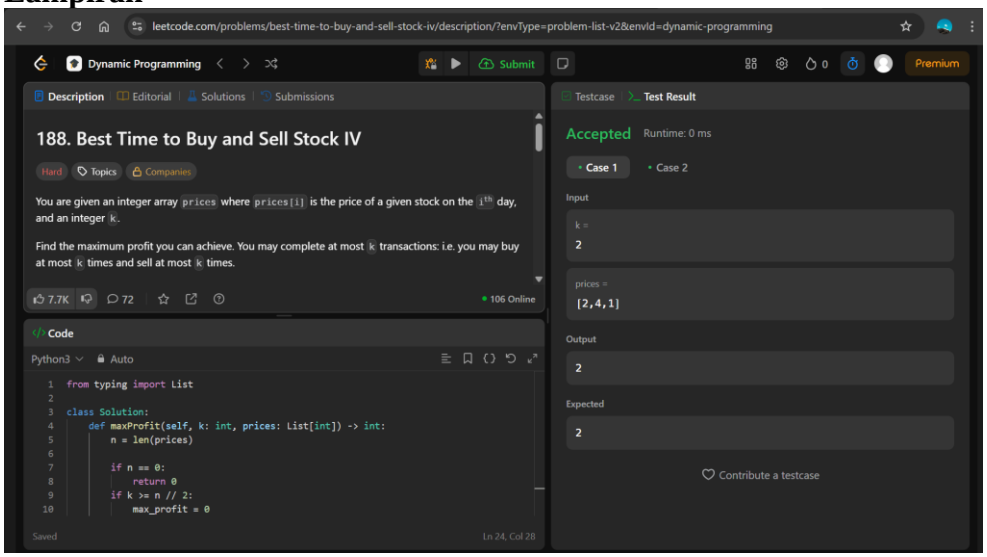
Link Github : [Code Soal 5]

```python
from typing import List

class Solution:
    def maxProfit(self, k: int, prices: List[int]) -> int:
        n = len(prices)

        if n == 0:
            return 0
        if k >= n // 2:
            max_profit = 0
            for i in range(1, n):
                if prices[i] > prices[i-1]:
                    max_profit += prices[i] - prices[i-1]
            return max_profit

        dp_buy = [-float('inf')] * (k + 1)
        dp_sell = [0] * (k + 1)

        for price in prices:
            for j in range(k, 0, -1):
                dp_sell[j] = max(dp_sell[j], dp_buy[j] + price)
                dp_buy[j] = max(dp_buy[j], dp_sell[j-1] - price)

        return max(dp_sell)
```

Kompleksitas :

- Kompleksitas Waktu : O(N×K), di mana N adalah jumlah hari (panjang array `prices`) dan K adalah jumlah transaksi maksimum. Ini karena terdapat dua loop bersarang: loop luar berjalan N kali (untuk setiap hari) dan loop dalam berjalan K kali (untuk setiap jumlah transaksi).
- Kompleksitas Ruang : O(K), karena saya menggunakan dua array `dp_buy` dan `dp_sell` yang ukurannya sebanding dengan K.

e. **Lampiran**



Soal 6 : Dungeon Game [(Link Soal 6)]

a. **Deskripsi Soal**

Seorang kesatria harus menyelamatkan putri yang dipenjara di sudut kanan bawah sebuah dungeon. Dungeon ini terdiri dari ruangan m×n yang tersusun dalam grid 2D. Kesatria memulai perjalanan dari ruangan paling kiri atas dengan sejumlah poin kesehatan (HP) positif. Jika HP-nya mencapai 0 atau kurang, kesatria akan segera mati.

Beberapa ruangan dijaga oleh iblis (direpresentasikan dengan bilangan bulat negatif), sehingga HP kesatria berkurang saat memasuki ruangan ini. Ruangan lain bisa kosong (0) atau berisi bola sihir yang meningkatkan HP kesatria (bilangan bulat positif).

Untuk mencapai putri secepat mungkin, kesatria hanya diperbolehkan bergerak ke kanan atau ke bawah pada setiap langkah.

Tugasnya adalah mengembalikan jumlah HP awal minimum yang harus dimiliki kesatria agar ia dapat menyelamatkan putri. Perlu diperhatikan bahwa setiap ruangan, termasuk ruangan awal dan ruangan tempat putri dipenjara, dapat berisi ancaman atau power-up.

Contoh Kasus 1 :

- Input : `dungeon = [[-2,-3,3],[-5,-10,1],[10,30,-5]]`
- Output : `7`
- Penjelasan : HP awal kesatria harus minimal 7. Jalur optimal adalah KANAN -> KANAN -> BAWAH -> BAWAH.
  - Mulai dengan HP 7.
  - Masuk `(-2)`: HP 7+(−2)=5.
  - Masuk `(-3)`: HP 5+(−3)=2.
  - Masuk `(1)`: HP 2+1=3.
  - Masuk `(-5)`: HP 3+(−5)=−2. (Mati!)

