4.1 Machine Bottleneck

In [ ]:

```
%%capture
!pip install -q pyomo
!apt-get install -y -qq coinor-cbc
```

Example

The problem is to schedule a sequence of jobs for a single machine. The data consists of a Python dictionary of jobs. Each job is labeled by a key, and an associated data dictionary provides the time at which the job is released to the for machine processing, the expected duration of the job, and the due date. The problem is to sequence the jobs on the machine to meet the due dates, or show that no such sequence is possible.

In [2]:

```
JOBS = {
    'A': {'release': 2, 'duration': 5, 'due': 10},
    'B': {'release': 5, 'duration': 6, 'due': 21},
    'C': {'release': 4, 'duration': 8, 'due': 15},
    'D': {'release': 0, 'duration': 4, 'due': 10},
    'E': {'release': 0, 'duration': 2, 'due':  5},
    'F': {'release': 8, 'duration': 3, 'due': 15},
    'G': {'release': 9, 'duration': 2, 'due': 22},
}
JOBS
```

Out[2]:

```
{'A': {'due': 10, 'duration': 5, 'release': 2},
 'B': {'due': 21, 'duration': 6, 'release': 5},
 'C': {'due': 15, 'duration': 8, 'release': 4},
 'D': {'due': 10, 'duration': 4, 'release': 0},
 'E': {'due': 5, 'duration': 2, 'release': 0},
 'F': {'due': 15, 'duration': 3, 'release': 8},
 'G': {'due': 22, 'duration': 2, 'release': 9}}
```

The Machine Scheduling Problem

A schedule consists of a dictionary listing the start and finish times for each job. Once the order of jobs has been determined, the start time can be no earlier than when the job is released for processing, and no earlier than the finish of the previous job.

The following cell presents a function which, given the JOBS data and an order list of jobs indices, computes the start and finish times for all jobs on a single machine. We use this to determine the schedule if the jobs are executed in alphabetical order.

In [3]:

```
def schedule(JOBS, order=sorted(JOBS.keys())):
    """Schedule a dictionary of JOBS on a single machine in a specified order."""
    start = 0
    finish = 0
    SCHEDULE = {}
    for job in order:
        start = max(JOBS[job]['release'], finish)
        finish = start + JOBS[job]['duration']
        SCHEDULE[job] = {'start': start, 'finish': finish, }
    return SCHEDULE

SCHEDULE = schedule(JOBS)
SCHEDULE
```

Out[3]:

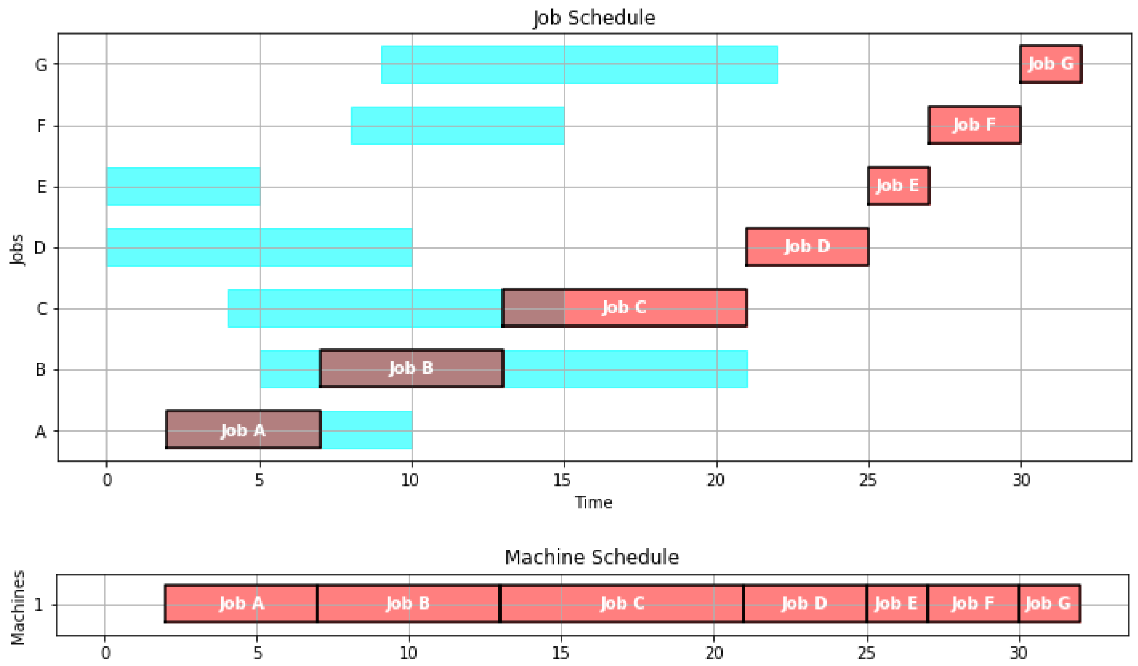
```
{'A': {'finish': 7, 'start': 2},
 'B': {'finish': 13, 'start': 7},
 'C': {'finish': 21, 'start': 13},
 'D': {'finish': 25, 'start': 21},
 'E': {'finish': 27, 'start': 25},
 'F': {'finish': 30, 'start': 27},
 'G': {'finish': 32, 'start': 30}}
```

Gantt Chart

A traditional means of visualizing scheduling data in the form of a Gantt chart. The next cell presents a function `gantt` that plots a Gantt chart given JOBS and SCHEDULE information. Two charts are presented showing job schedule and machine schedule. If no machine information is contained in SCHEDULE, then it assumed to be a single machine operation.

