In [ ]:
```
import numpy as np
from math import e
from copy import copy
```

Constants and globals

`SPEED` : the speed limit of a car through the system. 13.4 m/s = 30 mph

`TOLL_DELAY` : the time it takes for a payment to be processed. Online, we found that the average time may be around four seconds for a payment to be processed.

`paths` : shared among several functions. Stores a list of lists representing all possible paths through a toll system and their associated costs with respect to time.

In [ ]:
```
SPEED: float = 13.4
TOLL_DELAY: float = 4.0
paths: list = []
```

Time Delta Function:

This function computes time given a distance and a speed.

In [ ]:
```
def time_delta(distance: float, speed: float = SPEED) -> float:
    return distance / speed
```

Time to Merge

This function computes a time given distance and speed. Merge penalty curve is applied here.

In [ ]:
```
def time_to_merge(distance: float, speed: float = SPEED) -> float:
    t: float = time_delta(distance, speed)
    return (t / (pow(e, -1.2 * (1 - (t * (1 / 2.5))) - 1) - 0.1)) + t
```

Time Toll

This function computes the time given speed and distance, but adds the time taken to process a payment.

In [ ]:
```
def time_toll(distance, speed: float = SPEED, delay: float = TOLL_DELAY) ->
    return time_delta(distance, speed) + delay
```

Draw Path

This function takes a given toll plaza with defined paths, merges, distances, etc. and computes the deterministic set of paths with merges. The paths through a given toll plaza are stored in the global `paths` variable and will be used again later to calulate the cost in time for each given path.

In [ ]:
```python
def draw_path(
    toll,
    start: str,
    stop=False,
    segments: list = [],
    solution: list = [],
    weights: list = [],
) -> list:
    if stop:
        paths.append(weights[:-1].copy())

    for i in toll[start]:
        seg: list = [start, i[0]]
        if seg not in segments:
            segments.append(seg)
            solution.append(i[0])
            weights.append(i[1])

            draw_path(toll, i[0], i[1] == 0, segments, solution, weights)

            # Backtrack
            if len(segments) != 0:
                segments.pop()
            if len(solution) != 0:
                solution.pop()
            if len(weights) != 0:
                weights.pop()

    return paths
```

Calc Time

This function computes the time at each step on a path. and returns a total path time as well as the number of merges in the specific path.

In [ ]:
```
def calc_time(path: list) -> tuple[float | int]:
    t: float = 0
    merges: int = 0
    for i in path:
        if i == "10t":
            i: float = float(i[0:-1])
            t += time_toll(i)
        elif i < 0:
            t += time_to_merge(abs(i))
            merges += 1
        else:
            t += time_delta(i)
    return t, merges
```

Run

This function serves as a driver for computing times for a toll with any number of nodes. We use this to see the min, max, and average time taken to get through a toll as well as aditional information about specific paths that may allow us to make inferences.

In [ ]:
```
def run(t, s):

    # Collect all paths for a toll configuration
    for i in s:
        pathlist = []
        for i in draw_path(toll=t, start=i, solution=[i]):
            pathlist.append(i)

    # Generate list of times associated with their respective path
    T = []
    for p in pathlist:
        time, merges = calc_time(p)
        print(f"{p} --- {round(time, 3)} --- {merges}")
        T.append(time)

    print(f"\nMIN  = {round(min(T), 3)} s")
    print(f"MAX  = {round(max(T), 3)} s")
    print(f"MEAN = {round(np.mean(T), 3)} s")
```

`Note on toll configuration encoding`:

The format is as follows:

```
Name of toll = {
    startnode_lane: [(connectednode_lane, distance to node),
(...), ...],
    ...

    endnode_lane: [(endnode_lane, 0)]

}
```

lanes with distances marked "10t" are parsed as being toll booth segemnts. So "10t" would be read as a segment of length 10 that is a toll booth segment.

Lanes that have a negative distance are parsed as being merges. So "-100" would indicate that a lane is merging over a distance of 100 meters.