  Ternyata penjelasan saya salah. Jalur optimal harus yang menghasilkan HP minimum di akhir. Mari kita lihat lagi contoh LeetCode: Optimal path: `RIGHT -> RIGHT -> DOWN -> DOWN`. Start at (0,0) with 7 HP. (0,0): -2. HP becomes 7 + (-2) = 5. (0,1): -3. HP becomes 5 + (-3) = 2. (0,2): 3. HP becomes 2 + 3 = 5. (1,2): 1. HP becomes 5 + 1 = 6. (2,2): -5. HP becomes 6 + (-5) = 1. Kesatria selamat dengan 1 HP di akhir.

Contoh Kasus 2 :

- Input : `dungeon = [[0]]`
- Output : `1`
- Penjelasan : Ruangan awal kosong (0). Untuk bertahan hidup, HP harus minimal 1 saat keluar dari ruangan ini. Jadi, HP awal harus 1

b. Abstraksi Soal

Permasalahan "Dungeon Game" adalah masalah Dynamic Programming (DP) yang unik karena pendekatannya paling efektif dilakukan secara mundur (backward), yaitu dari tujuan (putri) ke titik awal (kesatria). Jika mencoba menyelesaikannya dari awal ke akhir, akan sulit menentukan HP awal yang dibutuhkan karena HP bisa berubah (bertambah atau berkurang) di setiap ruangan.

Alasan Pendekatan Mundur: Ketika bergerak mundur dari putri ke kesatria, pada setiap sel `(r, c)`, kita dapat menentukan HP minimum yang harus dimiliki kesatria *saat memasuki* sel tersebut agar ia dapat bertahan hidup melalui sel itu dan mencapai putri dari sana. Tujuan utama adalah memastikan HP tidak pernah jatuh di bawah 1.

Definisi `dp[r][c]`: HP minimum yang harus dimiliki kesatria saat memasuki sel `(r, c)` agar ia dapat mencapai putri dengan selamat (memiliki minimal 1 HP di setiap langkah, termasuk saat tiba di sel putri).

Relasi Rekursif (Backward DP):

Untuk setiap sel `(r, c)`, kesatria dapat bergerak menuju putri melalui sel di bawahnya `(r+1, c)` atau sel di kanannya `(r, c+1)`. Untuk mencapai putri, kesatria harus memilih jalur dari `(r, c)` yang membutuhkan HP minimum saat keluar dari `(r, c)`.

Jadi, untuk menghitung `dp[r][c]`, kita harus mempertimbangkan:

1. HP minimum yang dibutuhkan untuk mencapai putri dari `(r+1, c)`: `dp[r+1][c]`.
2. HP minimum yang dibutuhkan untuk mencapai putri dari `(r, c+1)`: `dp[r][c+1]`.

Kesatria akan memilih jalur yang membutuhkan HP minimum untuk melanjutkan perjalanan setelah melewati sel `(r, c)`. Misalkan nilai minimum dari kedua opsi ini adalah `min_hp_from_next_cell = min(dp[r+1][c], dp[r][c+1])`.

HP yang harus dimiliki saat masuk ke sel `(r, c)` adalah `min_hp_from_next_cell - dungeon[r][c]`. Namun, HP tidak boleh kurang dari 1. Jadi, jika hasil perhitungan ini kurang dari atau sama dengan 0, itu berarti kita hanya perlu 1 HP saat masuk ke `(r,c)` agar bisa bertahan hidup melewati `dungeon[r][c]` dan masih memiliki minimal 1 HP untuk melanjutkan.

Formulasi : `dp[r][c] = max(1, min(dp[r+1][c], dp[r][c+1]) - dungeon[r][c])`

Base Cases - Pengisian Tabel DP dari kanan bawah ke kiri atas:

1. Sel Putri **`(m-1, n-1)`:**
   - Pada sel ini, kesatria harus tiba dengan HP yang cukup agar setelah efek `dungeon[m-1][n-1]`, HP-nya minimal 1.