In [4]:

```
%matplotlib inline
import matplotlib.pyplot as plt

def gantt(JOBS, SCHEDULE):
    bw = 0.3
    plt.figure(figsize=(12, 0.7*(len(JOBS.keys()))))
    idx = 0
    for j in sorted(JOBS.keys()):
        x = JOBS[j]['release']
        y = JOBS[j]['due']
        plt.fill_between([x,y],[idx-bw,idx-bw],[idx+bw,idx+bw], color='cyan', alpha=0.6
)
        x = SCHEDULE[j]['start']
        y = SCHEDULE[j]['finish']
        plt.fill_between([x,y],[idx-bw,idx-bw],[idx+bw,idx+bw], color='red', alpha=0.5)
        plt.plot([x,y,y,x,x], [idx-bw,idx-bw,idx+bw,idx+bw,idx-bw],color='k')
        plt.text((SCHEDULE[j]['start'] + SCHEDULE[j]['finish'])/2.0,idx,
            'Job ' + j, color='white', weight='bold',
            horizontalalignment='center', verticalalignment='center')
        idx += 1

    plt.ylim(-0.5, idx-0.5)
    plt.title('Job Schedule')
    plt.xlabel('Time')
    plt.ylabel('Jobs')
    plt.yticks(range(len(JOBS)), JOBS.keys())
    plt.grid()
    xlim = plt.xlim()

    for j in JOBS.keys():
        if 'machine' not in SCHEDULE[j].keys():
            SCHEDULE[j]['machine'] = 1
    MACHINES = sorted(set([SCHEDULE[j]['machine'] for j in JOBS.keys()]))

    plt.figure(figsize=(12, 0.7*len(MACHINES)))
    for j in sorted(JOBS.keys()):
        idx = MACHINES.index(SCHEDULE[j]['machine'])
        x = SCHEDULE[j]['start']
        y = SCHEDULE[j]['finish']
        plt.fill_between([x,y],[idx-bw,idx-bw],[idx+bw,idx+bw], color='red', alpha=0.5)
        plt.plot([x,y,y,x,x], [idx-bw,idx-bw,idx+bw,idx+bw,idx-bw],color='k')
        plt.text((SCHEDULE[j]['start'] + SCHEDULE[j]['finish'])/2.0,idx,
            'Job ' + j, color='white', weight='bold',
            horizontalalignment='center', verticalalignment='center')
    plt.xlim(xlim)
    plt.ylim(-0.5, len(MACHINES)-0.5)
    plt.title('Machine Schedule')
    plt.yticks(range(len(MACHINES)), MACHINES)
    plt.ylabel('Machines')
    plt.grid()

gantt(JOBS, SCHEDULE)
```



Key Performance Indicators

As presented above, a given schedule may not meet all of the due time requirements. In fact, a schedule meeting all of the requirements might not even be possible. So given a schedule, it is useful to have a function that computes key performance indicators.

In [5]:

```
def kpi(JOBS, SCHEDULE):
    KPI = {}
    KPI['Makespan'] = max(SCHEDULE[job]['finish'] for job in JOBS)
    KPI['Max Pastdue'] = max(max(0, SCHEDULE[job]['finish'] - JOBS[job]['due']) for job in JOBS)
    KPI['Sum of Pastdue'] = sum(max(0, SCHEDULE[job]['finish'] - JOBS[job]['due']) for job in JOBS)
    KPI['Number Pastdue'] = sum(SCHEDULE[job]['finish'] > JOBS[job]['due'] for job in JOBS)
    KPI['Number on Time'] = sum(SCHEDULE[job]['finish'] <= JOBS[job]['due'] for job in JOBS)
    KPI['Fraction on Time'] = KPI['Number on Time']/len(JOBS)
    return KPI

kpi(JOBS, SCHEDULE)
```

Out[5]:

```
{'Fraction on Time': 0.2857142857142857,
 'Makespan': 32,
 'Max Pastdue': 22,
 'Number Pastdue': 5,
 'Number on Time': 2,
 'Sum of Pastdue': 68}
```