Toll Plaza Configurations

We have 4 different configurations:

`TOLL1A`: Base example. 2 lanes in, 3 toll booths, 2 lanes out with only 1 lane allowed to merge once before and after the toll.
`TOLL1B`: Similar to base example, but now both lanes are allowed to merge once before and after the toll.
`TOLL2A`: Similar to TOLL1B except all lanes are either splitting or merging, forcing a driver to eventually merge.
`TOLL3A`: 1 lane in, 4 toll booths, 1 lane out, with the toll plaza bisected, disallowing merging in some cases.

All toll configurations for this test of the implementation have a total toll plaza length of 370 meters.
The only thing that varries is the merge distance and configuration.

In [ ]:
```
TOLL1A: dict = {
    "t0_a": [("t1_a", 50)],
    "t0_b": [("t1_b", 50)],
    "t1_a": [("t2_a", 100)],
    "t1_b": [("t2_b", 100), ("t2_c", 100)],
    "t2_a": [("t3_a", 30)],
    "t2_b": [("t3_b", 30)],
    "t2_c": [("t3_c", 30)],
    "t3_a": [("t4_a", "10t")],
    "t3_b": [("t4_b", "10t")],
    "t3_c": [("t4_c", "10t")],
    "t4_a": [("t5_a", 30)],
    "t4_b": [("t5_b", 30)],
    "t4_c": [("t5_c", 30)],
    "t5_a": [("t6_a", 100)],
    "t5_b": [("t6_b", 100)],
    "t5_c": [("t6_b", -100)],
    "t6_a": [("t7_a", 50)],
    "t6_b": [("t7_b", 50)],
    "t7_a": [("t7_a", 0)],
    "t7_b": [("t7_b", 0)],
}
```

In [ ]:
```
# Lane a can now merge into and out of lane b
TOLL1B: dict = {
    "t0_a": [("t1_a", 50)],
    "t0_b": [("t1_b", 50)],
    "t1_a": [("t2_a", 100), ("t2_b", -100)],
    "t1_b": [("t2_b", 100), ("t2_c", 100)],
    "t2_a": [("t3_a", 30)],
    "t2_b": [("t3_b", 30)],
    "t2_c": [("t3_c", 30)],
    "t3_a": [("t4_a", "10t")],
    "t3_b": [("t4_b", "10t")],
    "t3_c": [("t4_c", "10t")],
    "t4_a": [("t5_a", 30)],
    "t4_b": [("t5_b", 30)],
    "t4_c": [("t5_c", 30)],
    "t5_a": [("t6_a", 100)],
    "t5_b": [("t6_b", 100), ("t6_a", -100)],
    "t5_c": [("t6_b", -100)],
    "t6_a": [("t7_a", 50)],
    "t6_b": [("t7_b", 50)],
    "t7_a": [("t7_a", 0)],
    "t7_b": [("t7_b", 0)],
}
```

In [ ]:
```
TOLL2A: dict = {
    "t0_a": [("t1_a", 50)],
    "t0_b": [("t1_b", 50)],
    "t1_a": [("t2_a", 100), ("t2_b", -100)],
    "t1_b": [("t2_b", -100), ("t2_c", 100)],
    "t2_a": [("t3_a", 30)],
    "t2_b": [("t3_b", 30)],
    "t2_c": [("t3_c", 30)],
    "t3_a": [("t4_a", "10t")],
    "t3_b": [("t4_b", "10t")],
    "t3_c": [("t4_c", "10t")],
    "t4_a": [("t5_a", 30)],
    "t4_b": [("t5_b", 30)],
    "t4_c": [("t5_c", 30)],
    "t5_a": [("t6_a", -100)],
    "t5_b": [("t6_a", -100), ("t6_b", -100)],
    "t5_c": [("t6_b", -100)],
    "t6_a": [("t7_a", 50)],
    "t6_b": [("t7_b", 50)],
    "t7_a": [("t7_a", 0)],
    "t7_b": [("t7_b", 0)],
}
```

