In [2]:

```
!pip install gurobipy
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/publi
c/simple/
Requirement already satisfied: gurobipy in /usr/local/lib/python3.7/dist-packages (9.5.2)
```

In [3]:

```
from google.colab import drive
import os
drive.mount('/content/drive', force_remount=True)
os.chdir(os.path.join(os.getcwd(), 'drive','MyDrive', 'Colab Notebooks', 'gurobi'))
```

```
Mounted at /content/drive
```

In [4]:

```
# !chmod 755 gurobi/grbgetkey
# !gurobi/grbgetkey ac8a9496-5310-11ed-994f-0242ac120002
import gurobipy as gp
with open('gurobi.lic', 'r') as f:
    lic = f.readlines()

WLSACCESSID = lic[-3].replace('\n', '').replace('WLSACCESSID=', '')
WLSSECRET = lic[-2].replace('\n', '').replace('WLSSECRET=', '')
LICENSEID = int( lic[-1].replace('\n', '').replace('LICENSEID=', '') )

e = gp.Env(empty=True)
e.setParam('WLSACCESSID', WLSACCESSID)
e.setParam('WLSSECRET', WLSSECRET)
e.setParam('LICENSEID', LICENSEID)
e.start()
```

```
Set parameter WLSAccessID
Set parameter WLSSecret
Set parameter LicenseID
Academic license - for non-commercial use only - registered to klt45@cornell.edu
```

Out[4]:

```
<gurobipy.Env, Parameter changes: WLSAccessID=(user-defined), WLSSecret=(user-defined), Li
censeID=(user-defined)>
```

The following code cells illustrate how I came up with valid constraints based on the decision variable:

$$X_{ait} = \begin{cases} 1 & \text{if job a is assigned to machine i at time t} \\ 0 & \text{otherwise} \end{cases}$$

$$a \in \{1, 2, \ldots, n\}, i \in \{1, 2, \ldots, m\}, t \in \{0, 1, \ldots, T\}$$

Note that as discussed in class, this formulation may be computationally expensive, as it will require

$$m \cdot n \cdot T$$

variables. Nonetheless, I wanted to devise a working solution before experimenting with other formulations. To come up with my constraints, I verified them against the feasible scheduling example in class where

$$m = 3, n = 7, T = 21$$

In [5]:

```
import pandas as pd
import numpy as np
from itertools import combinations, product, permutations

def getClassExample(zero_index=False):
    m = 3
    n = 7
    prec_dict = {1:[], 2:[], 3:[], 4: [1,3], 5:[1,2], 6:[4], 7:[]}
    proc_dict = {1: 3, 2: 1, 3:1, 4:1, 5:5, 6:5, 7:5}
```

```python
    T = sum(proc_dict.values())
    if zero_index:
        prec_dict = {key-1: [i-1 for i in value] for key, value in prec_dict.items()}
        proc_dict = {key-1: value for key, value in proc_dict.items()}
    return m, n, T, prec_dict, proc_dict

m, n, T, prec_dict, proc_dict = getClassExample()


#n by m by T
dp = [ [np.zeros(T).astype(int) for i in range(m)] for j in range(n) ]
dp = pd.DataFrame( dp, columns = [f'Machine {i}' for i in range(1, m+1)], index = [f'Job 

# x(a,i, t) = 1 if task a is assigned to machine i at time t
# 0<=a<=n-1, 0<=i<=m-1, 0<=t<=T  --> due to zero indexing of python

# a valid solution is:
# x(2, 1, 0) , x(3, 1, 1), x(4, 1, 3), x(6, 1, 4)
# x(1, 2, 0) , x(5, 2, 3),
# x(7, 3, 0)
for i,j,k in [ (2, 1, 0), (3, 1, 1), (4, 1, 3), (6, 1, 4), (1, 2, 0), (5, 2, 3), (7, 3, 0)
    dp.iloc[i-1,j-1][k] = 1

# I did A LOT of debugging:

# for i,j,k in [ [1,3,0], [2,2,0], [3,1,0], [4,1,1], [5,2,1], [6,1,2], [7,3,1] ]:
#     dp.iloc[i-1,j-1][k] = 1
# for i,j,k in [ [1,2,0], [2,1,0], [3,3,0], [4,1,1], [5,3,1], [6,1,2], [7,2,3] ]:
#     dp.iloc[i-1,j-1][k] = 1
# arr = [ [0,1,0], [1,0,0], [2,2,0], [3,0,1], [4,1,3], [5,0,2], [6,2,1] ]
# arr = [[1, 2, 0], [2, 1, 0], [3, 3, 0], [4, 1, 1], [5, 2, 3], [6, 1, 2], [7, 3, 1]]
# arr = [ [i+1, j+1, k] for i,j,k in arr ]
# for i,j,k in [[1, 2, 0], [2, 1, 0], [3, 3, 0], [4, 1, 1], [5, 2, 3], [6, 1, 2], [7, 3, 
#     dp.iloc[i-1,j-1][k] = 1

dp
```