Exercise

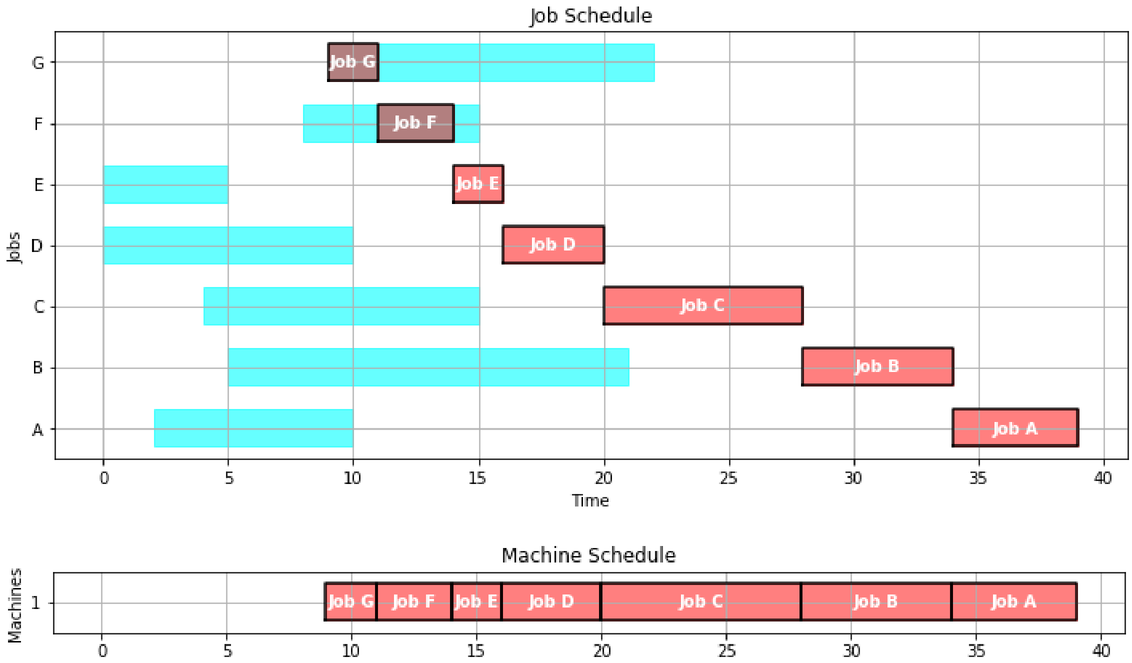
Show the Gantt chart and key performance metrics if the jobs are executed in reverse alphabetical order.

In [6]:

```
order = sorted(JOBS, reverse=True)
gantt(JOBS, schedule(JOBS,order))
kpi(JOBS, schedule(JOBS,order))
```

Out[6]:

```
{'Fraction on Time': 0.2857142857142857,
 'Makespan': 39,
 'Max Pastdue': 29,
 'Number Pastdue': 5,
 'Number on Time': 2,
 'Sum of Pastdue': 76}
```



Empirical Scheduling

There are a number of commonly encountered empirical rules for scheduling jobs on a single machine. These include:

- First-In First-Out (FIFO)
- Last-In, First-Out (LIFO)
- Shortest Processing Time First (SPT)
- Earliest Due Data (EDD)

First-In First-Out

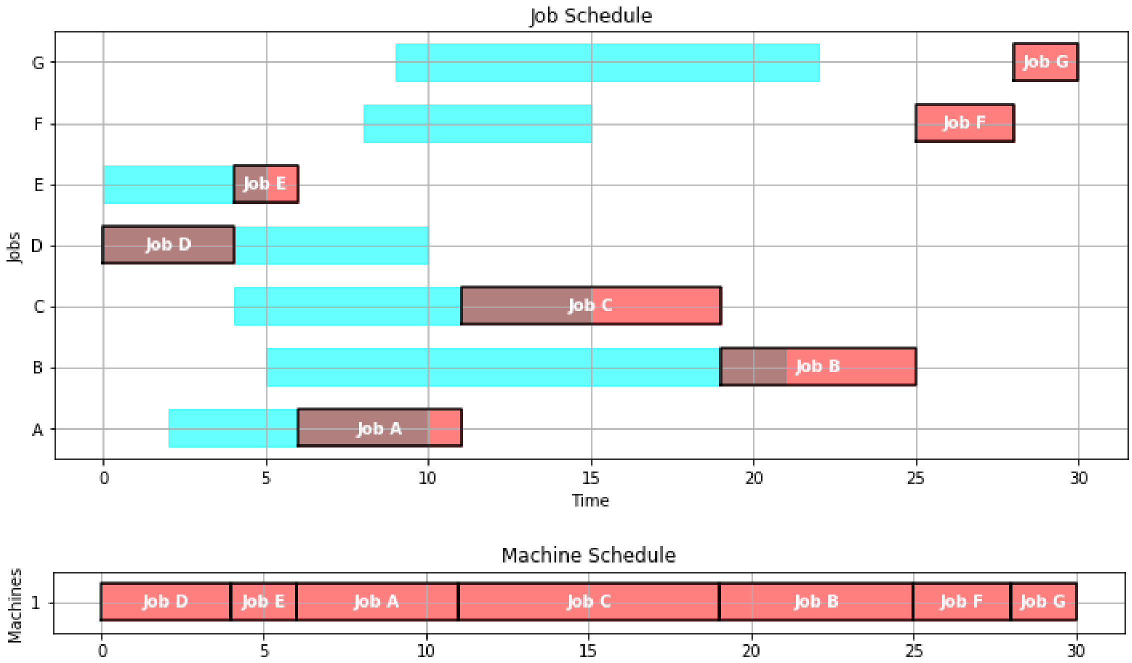
As an example, we'll first look at 'First-In-First-Out' scheduling which executes job in the order they are released. The following function sorts jobs by release time, then schedules the jobs to execute in that order. A job can only be started no earlier than when it is released.

