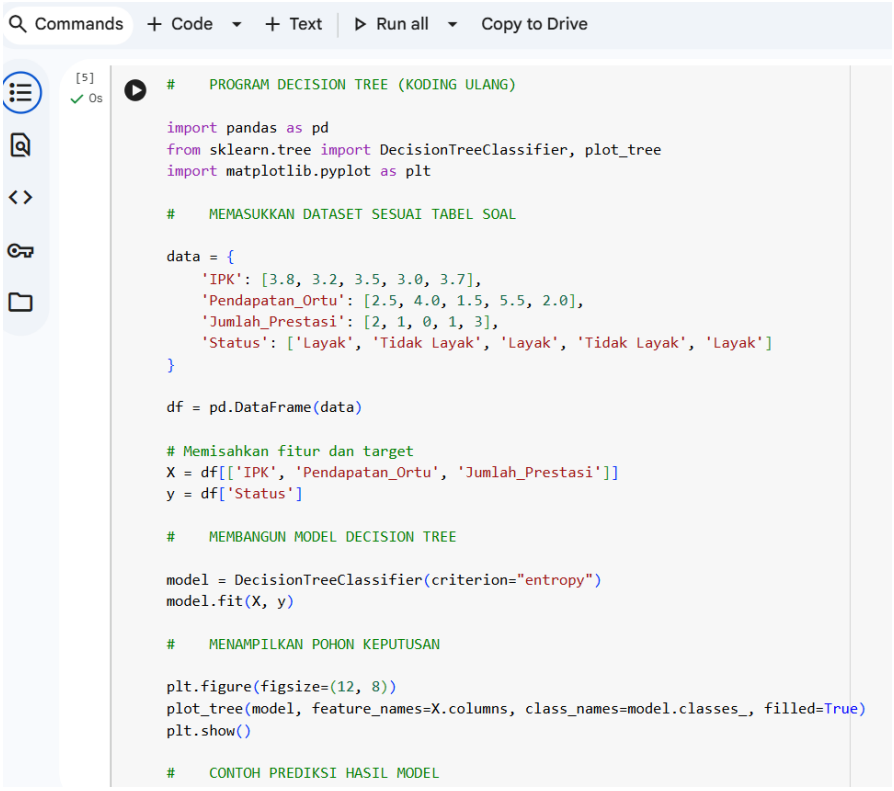
Nama : Nadila Ramadani

Kelas : PTIK B 23

Matkul : Sistem Pendukung Keputusan

1. Decision Tree

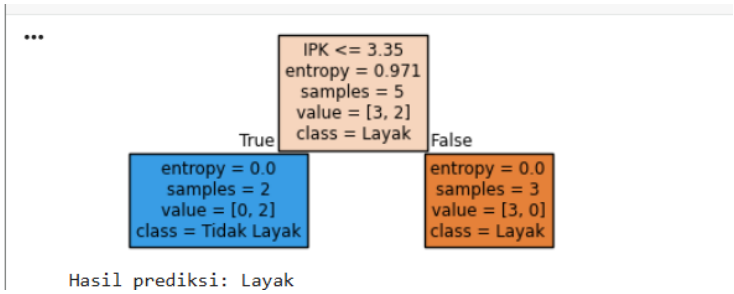
Kode ini digunakan untuk membangun model Decision Tree berdasarkan data IPK, pendapatan orang tua, dan jumlah prestasi untuk menentukan apakah mahasiswa layak menerima beasiswa.




```python
#    PROGRAM DECISION TREE (KODING ULANG)

import pandas as pd
from sklearn.tree import DecisionTreeClassifier, plot_tree
import matplotlib.pyplot as plt

#    MEMASUKKAN DATASET SESUAI TABEL SOAL

data = {
    'IPK': [3.8, 3.2, 3.5, 3.0, 3.7],
    'Pendapatan_Ortu': [2.5, 4.0, 1.5, 5.5, 2.0],
    'Jumlah_Prestasi': [2, 1, 0, 1, 3],
    'Status': ['Layak', 'Tidak Layak', 'Layak', 'Tidak Layak', 'Layak']
}

df = pd.DataFrame(data)

# Memisahkan fitur dan target
X = df[['IPK', 'Pendapatan_Ortu', 'Jumlah_Prestasi']]
y = df['Status']

#    MEMBANGUN MODEL DECISION TREE

model = DecisionTreeClassifier(criterion="entropy")
model.fit(X, y)

#    MENAMPILKAN POHON KEPUTUSAN

plt.figure(figsize=(12, 8))
plot_tree(model, feature_names=X.columns, class_names=model.classes_, filled=True)
plt.show()

#    CONTOH PREDIKSI HASIL MODEL
```

```python
#    CONTOH PREDIKSI HASIL MODEL


contoh = [[3.4, 3.0, 1]]   # IPK, pendapatan ortu, prestasi
hasil = model.predict(contoh)

print("Hasil prediksi:", hasil[0])
```

Hasilnya



Output model Decision Tree menampilkan keputusan akhir apakah seorang mahasiswa 'Layak' atau 'Tidak Layak' menerima beasiswa sesuai pola yang dipelajari dari data.