Out[5]:

	Machine 1	Machine 2	Machine 3
Job 1	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...	[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
Job 2	[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
Job 3	[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
Job 4	[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
Job 5	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...	[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
Job 6	[0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
Job 7	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...	[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...

Our first constraint is that each job can only be assigned to exactly one machine. This is equivalent to the following constraint:

$$\sum_{i=1}^{m} \sum_{t=0}^{T} X_{ait} = 1 \quad \forall a \in \{1, 2, \ldots, n\}$$

In [6]:

```python
#1. If this loop prints, then each job is not assigned to one machine
for a in range(n):
    arr = dp.iloc[a].to_numpy() # m by T
    if sum( arr[i][t] for i in range(m) for t in range(T) ) != 1:
        print('Each job can only be assigned to one machine at one time')

# check row sum of dp should all be 1 for each entry
dp.sum(axis=1).apply( lambda x: sum(x) )
```

Out[6]:

```
Job 1    1
Job 2    1
Job 3    1
Job 4    1
Job 5    1
Job 6    1
Job 7    1
dtype: int64
```

Our second constraint is that for every machine at time t, it can be assigned at most one job. This is equivalent to the following:

$$\sum_{a=1}^{n} X_{ait} \leq 1 \quad \forall i \in \{1, 2, \ldots, m\}, \forall t \in \{0, 1, \ldots, T\}$$

In [7]:

```python
#2. If this loop prints, then each job is not assigned at most one job
for j in range(m):
    for t in range(T):
        if sum(dp.iloc[a,j][t] for a in range(n) ) > 1:
            print('for every machine at time t, it can be assigned at most one job')

# column sum of dp should be <=1 for each entry
pd.set_option('display.max_colwidth',100)
dp.sum(axis=0)
```

Out[7]:

```
Machine 1    [1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Machine 2    [1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Machine 3    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
dtype: object
```

The third constraint is that for each job a' must be completed before job a can start. Denote a < a' to mean that job a must be completed before task a' can start. Denote p(a) to be the processing time of job a. This is equivalent to the following:

$$\sum_{i=1}^{m} \sum_{t'=0}^{t-1} X_{a'it'} \leq \sum_{i=1}^{m} \sum_{t'=0}^{t+p(a)-1} X_{ait'} \quad \forall a \prec a' \in \{1, 2, \ldots, n\}, \forall t \in \{0, 1, \ldots, T\}$$

In [8]:

```python
#3. If this loop prints, then each job a' is not completed before job a starts
# a < a_ to mean that job a must be completed before task a_ can start
# Example: if a_ = 4, a = 1 and p(1) = 3, then
# then sum over i, t' of x(4, i, t') <= sum over i, t' of x(1, i, t) + p(1)

preceding_pairs = [ i for j in [ list(product( value, [key]) ) for key, value in prec_dict

for a, a_ in preceding_pairs:
    for t in range(T):
        prev = dp.iloc[a-1].to_numpy()
        later = dp.iloc[a_-1].to_numpy()
```

```
            if sum( prev[i][t_] for i in range(m) for t_ in range(t)) < sum( later[i][t_] for
                print(f'job {a_} cannot begin while job {a} + p({a}) = {proc_dict[a] + t}' )
```

The fourth constraint is that job a cannot begin on a machine i while job a' + p(a') is still being processed on machine i. This is equivalent to the following constraint:

$$if \quad X_{a'it} == 1, \quad then \quad \sum_{t'=t}^{t+p(a')-1} X_{a'it'} + X_{ait'} \leq 1 \quad \forall perm(a,a'), \forall i \in \{1,2,\ldots,m\}, \forall t \in \{0,1,\ldots,T\}$$