In [7]:

```
def fifo(JOBS):
    order_by_release = sorted(JOBS, key=lambda job: JOBS[job]['release'])
    return schedule(JOBS, order_by_release)

SCHEDULE = fifo(JOBS)
gantt(JOBS, SCHEDULE)
kpi(JOBS, SCHEDULE)
```

Out[7]:

```
{'Fraction on Time': 0.14285714285714285,
 'Makespan': 30,
 'Max Pastdue': 13,
 'Number Pastdue': 6,
 'Number on Time': 1,
 'Sum of Pastdue': 31}
```



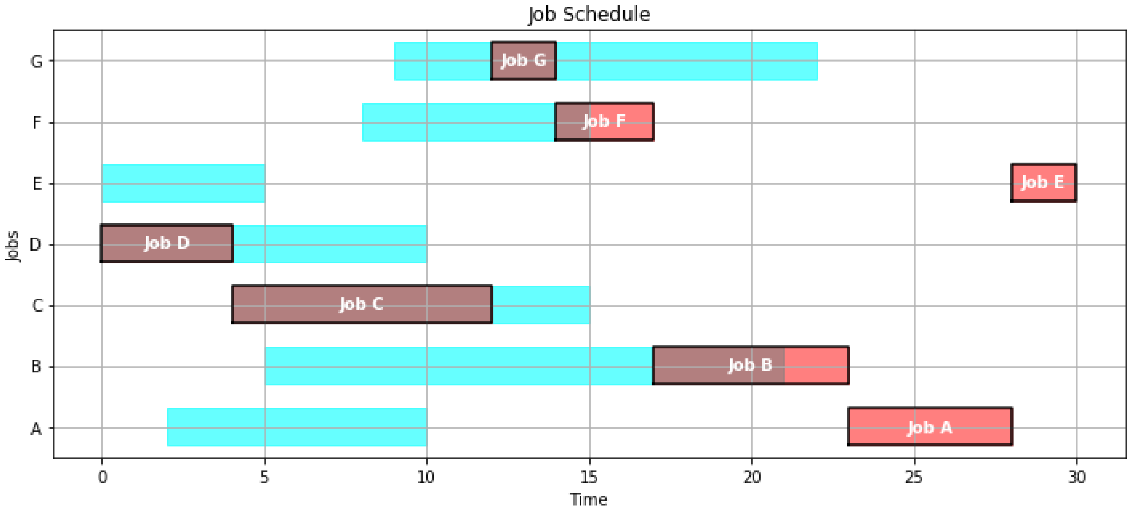
Last-In, First-Out

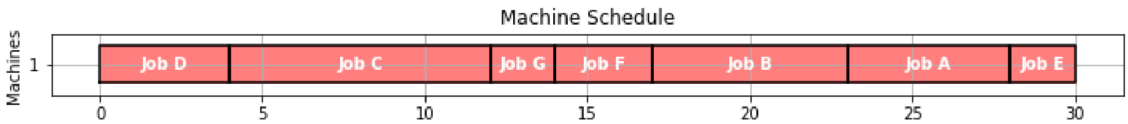
In [8]:

```
def lifo(JOBS):
    unfinished_jobs = set(JOBS.keys())
    start = 0
    while len(unfinished_jobs) > 0:
        start = max(start, min(JOBS[job]['release'] for job in unfinished_jobs))
        lifo = {job:JOBS[job]['release'] for job in unfinished_jobs if JOBS[job]
['release'] <= start}
        job = max(lifo, key=lifo.get)
        finish = start + JOBS[job]['duration']
        unfinished_jobs.remove(job)
        SCHEDULE[job] = {'machine': 1, 'start': start, 'finish': finish}
        start = finish
    return SCHEDULE

gantt(JOBS, lifo(JOBS))
kpi(JOBS, lifo(JOBS))
```

Out[8]:

```
{'Fraction on Time': 0.42857142857142855,
 'Makespan': 30,
 'Max Pastdue': 25,
 'Number Pastdue': 4,
 'Number on Time': 3,
 'Sum of Pastdue': 47}
```