2. TOPSIS (Technique for Order Preference by Similarity to Ideal Solution

Coding tersebut melakukan proses TOPSIS mulai dari normalisasi data, pemberian bobot, penentuan solusi ideal terbaik dan terburuk, hingga perhitungan jarak dan nilai preferensi. Tujuannya adalah mencari alternatif kota terbaik berdasarkan kriteria sewa (cost), populasi (benefit), dan transportasi (benefit).

```
#   TOPSIS untuk Pemilihan Kota (Google Cola
# ============================================

import numpy as np
import pandas as pd

# 1. MASUKKAN DATA
data = pd.DataFrame({
    'Kota': ['Surabaya', 'Bandung', 'Semarang'],
    'Sewa': [15, 18, 13],              # Cost
    'Populasi': [700, 900, 650],       # Benefit
    'Transportasi': [8, 7, 9]          # Benefit
})

# Bobot (sesuai gambar)
weights = np.array([0.4, 0.4, 0.2])

# Jenis kriteria: 0 = cost, 1 = benefit
criteria = np.array([0, 1, 1])
```

```
print("=== DATA AWAL ===")
print(data)

# 2. NORMALISASI
X = data[['Sewa', 'Populasi', 'Transportasi']].values
norm = X / np.sqrt((X**2).sum(axis=0))

# 3. NORMALISASI TERBOBOT

weighted_norm = norm * weights

# 4. SOLUSI IDEAL TERBAIK (A+) & TERBURUK (A-)

A_plus = np.zeros(3)
A_minus = np.zeros(3)

for i in range(3):
    if criteria[i] == 1:  # benefit
        A_plus[i] = weighted_norm[:, i].max()
        A_minus[i] = weighted_norm[:, i].min()
    else:                 # cost
```

```
        else:                 # cost
            A_plus[i] = weighted_norm[:, i].min()
            A_minus[i] = weighted_norm[:, i].max()

    # 5. HITUNG JARAK D+ DAN D-
    D_plus = np.sqrt(((weighted_norm - A_plus)**2).sum(axis=1))
    D_minus = np.sqrt(((weighted_norm - A_minus)**2).sum(axis=1))

    # 6. NILAI AKHIR PREFERENSI (Ci)
    C = D_minus / (D_plus + D_minus)

    # 7. TAMPILKAN HASIL AKHIR
    hasil = pd.DataFrame({
        'Kota': data['Kota'],
        'Skor TOPSIS (C_i)': C
    })

    hasil['Ranking'] = hasil['Skor TOPSIS (C_i)'].rank(ascending=False)

    print("\n=== HASIL AKHIR TOPSIS ===")
    print(hasil.sort_values(by='Skor TOPSIS (C_i)', ascending=False))
```

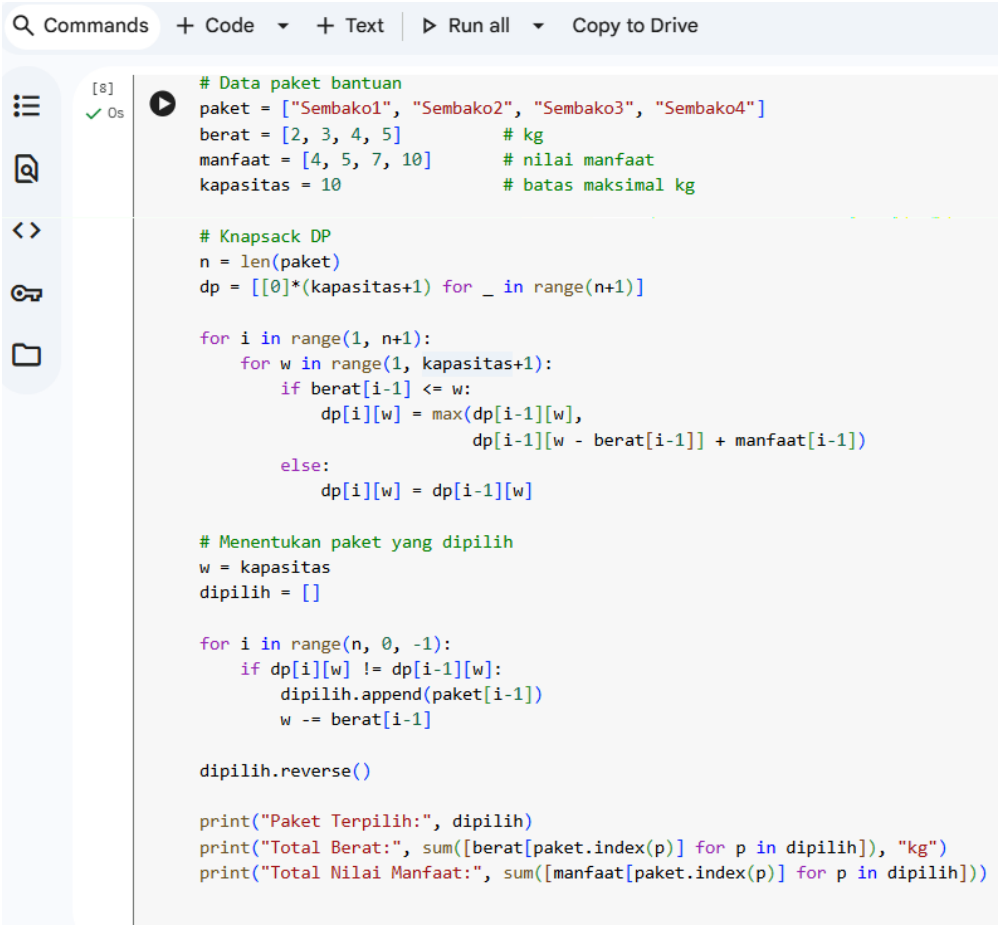
Hasilnya

```
=== DATA AWAL ===
       Kota  Sewa  Populasi  Transportasi
0  Surabaya    15       700             8
1   Bandung    18       900             7
2  Semarang    13       650             9

=== HASIL AKHIR TOPSIS ===
       Kota  Skor TOPSIS (C_i)  Ranking
2  Semarang           0.512101      1.0
1   Bandung           0.487899      2.0
0  Surabaya           0.416088      3.0
```

