Cyberdyne runs a production line with 8 machines M1 to M8. The products that go on this line require processing by all 8 machines in sequence (i.e. starting on M1 then processing on M2 and onwards through the rest of the machines until they finish processing on M8) but the time that each product requires on each machines varies. The company has received an order for 30 jobs (see table 1 - an Excel copy is also available on Blackboard).

0b. import liblaries

In [144]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from pulp import *
```

0a. load data

In [145]:
```
jobs = pd.read_csv('Jobs.csv')
```

In [146]:
```
jobs.set_index('Job')
```

Out[146]:

Job	Time on M_1	Time on M_2	Time on M_3	Time on M_4	Time on M_5	Time on M_6	Time on M_7	Time on M_8
1	9	3	6	10	8	5	3	2
2	2	4	4	1	1	5	5	3
3	5	9	8	2	9	8	1	10
4	3	5	10	6	8	9	1	2
5	2	2	7	9	7	8	8	1
6	1	5	7	6	6	5	2	10
7	8	1	2	1	6	6	4	2
8	9	3	2	4	6	3	6	10
9	4	4	2	3	7	7	10	8
10	1	4	3	2	3	10	9	6
11	2	8	8	8	3	4	7	10
12	6	8	10	8	10	3	6	7
13	6	2	7	1	10	1	10	7
14	9	7	5	4	8	1	2	2
15	3	1	10	4	10	2	8	3
16	9	2	1	1	4	3	7	4
17	7	1	7	6	6	3	6	7
18	8	5	5	10	5	9	2	4
19	6	1	6	1	1	7	5	2
20	7	1	5	9	7	1	8	2
21	2	8	5	2	3	4	6	10
22	1	8	9	9	7	1	10	8
23	10	3	3	10	6	7	2	4
24	7	1	6	2	4	4	9	2

Job	Time on M_1	Time on M_2	Time on M_3	Time on M_4	Time on M_5	Time on M_6	Time on M_7	Time on M_8
25	5	8	9	6	10	2	5	4
26	9	3	5	1	8	6	9	1
27	1	9	5	9	7	1	7	1
28	9	8	3	3	4	8	10	4
29	2	4	7	3	3	1	10	1
30	6	10	9	1	10	1	4	5

1. Create a computer tool to calculate the time of the end of the final job on the las machine, given a sorted list of jobs. Assume that jobs cannot overtake one another in the line.

In [147]:
```
sorted_list_of_jobs = range(0,30)
```

In [215]:
```
def total_time(list_of_jobs):


    # [0,3,1,2]

    time_passed = 0
    n = 8 # number of machines

    all_jobs_finished = False
    job_stage = np.zeros(len(list_of_jobs), dtype=int)
    job_time_left = np.zeros(len(list_of_jobs), dtype=int)
    is_job_busy = np.zeros(len(list_of_jobs), dtype=bool)


    machine_time_left = np.zeros(n, dtype=int)  # specifies how much left for the job
    machine_job_number = np.zeros(n, dtype=int)  # specifies which job INDEX from list_of_jobs is being executed
    is_machine_busy = np.zeros(8, dtype=bool)

    while not all_jobs_finished:


        # iterate through machines to fill them with jobs
        for ind in range(0,n):

            # add appropraite amount of time to machines_time_left
            if not is_machine_busy[ind] and machine_job_number[ind] != -1:

                # get ind of next job
                next_job = machine_job_number[ind]
                # check wether next_job is busy
                if not is_job_busy[next_job] and job_stage[next_job] == ind:

                    job_stage[next_job] += 1
                    is_job_busy[next_job] = True
                    machine_job_number[ind] += 1

                    job_time_left[next_job] += jobs.iloc[list_of_jobs[next_job],ind+1]
                    machine_time_left[ind] += jobs.iloc[list_of_jobs[next_job],ind+1]
#                     print("Machine: " + str(ind) + " - " + str(machine_job_number[ind]))

                    # if machine took its last item then tell it not to take any more
```

```
                    if machine_job_number[ind] >= len(list_of_jobs):
                        machine_job_number[ind] = -1