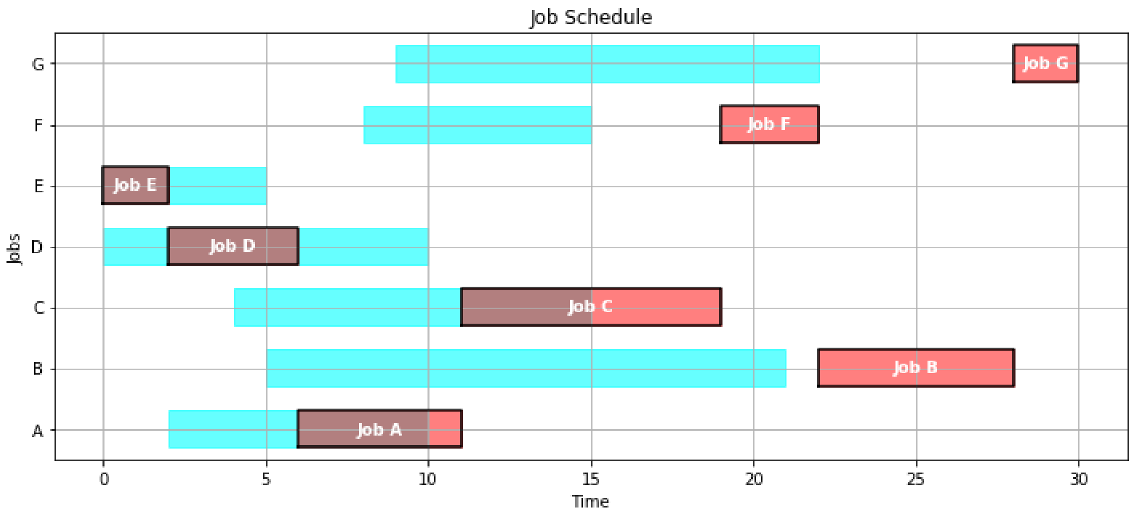
Earliest Due Date

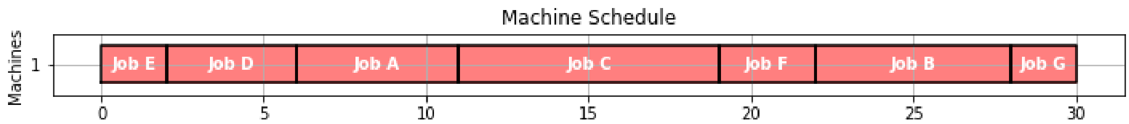
In [9]:

```
def edd(JOBS):
    unfinished_jobs = set(JOBS.keys())
    start = 0
    while len(unfinished_jobs) > 0:
        start = max(start, min(JOBS[job]['release'] for job in unfinished_jobs))
        edd = {job:JOBS[job]['due'] for job in unfinished_jobs if JOBS[job]['release']
<= start}
        job = min(edd, key=edd.get)
        finish = start + JOBS[job]['duration']
        unfinished_jobs.remove(job)
        SCHEDULE[job] = {'machine': 1, 'start': start, 'finish': finish}
        start = finish
    return SCHEDULE

gantt(JOBS, edd(JOBS))
kpi(JOBS, edd(JOBS))
```

Out[9]:

```
{'Fraction on Time': 0.2857142857142857,
 'Makespan': 30,
 'Max Pastdue': 8,
 'Number Pastdue': 5,
 'Number on Time': 2,
 'Sum of Pastdue': 27}
```





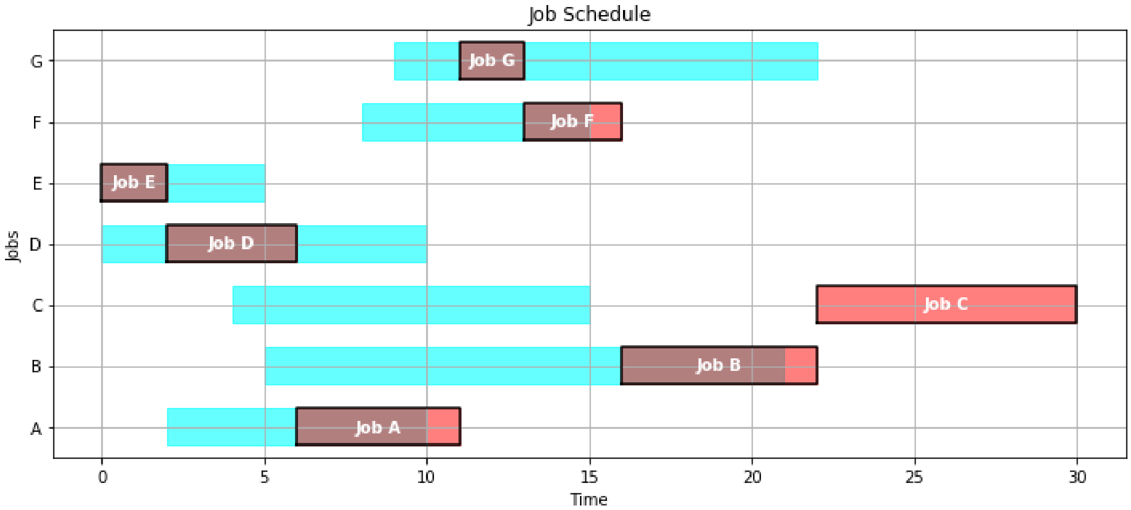
Shortest Processing Time

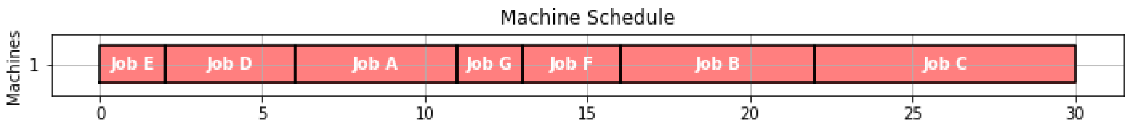
In [10]:

```
def spt(JOBS):
    unfinished_jobs = set(JOBS.keys())
    start = 0
    while len(unfinished_jobs) > 0:
        start = max(start, min(JOBS[job]['release'] for job in unfinished_jobs))
        spt = {job:JOBS[job]['duration'] for job in unfinished_jobs if JOBS[job]
['release'] <= start}
        job = min(spt, key=spt.get)
        finish = start + JOBS[job]['duration']
        unfinished_jobs.remove(job)
        SCHEDULE[job] = {'machine': 1, 'start': start, 'finish': finish}
        start = finish
    return SCHEDULE

gantt(JOBS, spt(JOBS))
kpi(JOBS, spt(JOBS))
```

Out[10]:

```
{'Fraction on Time': 0.42857142857142855,
 'Makespan': 30,
 'Max Pastdue': 15,
 'Number Pastdue': 4,
 'Number on Time': 3,
 'Sum of Pastdue': 18}
```





Modeling

Data

The data for this problem consists of a list of jobs. Each job is tagged with a unique ID along with numerical data giving the time at which the job will be released for machine processing, the expected duration, and the time at which it is due.

Symbol	Description
ID_j	Unique ID for task j
due_j	Due time for task j
$\mathrm{duration}_j$	Duration of task j
$\mathrm{release}_j$	Time task j becomes available for processing

Decision Variables

For a single machine, the essential decision variable is the start time at which the job begins processing.