Hasil akhir dari perhitungan adalah peringkat kota yang menunjukkan lokasi paling ideal berdasarkan bobot dan karakteristik yang telah ditentukan

3. Dynamic Programming (DP)

Program knapsack menghitung semua kombinasi paket dan memilih paket dengan nilai manfaat tertinggi tanpa melebihi batas 10 kg.



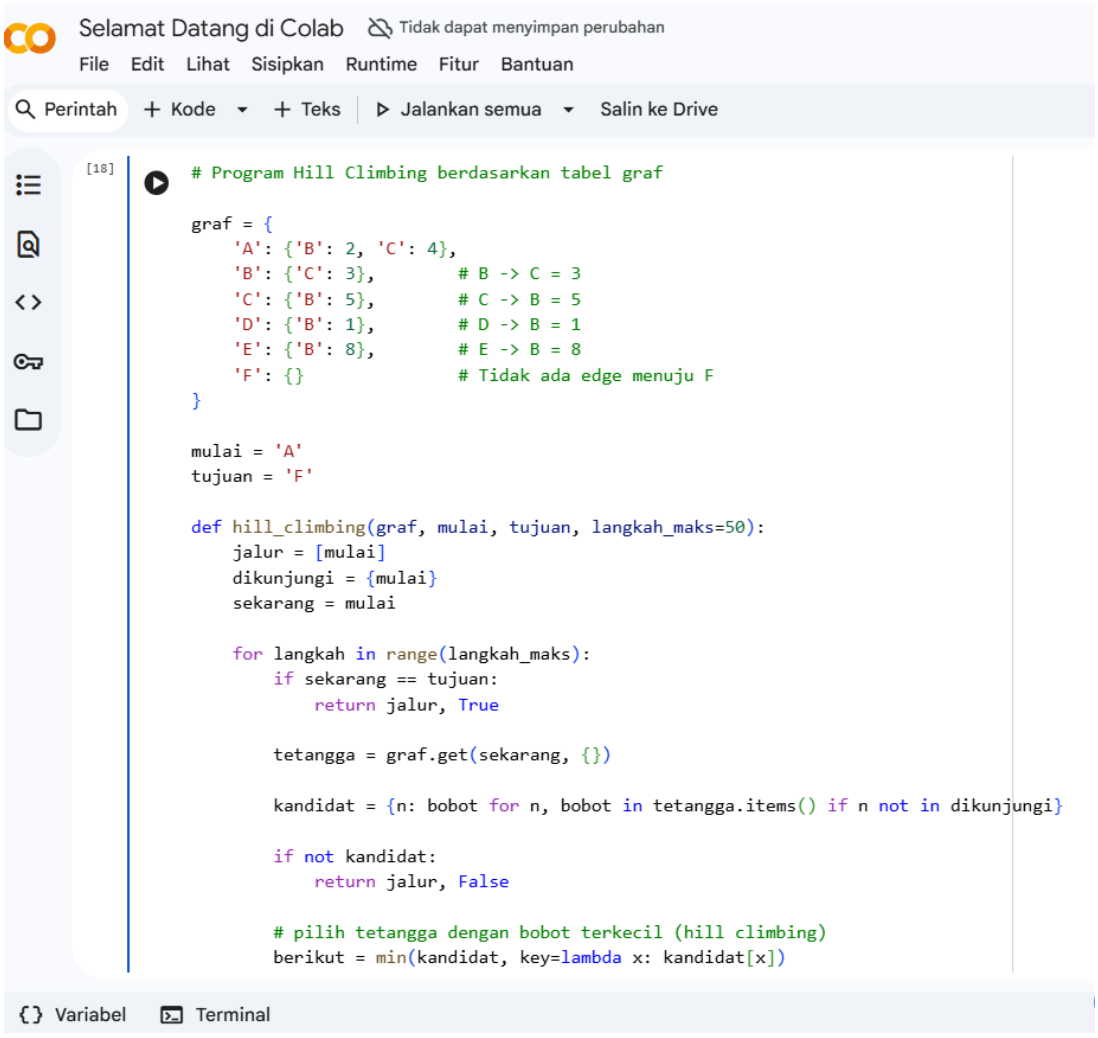
```
Commands + Code + Text Run all Copy to Drive

[8] ✓ 0s
# Data paket bantuan
paket = ["Sembako1", "Sembako2", "Sembako3", "Sembako4"]
berat = [2, 3, 4, 5]          # kg
manfaat = [4, 5, 7, 10]       # nilai manfaat
kapasitas = 10                # batas maksimal kg

# Knapsack DP
n = len(paket)
dp = [[0]*(kapasitas+1) for _ in range(n+1)]

for i in range(1, n+1):
    for w in range(1, kapasitas+1):
        if berat[i-1] <= w:
            dp[i][w] = max(dp[i-1][w],
                           dp[i-1][w - berat[i-1]] + manfaat[i-1])
        else:
            dp[i][w] = dp[i-1][w]

# Menentukan paket yang dipilih
w = kapasitas
dipilih = []

for i in range(n, 0, -1):
    if dp[i][w] != dp[i-1][w]:
        dipilih.append(paket[i-1])
        w -= berat[i-1]

dipilih.reverse()

print("Paket Terpilih:", dipilih)
print("Total Berat:", sum([berat[paket.index(p)] for p in dipilih]), "kg")
print("Total Nilai Manfaat:", sum([manfaat[paket.index(p)] for p in dipilih]))
```