#         print("--------------")  # this indicates that one unit time passed
        # decrement all machines in machines_time_left by 1
        time_passed += 1
        machine_time_left = np.where(machine_time_left < 1, machine_time_left, machine_time_left - 1)
        job_time_left = np.where(job_time_left < 1, job_time_left, job_time_left - 1)

        # update which machines are ready to take new jobs
        is_machine_busy = [machine != 0 for machine in machine_time_left]
        is_job_busy = [job != 0 for job in job_time_left]


        # all jobs finished
        all_jobs_finished = all(job == 8 and busy == False for job in job_stage for busy in is_job_busy)

        # stop if your program enters infinite loop
        if (time_passed > 400):
            all_jobs_finished = True

    return time_passed
```

In [216]:

```
time = total_time(list(sorted_list_of_jobs))
print("Time for this sequence: " + str(time))
```

```
Time for this sequence: 256
```

2. Use the sum of time that the job requires on all machines as a heuristic. Order the jobs in an ascending and descending order and calculate the total makespan of the resulting job sequence (the time that it takes to complete all 8 processes on all jobs).

In [217]:
```
jobs_2 = jobs.copy()
jobs_2['total_time'] = jobs_2[jobs_2.columns[1:]].sum(axis=1)

jobs_2_ascending = jobs_2.sort_values(by ='total_time', kind='mergesort')
jobs_2_descending = jobs_2.sort_values(by ='total_time',ascending=False, kind='mergesort')

jobs_heuristic_ascending = jobs_2_ascending.index.to_series().values
jobs_heuristic_descending = jobs_2_descending.index.to_series().values
```

In [218]:
```
time_ascending = total_time(jobs_heuristic_ascending)
time_descending = total_time(jobs_heuristic_descending)

print("Time for ASCENDING sequence: " + str(time_ascending))
print("Time for DESCENDING sequence: " + str(time_descending))
```

```
Time for ASCENDING sequence: 262
Time for DESCENDING sequence: 267
```

3. Use the sum of the time of the first four jobs and the sum of the time of the last four jobs and use Johnson’s rule 1 to schedule the jobs. Compute the makespan and compare with the previous ones.

In [224]:
```
jobs_3 = jobs.copy()
jobs_3['first_four_m'] = jobs_3[jobs_3.columns[1:5]].sum(axis=1)
jobs_3['last_four_m'] = jobs_3[jobs_3.columns[5:9]].sum(axis=1)
jobs_3['lowest'] = [min(first, last) for first, last in zip(jobs_3['first_four_m'], jobs_3['last_four_m'])]

jobs_3_johnsons = jobs_3.sort_values(by ='lowest', kind='mergesort')
```

create sequence of jobs according to johnson's rule

In [226]:
```
jobs_johnsons = np.zeros(30, dtype=int)
first_ind = 0
last_ind = 29

for index, row in jobs_3_johnsons.iterrows():
    if row['lowest'] == row['first_four_m']:
        jobs_johnsons[first_ind] = index
        first_ind += 1
    if row['lowest'] == row['last_four_m']:
        jobs_johnsons[last_ind] = index
        last_ind -= 1
```

In [227]:
```
time_johnsons = total_time(jobs_johnsons)

print("Time for JOHNSON'S sequence: " + str(time_johnsons))
```

```
Time for JOHNSON'S sequence: 243
```

4. Use a metaheuristic method to find a good schedule and compare with the results of the previous sections

Genetic algorithm

1) initialisation

In [260]:
```
# an initial population of feasible trial solutions - 10 initial cases
list_of_sequences_original = [list(range(0,30)),
                     jobs_heuristic_ascending,
                     jobs_heuristic_descending,
                     jobs_johnsons,
                    np.random.permutation(list(range(0,30))),
                    np.random.permutation(list(range(0,30))),
                    np.random.permutation(list(range(0,30))),
                    np.random.permutation(list(range(0,30))),
                    np.random.permutation(list(range(0,30))),
                    np.random.permutation(list(range(0,30)))]

list_of_sequences = list_of_sequences_original.copy()  # this gets overwriten later, so keep original safe in case we
 want to run it again
```

2) iteration

In [261]:
```
import random
random.seed(a=100)  # set seed value, such that your psudo-random generated numbers are reproducible