In [9]:

```
#4 If this loop prints, then job a began on a machine i while job a' + p(a') is still bein
# Example: if a = 7 and a' = 1 with p(1) = 3, then
# x(1, i, t) + x(7, i, t) <= 1 for every i, t'<=t+3 (t' = t, t+1, t+2, t+3)
from itertools import permutations
jobs = list( permutations( prec_dict.keys(), 2) )
for i in range(m):
    for a, a_ in jobs:
        for t in range(T):
            if dp.iloc[a_-1,i][t] == 1:
                arr = dp.iloc[a-1,i]
                arr_ = dp.iloc[a_-1,i]
                if sum( [arr[t_] + arr_[t_] for t_ in range(t, min(T, t+proc_dict[a_]))] )
                    print(f'job {a} cannot begin on a machine {i+1} while job {a_} + p({a_
```

Our objective function is to minimize the makespan, which is the time at which the last job is completed. The following linear objective function can minimize the makespan:

$$\min \sum_{a=1}^{n} \sum_{i=1}^{m} \sum_{t=0}^{T} X_{ait} \cdot (t + p(a)) \quad \forall a \in \{1,2,\ldots,n\}, \forall i \in \{1,2,\ldots,m\}, \forall t \in \{0,1,\ldots,T\}$$

Let's calculate our makespan (should be 9) and objective function value

In [10]:

```
objectiveFunction = 0
makespan = float('-inf')
for i in range(n):
    for j in range(m):
        for t in range(T):
            if dp.iloc[i,j][t] == 1:
                objectiveFunction += t + proc_dict[i + 1]
                makespan = max(makespan, t + proc_dict[i + 1])

print('Makespan:', makespan)
print('Objective Function Value:', objectiveFunction)
```

```
Makespan: 9
Objective Function Value: 32
```

Putting everything together! Note: 500 jobs took too long (sorry!)

10 jobs completes instantly, however, when I try 100, I run out of RAM (which is expected, as my formulation has O(n*m*T) variables

In [11]:

```
import gurobipy as gp
from gurobipy import GRB
```

```
df=pd.read_csv('sched_med_proc_times.csv', header=None)
df2=pd.read_csv('schedmed_prec.csv') # precedence constraints
n=10#df.shape[0]          # Number of Jobs
m=2            # Number of machines
prec_dict={}  # Should have keys 1....N and each key is mapped to the set of preceding jol
proc_dict={} # Same keys as prec_dict, but mapped to processing times instead.
T= df.iloc[:n, 1].sum()
for j in range(n):
    proc_dict[j]=df.iloc[j,1]
    prec_dict[j]=list(df2.iloc[j].dropna().to_numpy() -1 )[1:]

def Scheduling(m,n,prec_dict,proc_dict,T):
    """
    Parameters
    ----------
    m : int - number of machines
    n : int - number of jobs
    prec_dict : dictionary - keys are jobs, values are the set of jobs that must be comple
    Example: prec_dict[3] = [1,2] means that jobs 1 and 2 must be completed before job 3 c
    proc_dict : dictionary - keys are jobs, values are the processing times of the jobs
    Example: proc_dict[3] = 5 means that job 3 takes 5 time units to complete
    T : int - upper bound on the makespan
    """

    model = gp.Model(env=e)
    list_of_ait = list(product(range(n), range(m), range(T)))

    x_ait = model.addVars(list_of_ait, vtype=GRB.BINARY, name="x_ait")

    #1. each job can only be assigned to one machine at one time
    model.addConstrs(
        gp.quicksum(x_ait[a,i,t] for i in range(m) for t in range(T)) == 1 for a in range
        )

    #2. for every machine at time t, it can be assigned at most one job
    model.addConstrs(
        gp.quicksum(x_ait[a,i,t] for a in range(n)) <= 1 for i in range(m) for t in range
        )

    #3. for each job a' must be completed before job a can start
    # a < a_ to mean that job a must be completed before task a_ can start
    preceding_pairs = [ i for j in [ list(product( value, [key]) ) for key, value in prec_
    for a, a_ in preceding_pairs:
        for t in range(T):
            model.addConstr(
                gp.quicksum(x_ait[a,i,t_] for i in range(m) for t_ in range(t)) >= gp.quic
                )

    #4. job a cannot begin on a machine i while job a' + p(a') is still being processed or
    jobs = list(permutations(range(n), 2))
    model.addConstrs(
        ( (x_ait[a_,i,t] == 1) >> ( gp.quicksum( x_ait[a,i,t_] + x_ait[a_,i,t_] for t_ in
    )


    model.setObjective(gp.quicksum( x_ait[a,i,t] * (t+ proc_dict[a]) for a in range(n) for

    model.optimize()

    if model.status == GRB.Status.OPTIMAL:
        print('Optimal objective: %g' % model.objVal)
        makespan = float('-inf')
        print('Optimal solution:')
        for v in model.getVars():
            if v.x > 0:
```

```python
                print('%s %g' % (v.varName, v.x))
                makespan = max(makespan, int(v.varName.split('[')[1].split(']')[0].split('
        print(f'makespan: {makespan}')
    else:
        print('No solution')

Scheduling(m,n,prec_dict,proc_dict,T)
```

```
Gurobi Optimizer version 9.5.2 build v9.5.2rc0 (linux64)
Thread count: 1 physical cores, 2 logical processors, using up to 2 threads
Academic license - for non-commercial use only - registered to klt45@cornell.edu
Optimize a model with 2098 rows, 1160 columns and 252084 nonzeros
Model fingerprint: 0xd548b1ef
Model has 10440 general constraints
Variable types: 0 continuous, 1160 integer (1160 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [2e+00, 7e+01]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
  GenCon rhs range [1e+00, 1e+00]
  GenCon coe range [1e+00, 1e+00]
Presolve removed 1553 rows and 778 columns
Presolve time: 0.91s
Presolved: 545 rows, 382 columns, 7230 nonzeros
Variable types: 0 continuous, 382 integer (382 binary)
Found heuristic solution: objective 365.0000000

Root relaxation: objective 2.620000e+02, 95 iterations, 0.00 seconds (0.00 work units)

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

*    0     0               0     262.0000000  262.00000  0.00%     -    0s

Explored 1 nodes (95 simplex iterations) in 0.99 seconds (0.84 work units)
Thread count was 2 (of 2 available processors)

Solution count 2: 262 365

Optimal solution found (tolerance 1.00e-04)
Best objective 2.620000000000e+02, best bound 2.620000000000e+02, gap 0.0000%
Optimal objective: 262
Optimal solution:
x_ait[0,1,0] 1
x_ait[1,0,0] 1
x_ait[2,0,10] 1
x_ait[3,0,13] 1
x_ait[4,0,19] 1
x_ait[5,1,27] 1
x_ait[6,0,27] 1
x_ait[7,0,33] 1
x_ait[8,0,35] 1
x_ait[9,0,40] 1
makespan: 45
```