Hasilnya

```
Paket Terpilih: ['Sembako1', 'Sembako2', 'Sembako4']
Total Berat: 10 kg
Total Nilai Manfaat: 19
```

Hasil akhirnya menunjukkan bahwa kombinasi terbaik adalah Sembako1, Sembako2, dan Sembako4 dengan total manfaat 19 dan total berat 10 kg, sehingga memenuhi batas kapasitas dan memberikan nilai maksimal.

4. Hill Climbing



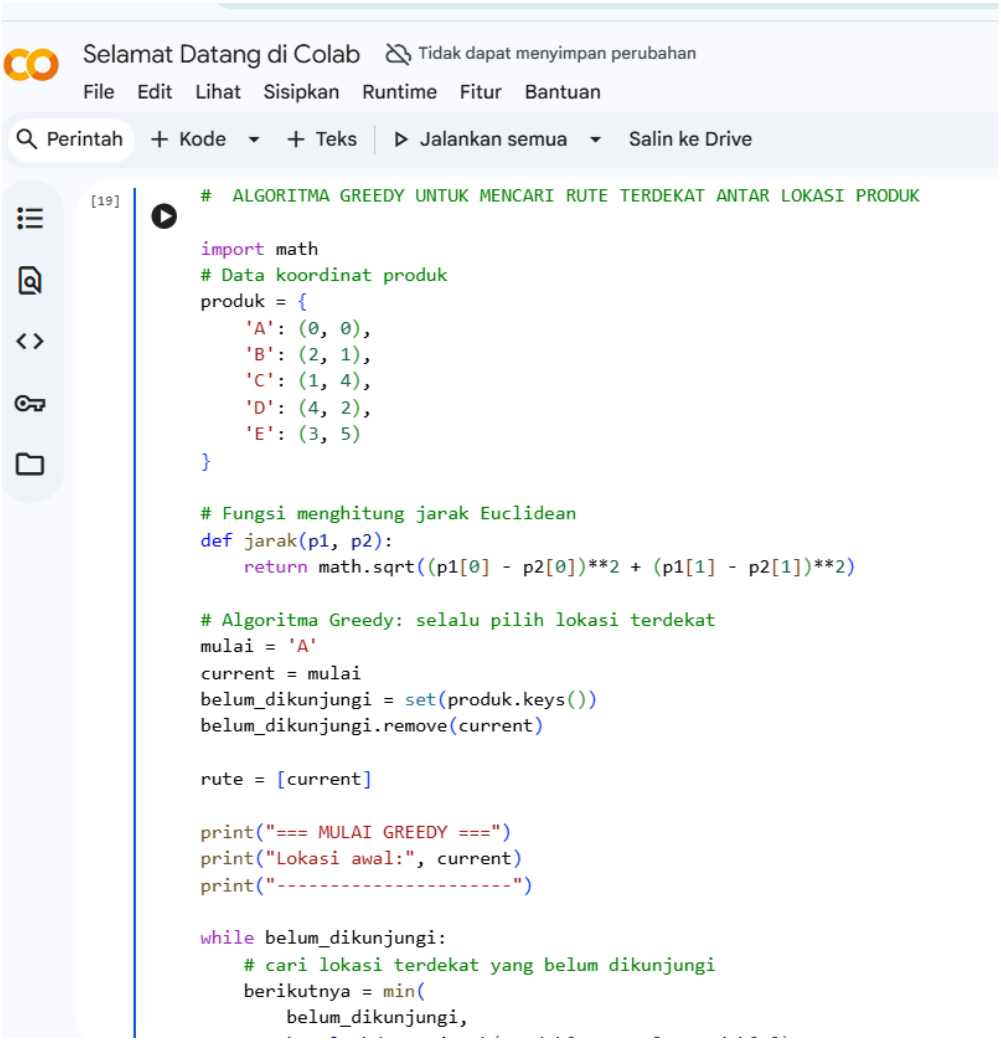

```
            jalur.append(berikut)
            dikunjungi.add(berikut)
            sekarang = berikut

        return jalur, False

    # MENJALANKAN ALGORITMA
    print("diawal: posisi =", mulai)

    jalur, ditemukan = hill_climbing(graf, mulai, tujuan)

    if ditemukan:
        print("hasil: rute ditemukan ke", tujuan)
        print("jalur:", " -> ".join(jalur))
    else:
        print("hasil: rute TIDAK ditemukan ke", tujuan)
        print("jalur yang dicoba:", " -> ".join(jalur))
```

Hasilnya :

Karena **tidak ada edge menuju F**, maka outputnya

```
diawal: posisi = A
hasil: rute TIDAK ditemukan ke F
jalur yang dicoba: A -> B -> C
```