- Formulasi : `dp[m-1][n-1] = max(1, 1 - dungeon[m-1][n-1])`.
  - Contoh : Jika `dungeon[m-1][n-1] = -5`, maka `max(1, 1 - (-5)) = max(1, 6) = 6`. Artinya, harus masuk dengan minimal 6 HP agar keluar dengan 1 HP.
  - Contoh : Jika `dungeon[m-1][n-1] = 3`, maka `max(1, 1 - 3) = max(1, -2) = 1`. Artinya, cukup masuk dengan minimal 1 HP karena HP akan bertambah.
2. Baris Terakhir **(`r = m-1, c < n-1`):**
   - Dari sel di baris terakhir, kesatria hanya bisa bergerak ke kanan.
   - Formulasi: `dp[m-1][c] = max(1, dp[m-1][c+1] - dungeon[m-1][c])`
3. Kolom Terakhir **(`c = n-1, r < m-1`):**
   - Dari sel di kolom terakhir, kesatria hanya bisa bergerak ke bawah.
   - Formulasi: `dp[r][n-1] = max(1, dp[r+1][n-1] - dungeon[r][n-1])`

Model Dynamic Programming (Bottom-Up - dari kanan bawah ke kiri atas):

1. Buat tabel DP `dp` dengan ukuran yang sama dengan `dungeon` (`m x n`).
2. Isi kasus dasar: `dp[m-1][n-1]`.
3. Isi baris terakhir (mulai dari `c = n-2` mundur ke `0`).
4. Isi kolom terakhir (mulai dari `r = m-2` mundur ke `0`).
5. Iterasi melalui sisa sel-sel di grid, mulai dari `r = m-2` mundur ke `0`, dan `c = n-2` mundur ke `0`. Untuk setiap sel `(r, c)`, terapkan formulasi umum: `dp[r][c] = max(1, min(dp[r+1][c], dp[r][c+1]) - dungeon[r][c])`

Hasil akhir adalah `dp[0][0]`, yaitu HP minimum yang dibutuhkan saat memasuki ruangan awal

c. Pseudocode

```
function calculateMinimumHP(dungeon):
  rows = dungeon.length
  cols = dungeon[0].length
  → dp[r][c] akan menyimpan HP minimum yang dibutuhkan saat memasuki sel (r,c) agar dapat mencapai putri dengan selamat.
  dp = 2D array of size rows x cols
  → base case : HP minimum yang dibutuhkan saat masuk sel putri
  dp[rows-1][cols-1] = max(1, 1 - dungeon[rows-1][cols-1])
  → Isi Baris Terakhir (dari kanan ke kiri, kecuali sel putri
  for c from cols-2 down to 0:
    dp[rows-1][c] = max(1, dp[rows-1][c+1] - dungeon[rows-1][c])
  → Isi Kolom Terakhir (dari bawah ke atas, kecuali sel putri)
  for r from rows-2 down to 0:
    dp[r][cols-1] = max(1, dp[r+1][cols-1] - dungeon[r][cols-1])
  → Isi Sel-sel Lainnya (dari kanan bawah ke kiri atas)
  for r from rows-2 down to 0:
    for c from cols-2 down to 0:
      → HP minimum yang dibutuhkan dari sel berikutnya (kanan atau bawah)
      min_hp_from_next_cell = min(dp[r+1][c], dp[r][c+1])
      → HP minimum yang dibutuhkan dari sel berikutnya (kanan atau bawah)
      dp[r][c] = max(1, min_hp_from_next_cell - dungeon[r][c])
  → Hasil akhir adalah HP minimum yang dibutuhkan saat masuk sel (0,0)
  return dp[0][0]
```

d. Implementasi

Link Github : [Code Soal 6]

```python
from typing import List

class Solution:
    def calculateMinimumHP(self, dungeon: List[List[int]]) -> int:
        rows = len(dungeon)
        cols = len(dungeon[0])

        dp = [[0] * cols for _ in range(rows)]
        dp[rows - 1][cols - 1] = max(1, 1 - dungeon[rows - 1][cols - 1])

        for c in range(cols - 2, -1, -1):
            dp[rows - 1][c] = max(1, dp[rows - 1][c + 1] - dungeon[rows - 1][c])

        for r in range(rows - 2, -1, -1):
            dp[r][cols - 1] = max(1, dp[r + 1][cols - 1] - dungeon[r][cols - 1])

        for r in range(rows - 2, -1, -1):
            for c in range(cols - 2, -1, -1):
                min_hp_from_next_cell = min(dp[r + 1][c], dp[r][c + 1])
                dp[r][c] = max(1, min_hp_from_next_cell - dungeon[r][c])

        return dp[0][0]
```

Kompleksitas:

- Kompleksitas Waktu : O(M×N), di mana M adalah jumlah baris dan N adalah jumlah kolom dalam `dungeon`. Setiap sel di tabel DP dihitung satu kali.
- Kompleksitas Ruang : O(M×N), karena saya menggunakan tabel DP 2D dengan ukuran yang sama dengan `dungeon` untuk menyimpan hasil sub-masalah.

e. Lampiran