In [ ]:
```
TOLL3A: dict = {
    "t0_a": [("t1_a", 50), ("t1_b", 50)],
    "t1_a": [("t2_a", 50)],
    "t1_b": [("t2_b", 50)],
    "t2_a": [("t3_a", 50), ("t3_b", 50)],
    "t2_b": [("t3_c", 50), ("t3_d", 50)],
    "t3_a": [("t4_a", 30)],
    "t3_b": [("t4_b", 30)],
    "t3_c": [("t4_c", 30)],
    "t3_d": [("t4_d", 30)],
    "t4_a": [("t5_a", "10t")],
    "t4_b": [("t5_b", "10t")],
    "t4_c": [("t5_c", "10t")],
    "t4_d": [("t5_d", "10t")],
    "t5_a": [("t6_a", 30)],
    "t5_b": [("t6_b", 30)],
    "t5_c": [("t6_c", 30)],
    "t5_d": [("t6_d", 30)],
    "t6_a": [("t7_a", -50)],
    "t6_b": [("t7_a", 50)],
    "t6_c": [("t7_b", 50)],
    "t6_d": [("t7_b", -50)],
    "t7_a": [("t8_a", 50)],
    "t7_b": [("t8_b", 50)],
    "t8_a": [("t9_a", -50)],
    "t8_b": [("t9_a", -50)],
    "t9_a": [("t9_a", 0)],
}
```

Toll 1A

In [ ]:
```
# Reset paths variable
paths = []
run(TOLL1A, ["t0_a", "t0_b"])
```

```
[50, 100, 30, '10t', 30, 100, 50] --- 31.612 --- 0
[50, 100, 30, '10t', 30, 100, 50] --- 31.612 --- 0
[50, 100, 30, '10t', 30, -100, 50] --- 33.534 --- 1

MIN  = 31.612 s
MAX  = 33.534 s
MEAN = 32.253 s
```

Toll 1B

In [ ]:
```
paths = []
run(TOLL1B, ["t0_a", "t0_b"])
```

```
[50, 100, 30, '10t', 30, 100, 50] --- 31.612 --- 0
[50, -100, 30, '10t', 30, 100, 50] --- 33.534 --- 1
[50, -100, 30, '10t', 30, -100, 50] --- 35.456 --- 2
[50, 100, 30, '10t', 30, 100, 50] --- 31.612 --- 0
[50, 100, 30, '10t', 30, -100, 50] --- 33.534 --- 1
[50, 100, 30, '10t', 30, -100, 50] --- 33.534 --- 1

MIN  = 31.612 s
MAX  = 35.456 s
MEAN = 33.213 s
```

Toll 2A

In [ ]:
```
paths = []
run(TOLL2A, ["t0_a", "t0_b"])
```

```
[50, 100, 30, '10t', 30, -100, 50] --- 33.534 --- 1
[50, -100, 30, '10t', 30, -100, 50] --- 35.456 --- 2
[50, -100, 30, '10t', 30, -100, 50] --- 35.456 --- 2
[50, -100, 30, '10t', 30, -100, 50] --- 35.456 --- 2
[50, -100, 30, '10t', 30, -100, 50] --- 35.456 --- 2
[50, 100, 30, '10t', 30, -100, 50] --- 33.534 --- 1

MIN  = 33.534 s
MAX  = 35.456 s
MEAN = 34.815 s
```

Toll3A

In [ ]:
```
paths = []
run(TOLL3A, ["t0_a"])
```

```
[50, 50, 50, 30, '10t', 30, -50, 50, -50] --- 44.836 --- 2
[50, 50, 50, 30, '10t', 30, 50, 50, -50] --- 38.224 --- 1
[50, 50, 50, 30, '10t', 30, 50, 50, -50] --- 38.224 --- 1
[50, 50, 50, 30, '10t', 30, -50, 50, -50] --- 44.836 --- 2

MIN  = 38.224 s
MAX  = 44.836 s
MEAN = 41.53 s
```