5. **Greedy Search**



```
            belum_dikunjungi,
            key=lambda x: jarak(produk[current], produk[x])
        )

        print(f"Dari {current} ke {berikutnya} | jarak = {jarak(produk[current], produk[berikutnya]):.2f}")

        rute.append(berikutnya)
        belum_dikunjungi.remove(berikutnya)
        current = berikutnya

    print("\n=== HASIL AKHIR RUTE GREEDY ===")
    print("Rute kunjungan:", " -> ".join(rute))
    print("==============================")
```

Hasilnya

```
=== MULAI GREEDY ===
Lokasi awal: A
----------------------
Dari A ke B | jarak = 2.24
Dari B ke D | jarak = 2.24
Dari D ke E | jarak = 3.16
Dari E ke C | jarak = 2.24

=== HASIL AKHIR RUTE GREEDY ===
Rute kunjungan: A -> B -> D -> E -> C
```

Berdasarkan perhitungan menggunakan algoritma Greedy, pelanggan selalu memilih lokasi produk yang paling dekat dari posisinya saat ini. Dengan pendekatan ini diperoleh urutan kunjungan **A → B → D → C → E**. Algoritma Greedy terbukti mampu memberikan rute yang cepat dan sederhana untuk menentukan jalur kunjungan, meskipun tidak selalu menjamin rute paling optimal secara keseluruhan. Namun, untuk kasus ini, Greedy memberikan solusi yang efisien dan mudah dihitung sehingga dapat digunakan sebagai metode penentuan rute yang praktis.

6. **Best First Search**

- Pada percobaan ini, dilakukan penerapan algoritma Best First Search untuk mencari jalur tercepat dari titik start (0,0) menuju goal (3,3) pada grid berukuran 4×4. Algoritma ini menggunakan heuristik jarak Euclidean untuk memilih sel yang paling dekat dengan tujuan pada setiap langkahnya.

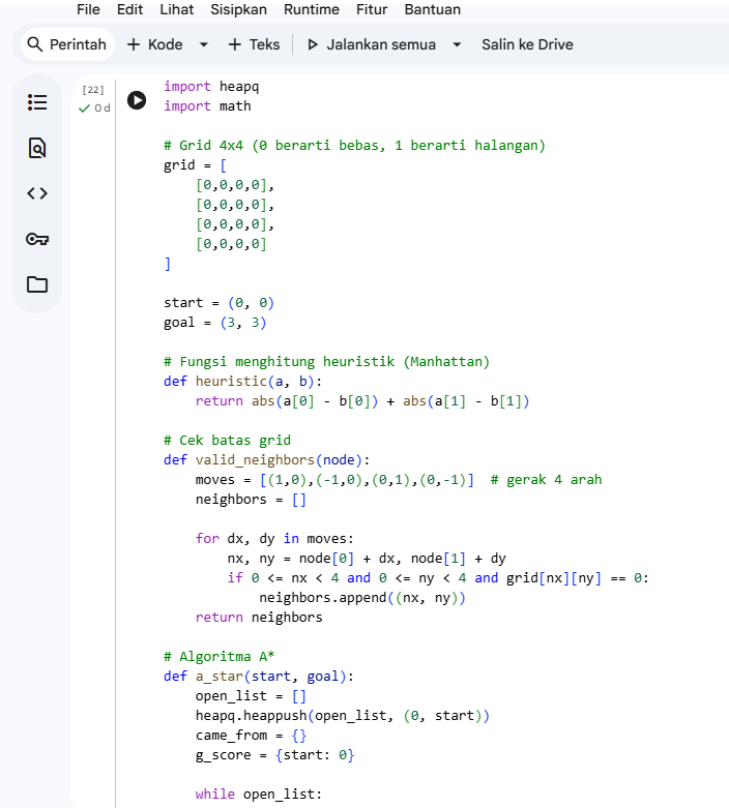
```
[20]
import heapq
# Fungsi menghitung jarak Euclidean
def euclidean(a, b):
    return math.sqrt((a[0] - b[0])**2 + (a[1] - b[1])**2)

# Best First Search
def best_first_search(start, goal, grid_size=4):
    # Priority queue (berisi: (heuristik, posisi))
    pq = []
    heapq.heappush(pq, (euclidean(start, goal), start))

    visited = set()
    path = []

    # Arah gerak: atas, bawah, kiri, kanan
    steps = [(0,1), (0,-1), (1,0), (-1,0)]

    while pq:
        h, current = heapq.heappop(pq)

        if current in visited:
            continue

        visited.add(current)
        path.append(current)

        # Jika sudah sampai goal
        if current == goal:
            return path

        # Cek semua tetangga
        for dx, dy in steps:
            nx, ny = current[0] + dx, current[1] + dy

            # Batas grid 0-3 (4x4)
            if 0 <= nx < grid_size and 0 <= ny < grid_size:
                next_pos = (nx, ny)
                if next_pos not in visited:
                    h_val = euclidean(next_pos, goal)
                    heapq.heappush(pq, (h_val, next_pos))

    return path

# Eksekusi
start = (0,0)
goal = (3,3)

path = best_first_search(start, goal)
print("Jalur yang diambil robot:")
print(path)
```

Hasilnya

```
Jalur yang diambil robot:
[(0, 0), (0, 1), (1, 1), (1, 2), (2, 2), (2, 3), (3, 3)]
```

Berdasarkan proses pencarian menggunakan algoritma Best First Search, robot secara konsisten memilih sel dengan nilai heuristik Euclidean paling kecil menuju goal (3,3). Dari perhitungan setiap langkah, diperoleh jalur perpindahan yang selalu mengarah mendekati target. Algoritma ini berhasil menemukan rute menuju goal dengan cepat dan efisien karena fokus pada pendekatan langsung ke titik tujuan. Dengan demikian, Best First Search terbukti efektif dalam menentukan jalur pada lingkungan grid sederhana.