In [12]:

```python
# With the class example!  Our objective should be 32 (9 is the minimum makespan)
m, n, T, prec_dict, proc_dict = getClassExample(zero_index=True)
Scheduling(m,n,prec_dict,proc_dict,T)
```

```
Gurobi Optimizer version 9.5.2 build v9.5.2rc0 (linux64)
Thread count: 1 physical cores, 2 logical processors, using up to 2 threads
Academic license - for non-commercial use only - registered to klt45@cornell.edu
Optimize a model with 175 rows, 441 columns and 7731 nonzeros
Model fingerprint: 0xfe69a4a1
```

```
Model has 2646 general constraints
Variable types: 0 continuous, 441 integer (441 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+00, 2e+01]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
  GenCon rhs range [1e+00, 1e+00]
  GenCon coe range [1e+00, 1e+00]
Presolve added 321 rows and 0 columns
Presolve removed 0 rows and 54 columns
Presolve time: 0.10s
Presolved: 496 rows, 387 columns, 9354 nonzeros
Variable types: 0 continuous, 387 integer (387 binary)
Found heuristic solution: objective 66.0000000
Found heuristic solution: objective 37.0000000

Root relaxation: objective 3.200000e+01, 19 iterations, 0.00 seconds (0.00 work units)

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

*    0     0               0      32.0000000   32.00000  0.00%     -    0s

Explored 1 nodes (19 simplex iterations) in 0.16 seconds (0.11 work units)
Thread count was 2 (of 2 available processors)

Solution count 3: 32 37 66

Optimal solution found (tolerance 1.00e-04)
Best objective 3.200000000000e+01, best bound 3.200000000000e+01, gap 0.0000%
Optimal objective: 32
Optimal solution:
x_ait[0,1,0] 1
x_ait[1,0,0] 1
x_ait[2,2,0] 1
x_ait[3,1,3] 1
x_ait[4,0,3] 1
x_ait[5,1,4] 1
x_ait[6,2,1] 1
makespan: 9
```