Symbol	Description
start_j	Start of task j
$\mathrm{makespan}$	Time to complete *all* jobs.
$\mathrm{pastdue}_j$	Time by which task j is past due
early_j	Time by which task j is finished early

A job cannot start until it is released for processing

$$\mathrm{start}_j \geq \mathrm{release}_j$$

Once released for processing, we assume the processing continues until the job is finished. The finish time is compared to the due time, and the result stored in either the early or pastdue decision variables. These decision variables are needed to handle cases where it might not be possible to complete all jobs by the time they are due.

$$\begin{aligned}
\mathrm{start}_j + \mathrm{duration}_j + \mathrm{early}_j & = \mathrm{due}_j + \mathrm{pastdue}_j \\
\mathrm{early}_j & \geq 0 \\
\mathrm{pastdue}_j & \geq 0
\end{aligned}$$

Finally, we include a single decision variable measuring the overall makespan for all jobs.

$$\mathrm{start}_j + \mathrm{duration}_j \leq \mathrm{makespan}$$

The final set of constraints requires that, for any given pair of jobs j and k, that either j starts before k finishes, or k finishes before j starts. The boolean variable $y_{jk} = 1$ indicates j finishes before k starts, and is 0 for the opposing case. Note that we only need to consider cases $j > k$

$$\begin{aligned}
\mathrm{start}_i + \mathrm{duration}_i & \leq \mathrm{start}_j + M y_{i,j} \\
\mathrm{start}_j + \mathrm{duration}_j & \leq \mathrm{start}_i + M(1 - y_{i,j})
\end{aligned}$$

where M is a sufficiently large enough to assure the relaxed constraint is satisfied for all plausible values of the decision variables.

Pyomo Model

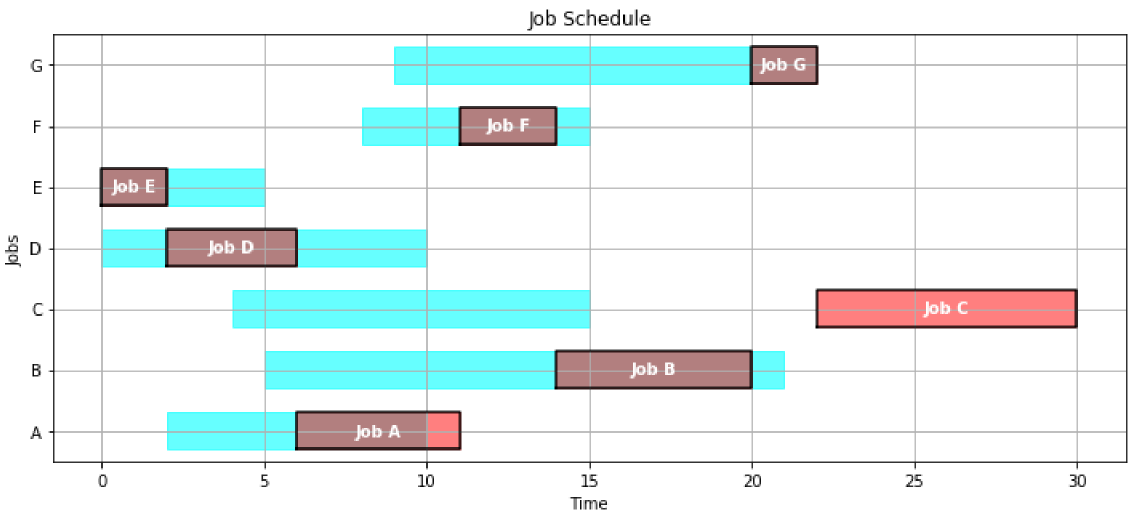
In [11]:

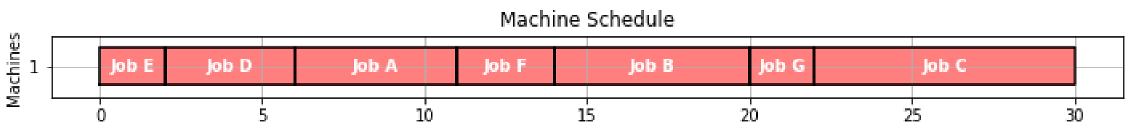
```
from pyomo.environ import *
from pyomo.gdp import *

def opt_schedule(JOBS):

    # create model
    m = ConcreteModel()

    # index set to simplify notation
    m.J = Set(initialize=JOBS.keys())
    m.PAIRS = Set(initialize = m.J * m.J, dimen=2, filter=lambda m, j, k : j < k)

    # upper bounds on how long it would take to process all jobs
    tmax = max([JOBS[j]['release'] for j in m.J]) + sum([JOBS[j]['duration'] for j in m.
J])

    # decision variables
    m.start      = Var(m.J, domain=NonNegativeReals, bounds=(0, tmax))
    m.pastdue    = Var(m.J, domain=NonNegativeReals, bounds=(0, tmax))
    m.early      = Var(m.J, domain=NonNegativeReals, bounds=(0, tmax))

    # additional decision variables for use in the objective
    m.makespan   = Var(domain=NonNegativeReals, bounds=(0, tmax))
    m.maxpastdue = Var(domain=NonNegativeReals, bounds=(0, tmax))
    m.ispastdue  = Var(m.J, domain=Binary)

    # objective function
    m.OBJ = Objective(expr = sum([m.pastdue[j] for j in m.J]), sense = minimize)

    # constraints
    m.c1 = Constraint(m.J, rule=lambda m, j: m.start[j] >= JOBS[j]['release'])
    m.c2 = Constraint(m.J, rule=lambda m, j:
            m.start[j] + JOBS[j]['duration'] + m.early[j] == JOBS[j]['due'] + m.pastdue[
j])
    m.c3 = Disjunction(m.PAIRS, rule=lambda m, j, k:
        [m.start[j] + JOBS[j]['duration'] <= m.start[k],
         m.start[k] + JOBS[k]['duration'] <= m.start[j]])

    m.c4 = Constraint(m.J, rule=lambda m, j: m.pastdue[j] <= m.maxpastdue)
    m.c5 = Constraint(m.J, rule=lambda m, j: m.start[j] + JOBS[j]['duration'] <= m.makes
pan)
    m.c6 = Constraint(m.J, rule=lambda m, j: m.pastdue[j] <= tmax*m.ispastdue[j])

    TransformationFactory('gdp.chull').apply_to(m)
    SolverFactory('cbc').solve(m).write()

    SCHEDULE = {}
    for j in m.J:
        SCHEDULE[j] = {'machine': 1, 'start': m.start[j](), 'finish': m.start[j]() + JOB
S[j]['duration']}

    return SCHEDULE

SCHEDULE = opt_schedule(JOBS)
gantt(JOBS, SCHEDULE)
kpi(JOBS, SCHEDULE)
```

```
# ==========================================================
# = Solver Results                                         =
# ==========================================================
# ----------------------------------------------------------
#   Problem Information
# ----------------------------------------------------------
Problem:
- Name: unknown
  Lower bound: 16.0
  Upper bound: 16.0
  Number of objectives: 1
  Number of constraints: 225
  Number of variables: 157
  Number of nonzeros: 533
  Sense: minimize
# ----------------------------------------------------------
#   Solver Information
# ----------------------------------------------------------
Solver:
- Status: ok
  User time: -1.0
  Termination condition: optimal
  Error rc: 0
  Time: 0.56770920753479
# ----------------------------------------------------------
#   Solution Information
# ----------------------------------------------------------
Solution:
- number of solutions: 0
  number of solutions displayed: 0
```

Out[11]:

```
{'Fraction on Time': 0.7142857142857143,
 'Makespan': 30.0,
 'Max Pastdue': 15.0,
 'Number Pastdue': 2,
 'Number on Time': 5,
 'Sum of Pastdue': 16.0}
```





Multiple Machines

The case of multiple machines requires a modest extension of model described above. Given a set M of machines, we introduce an additional decision binary variable $z_{j,m}$ indicating if job j has been assigned to machine m. The additional constraints

$$\sum_{m \in M} z_{j,m} = 1 \qquad \forall j$$

require each job to be assigned to exactly one machine for processing.

If both jobs j and k have been assigned to machine m, then the disjunctive ordering constraints must apply. This logic is equivalent to the following constraints for $j < k$.

$$\begin{aligned}
\text{start}_j + \text{duration}_j & \leq \text{start}_k + My_{j,k} + M(1 - z_{j,m}) + M(1 - z_{k,m}) \\
\text{start}_k + \text{duration}_k & \leq \text{start}_j + M(1 - y_{j,k}) + M(1 - z_{j,m}) + M(1 - z_{k,m}))
\end{aligned}$$

In [12]:

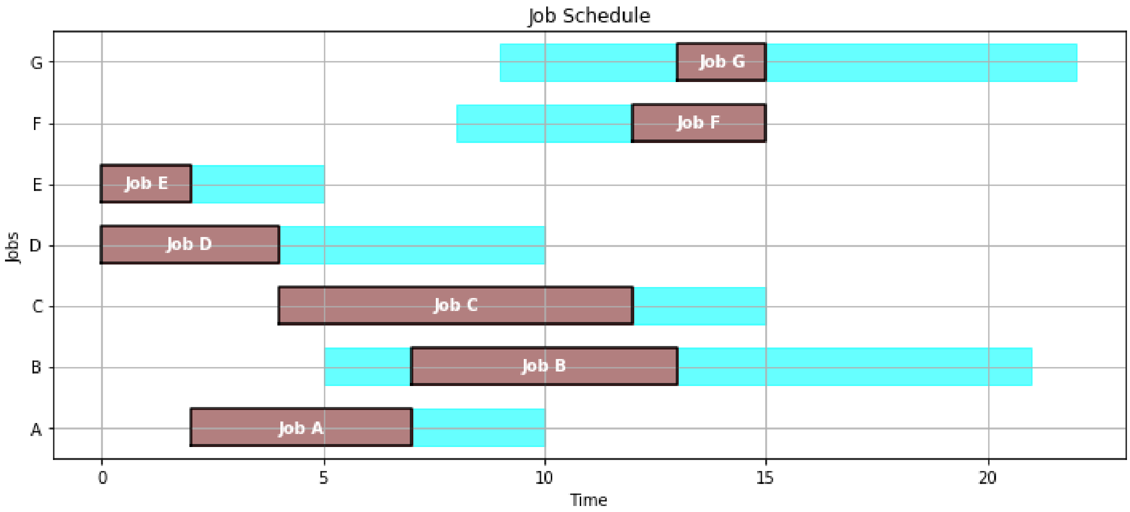
```
from pyomo.environ import *
from IPython.display import display
import pandas as pd

MACHINES = ['A','B']

def schedule_machines(JOBS, MACHINES):

    # create model
    m = ConcreteModel()

    # index set to simplify notation
    m.J = Set(initialize=JOBS.keys())
    m.M = Set(initialize=MACHINES)
    m.PAIRS = Set(initialize = m.J * m.J, dimen=2, filter=lambda m, j, k : j < k)

    # decision variables
    m.start      = Var(m.J, bounds=(0, 1000))
    m.makespan   = Var(domain=NonNegativeReals)
    m.pastdue    = Var(m.J, domain=NonNegativeReals)
    m.early      = Var(m.J, domain=NonNegativeReals)

    # additional decision variables for use in the objective
    m.ispastdue  = Var(m.J, domain=Binary)
    m.maxpastdue = Var(domain=NonNegativeReals)

    # for binary assignment of jobs to machines
    m.z = Var(m.J, m.M, domain=Binary)

    # for modeling disjunctive constraints
    m.y = Var(m.PAIRS, domain=Binary)
    BigM = max([JOBS[j]['release'] for j in m.J]) + sum([JOBS[j]['duration'] for j in m.
J])

    m.OBJ = Objective(expr = sum(m.pastdue[j] for j in m.J) + m.makespan - sum(m.early[j
] for j in m.J), sense = minimize)

    m.c1 = Constraint(m.J, rule=lambda m, j:
            m.start[j] >= JOBS[j]['release'])
    m.c2 = Constraint(m.J, rule=lambda m, j:
            m.start[j] + JOBS[j]['duration'] + m.early[j] == JOBS[j]['due'] + m.pastdue[
j])
    m.c3 = Constraint(m.J, rule=lambda m, j:
            sum(m.z[j,mach] for mach in m.M) == 1)
    m.c4 = Constraint(m.J, rule=lambda m, j:
            m.pastdue[j] <= BigM*m.ispastdue[j])
    m.c5 = Constraint(m.J, rule=lambda m, j:
            m.pastdue[j] <= m.maxpastdue)
```

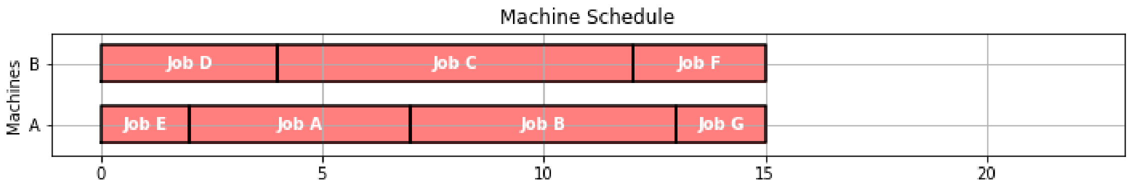
```
            m.pastdue[j] <= m.maxpastdue)
    m.c6 = Constraint(m.J, rule=lambda m, j:
            m.start[j] + JOBS[j]['duration'] <= m.makespan)
    m.d1 = Constraint(m.M, m.PAIRS, rule = lambda m, mach, j, k:
            m.start[j] + JOBS[j]['duration'] <= m.start[k] + BigM*(m.y[j,k] + (1-m.z[j,ma
ch]) + (1-m.z[k,mach])))
    m.d2 = Constraint(m.M, m.PAIRS, rule = lambda m, mach, j, k:
            m.start[k] + JOBS[k]['duration'] <= m.start[j] + BigM*((1-m.y[j,k]) + (1-m.z
j,mach]) + (1-m.z[k,mach])))

    SolverFactory('cbc').solve(m).write()

    SCHEDULE = {}
    for j in m.J:
        SCHEDULE[j] = {
            'start': m.start[j](),
            'finish': m.start[j]() + JOBS[j]['duration'],
            'machine': [mach for mach in MACHINES if m.z[j,mach]][0]
        }

    return SCHEDULE

SCHEDULE = schedule_machines(JOBS,MACHINES)
gantt(JOBS, SCHEDULE)
kpi(JOBS, SCHEDULE)
```

```
# ==========================================================
# = Solver Results                                         =
# ==========================================================
# ----------------------------------------------------------
#   Problem Information
# ----------------------------------------------------------
Problem:
- Name: unknown
  Lower bound: -15.0
  Upper bound: -15.0
  Number of objectives: 1
  Number of constraints: 127
  Number of variables: 66
  Number of nonzeros: 505
  Sense: minimize
# ----------------------------------------------------------
#   Solver Information
# ----------------------------------------------------------
Solver:
- Status: ok
  User time: -1.0
  Termination condition: optimal
  Error rc: 0
  Time: 0.5222530364990234
# ----------------------------------------------------------
#   Solution Information
# ----------------------------------------------------------
Solution:
- number of solutions: 0
  number of solutions displayed: 0
```

Out[12]:

```
{'Fraction on Time': 1.0,
 'Makespan': 15.0,
 'Max Pastdue': 0,
 'Number Pastdue': 0,
 'Number on Time': 7,
 'Sum of Pastdue': 0}
```


Job Schedule
G
F
E
D
C
B
A
Jobs
Job G
Job F
Job E
Job D
Job C
Job B
Job A
0
5
10
15
20
Time


Machine Schedule
B
A
Machines
Job D
Job C
Job F
Job E
Job A
Job B
Job G
0
5
10
15
20

```
In [ ]:
```