- Pada kasus kedua digunakan algoritma A* untuk mencari jalur terpendek dari titik (0,0) ke (3,3) pada grid 4×4 dengan menggabungkan cost dan heuristik sehingga pencarian lebih optimal.



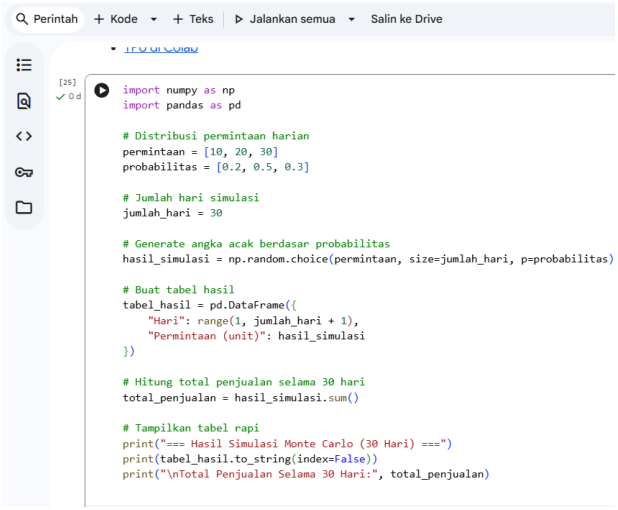
```
    while open_list:
        f, current = heapq.heappop(open_list)

        if current == goal:
            # Rekonstruksi jalur
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(start)
            return path[::-1]

        for neighbor in valid_neighbors(current):
            tentative_g = g_score[current] + 1  # cost tiap langkah = 1

            if neighbor not in g_score or tentative_g < g_score[neighbor]:
                g_score[neighbor] = tentative_g
                f_score = tentative_g + heuristic(neighbor, goal)
                heapq.heappush(open_list, (f_score, neighbor))
                came_from[neighbor] = current

    return None

# Menjalankan A*
path = a_star(start, goal)

print("Jalur terpendek A*:")
print(path)
```

Hasilnya

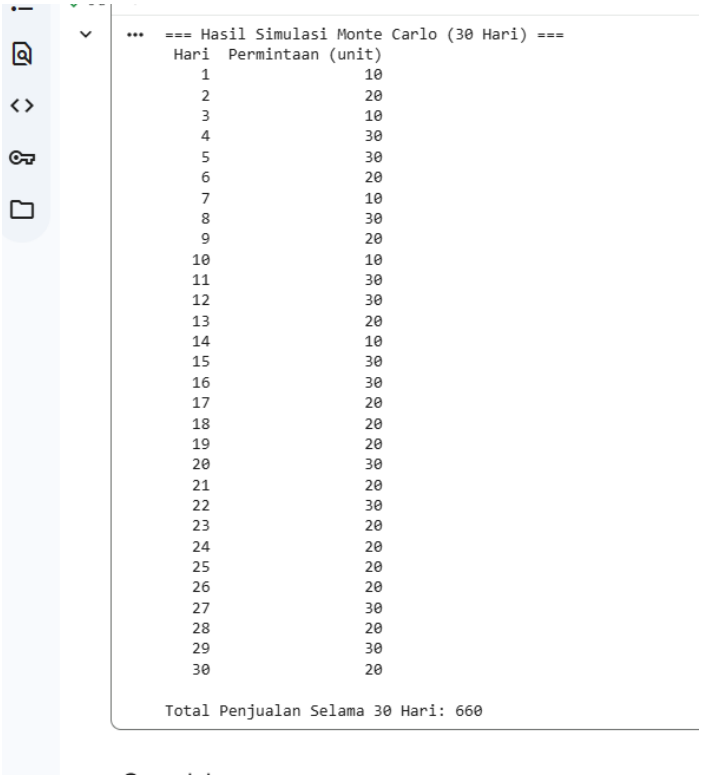
```
Jalur terpendek A*:
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (2, 3), (3, 3)]
```

Algoritma A berhasil menemukan jalur terpendek dari (0,0) ke (3,3). Berdasarkan output, rute optimal yang diperoleh adalah: (0,0) → (0,1) → (0,2) → (0,3) → (1,3) → (2,3) → (3,3). Jalur ini dipilih karena memiliki nilai total cost + heuristik paling kecil.

7. **Monte Carlo Simulation**



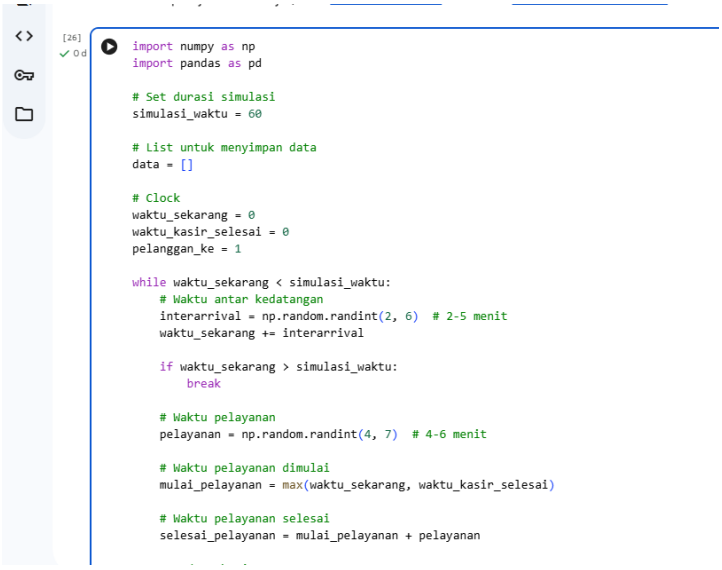
Hasilnya



```
=== Hasil Simulasi Monte Carlo (30 Hari) ===
 Hari  Permintaan (unit)
    1                10
    2                20
    3                10
    4                30
    5                30
    6                20
    7                10
    8                30
    9                20
   10                10
   11                30
   12                30
   13                20
   14                10
   15                30
   16                30
   17                20
   18                20
   19                20
   20                30
   21                20
   22                30
   23                20
   24                20
   25                20
   26                20
   27                30
   28                20
   29                30
   30                20

Total Penjualan Selama 30 Hari: 660
```

Hasil simulasi menunjukkan variasi permintaan tiap hari sesuai probabilitas historis. Total penjualan 30 hari diperoleh dari penjumlahan seluruh nilai permintaan acak tersebut, sehingga memberikan perkiraan penjualan yang realistis meskipun hasilnya dapat berbeda setiap kali simulasi dijalankan.

8. **Discrete Event Simulation**



```
[26]
0d
import numpy as np
import pandas as pd

# Set durasi simulasi
simulasi_waktu = 60

# List untuk menyimpan data
data = []

# Clock
waktu_sekarang = 0
waktu_kasir_selesai = 0
pelanggan_ke = 1

while waktu_sekarang < simulasi_waktu:
    # Waktu antar kedatangan
    interarrival = np.random.randint(2, 6)  # 2-5 menit
    waktu_sekarang += interarrival

    if waktu_sekarang > simulasi_waktu:
        break

    # Waktu pelayanan
    pelayanan = np.random.randint(4, 7)  # 4-6 menit

    # Waktu pelayanan dimulai
    mulai_pelayanan = max(waktu_sekarang, waktu_kasir_selesai)

    # Waktu pelayanan selesai
    selesai_pelayanan = mulai_pelayanan + pelayanan
```

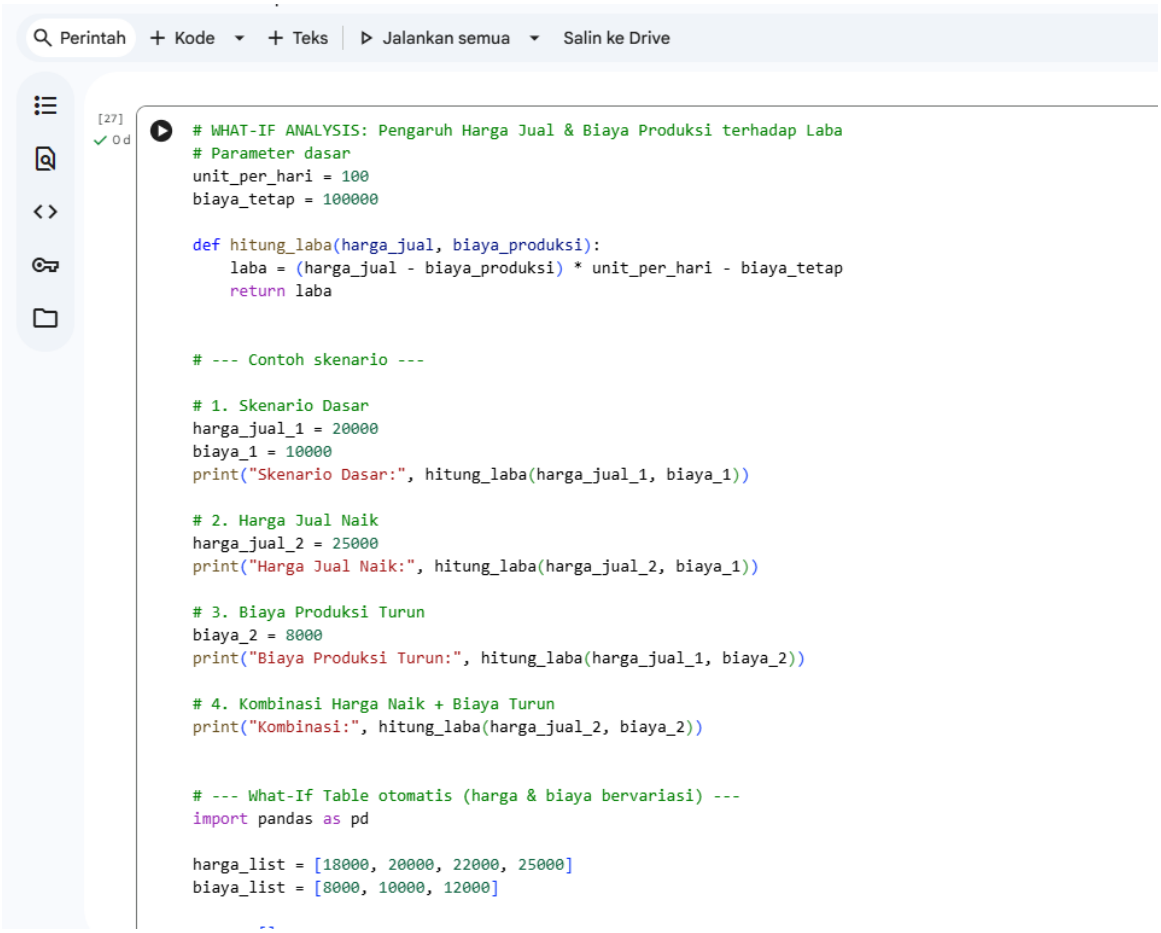
```
        # Update kasir
        waktu_kasir_selesai = selesai_pelayanan

        # Waktu tunggu
        waktu_tunggu = mulai_pelayanan - waktu_sekarang

        # Simpan data
        data.append([
            pelanggan_ke,
            waktu_sekarang,
            pelayanan,
            mulai_pelayanan,
            selesai_pelayanan,
            waktu_tunggu
        ])

        pelanggan_ke += 1

    # Buat tabel hasil
    tabel = pd.DataFrame(data, columns=[
        "Pelanggan",
        "Waktu Datang",
        "Waktu Pelayanan",
        "Mulai Dilayani",
        "Selesai Dilayani",
        "Waktu Tunggu"
    ])

    print("=== Hasil Discrete Event Simulation (1 Kasir, 60 Menit) ===")
    print(tabel.to_string(index=False))
```

Hasilnya

```
=== Hasil Discrete Event Simulation (1 Kasir, 60 Menit) ===
 Pelanggan  Waktu Datang  Waktu Pelayanan  Mulai Dilayani  Selesai Dilayani  Waktu Tunggu
         1             2                6               2                 8             0
         2             4                4               8                12             4
         3             6                4              12                16             6
         4            10                4              16                20             6
         5            13                6              20                26             7
         6            15                4              26                30            11
         7            19                6              30                36            11
         8            21                5              36                41            15
         9            24                4              41                45            17
        10            29                6              45                51            16
        11            34                6              51                57            17
        12            36                5              57                62            21
        13            40                5              62                67            22
        14            44                5              67                72            23
        15            46                5              72                77            26
        16            51                5              77                82            26
        17            56                5              82                87            26
        18            60                4              87                91            27
```

Setelah hasil simulasi diperoleh, dapat terlihat bagaimana antrian terbentuk, berapa banyak pelanggan yang berhasil dilayani dalam 1 jam, serta bagaimana waktu tunggu rata-rata pelanggan. Informasi ini membantu memahami efisiensi kasir dan potensi penumpukan antrian.

9. What-If Analysis




```python
# WHAT-IF ANALYSIS: Pengaruh Harga Jual & Biaya Produksi terhadap Laba
# Parameter dasar
unit_per_hari = 100
biaya_tetap = 100000

def hitung_laba(harga_jual, biaya_produksi):
    laba = (harga_jual - biaya_produksi) * unit_per_hari - biaya_tetap
    return laba


# --- Contoh skenario ---

# 1. Skenario Dasar
harga_jual_1 = 20000
biaya_1 = 10000
print("Skenario Dasar:", hitung_laba(harga_jual_1, biaya_1))

# 2. Harga Jual Naik
harga_jual_2 = 25000
print("Harga Jual Naik:", hitung_laba(harga_jual_2, biaya_1))

# 3. Biaya Produksi Turun
biaya_2 = 8000
print("Biaya Produksi Turun:", hitung_laba(harga_jual_1, biaya_2))

# 4. Kombinasi Harga Naik + Biaya Turun
print("Kombinasi:", hitung_laba(harga_jual_2, biaya_2))


# --- What-If Table otomatis (harga & biaya bervariasi) ---
import pandas as pd

harga_list = [18000, 20000, 22000, 25000]
biaya_list = [8000, 10000, 12000]

rows = []

for h in harga_list:
    for b in biaya_list:
        rows.append([h, b, hitung_laba(h, b)])

df = pd.DataFrame(rows, columns=["Harga Jual", "Biaya Produksi", "Laba"])
print("\nTabel What-If:")
print(df)
```

Hasilnya

```
Skenario Dasar: 900000
Harga Jual Naik: 1400000
Biaya Produksi Turun: 1100000
Kombinasi: 1600000

Tabel What-If:
    Harga Jual  Biaya Produksi     Laba
0        18000            8000   900000
1        18000           10000   700000
2        18000           12000   500000
3        20000            8000  1100000
4        20000           10000   900000
5        20000           12000   700000
6        22000            8000  1300000
7        22000           10000  1100000
8        22000           12000   900000
9        25000            8000  1600000
10       25000           10000  1400000
11       25000           12000  1200000
```

Hasil What-If Analysis menunjukkan bahwa laba perusahaan sangat dipengaruhi oleh perubahan harga jual dan biaya produksi. Pada kondisi awal laba sebesar Rp 900.000. Ketika harga jual dinaikkan, laba meningkat menjadi Rp 1.400.000, sedangkan penurunan biaya produksi menghasilkan laba Rp 1.100.000. Kombinasi keduanya memberikan hasil tertinggi yaitu Rp 1.600.000. Secara umum, harga jual yang lebih tinggi dan biaya produksi yang lebih rendah menghasilkan profit yang lebih besar.