# function used to create child from two parents
def breed(parent1, parent2):
    child = []
    childP1 = []
    childP2 = []

    geneA = int(random.random() * len(parent1))
    geneB = int(random.random() * len(parent1))

    startGene = min(geneA, geneB)
    endGene = max(geneA, geneB)

    for i in range(startGene, endGene):
        childP1.append(parent1[i])

    childP2 = [item for item in parent2 if item not in childP1]

    child = childP1 + childP2
    return child


def mutate(individual, mutationRate):
    for swapped in range(len(individual)):
        if(random.random() < mutationRate):
            swapWith = int(random.random() * len(individual))

            job1 = individual[swapped]
            job2 = individual[swapWith]

            individual[swapped] = job2
            individual[swapWith] = job1
    return individual


# above functions are adapted from:
# https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35
```

In [262]:
```
results = np.zeros(len(list_of_sequences), dtype=int)
n = 0  # variabel for termination condition
min_time_old = None
min_time = None
iteration = 1

while n < 10:
    print("------ ITERATION " + str(iteration) + " ------")
    iteration += 1

    i = 0
    # update termination condition
    if min_time == min_time_old:
        n += 1
    else:
        n = 0
    min_time_old = min_time



    # ASSSES FITNESS

    for seq in list_of_sequences:
        results[i] = total_time(seq)
        print(str(i) + ": " + str(results[i]))
        i += 1
    results_sorted = np.sort(results)



    # SELECT PARENTS

    # choose 4 parents from 5 best samples
    parents_top = np.random.choice(results_sorted[:5], size=4, replace=False)
    # choose 2 parents from other 5 samples
    parents_bottom = np.random.choice(results_sorted[5:], size=2, replace=False)
    # combine all 6 parents into a single array
    parents = np.concatenate((parents_top, parents_bottom))

    # form 3 random couples
    parents_perm = np.random.permutation(parents)
```

```
    couples = [[i,j] for i,j in zip(parents_perm[:3], parents_perm[3:])]

    # get sequences from fitness values
    couples_sequences = [ [list_of_sequences[results.tolist().index(x)], list_of_sequences[results.tolist().index(y)]
] for x, y in couples]



    # GET CHILDREN

    children_1 = [breed(p1, p2) for p1, p2 in couples_sequences] # get 1st children from each couple
    children_2 = [breed(p1, p2) for p1, p2 in couples_sequences] # get 2nd children from each couple
    children = np.concatenate((children_1, children_2))



    # MUTATE CHILDREN
    for i in range(len(children)):
        children[i] = mutate(children[i], 0.2)



    # GET NEW POPULATION (get new -> list_of_sequences)

    list_of_sequences = np.concatenate(([list_of_sequences[results.tolist().index(x)] for x in results_sorted[:4]], ch
ildren))
    min_time = results_sorted[0]



print()
print("Best time: " + str(min_time))
print("The best sequence is: " + str(list_of_sequences[results.tolist().index(min_time)]))
```

```
------ ITERATION 1 ------
0: 256
1: 262
2: 267
3: 243
4: 243
5: 250
6: 265
7: 245
8: 258
9: 271
------ ITERATION 2 ------
0: 243
1: 243
2: 245
3: 250
4: 247
5: 252
6: 241
7: 261
8: 253
9: 258
------ ITERATION 3 ------
0: 241
1: 243
2: 243
3: 245
4: 253
5: 256
6: 240
7: 247
8: 274
9: 247
------ ITERATION 4 ------
0: 240
1: 241
2: 243
3: 243
4: 254
5: 266
6: 249
```

```
7: 238
8: 262
9: 253
------ ITERATION 5 ------
0: 238
1: 240
2: 241
3: 243
4: 240
5: 252
6: 242
7: 252
8: 249
9: 241
------ ITERATION 6 ------
0: 238
1: 240
2: 240
3: 241
4: 241
5: 238
6: 245
7: 245
8: 245
9: 237
------ ITERATION 7 ------
0: 237
1: 238
2: 238
3: 240
4: 241
5: 234
6: 263
7: 242
8: 264
9: 226
------ ITERATION 8 ------
0: 226
1: 234
2: 237
3: 238
4: 233
```

```
5: 242
6: 254
7: 258
8: 252
9: 247
------ ITERATION 9 ------
0: 226
1: 233
2: 234
3: 237
4: 252
5: 236
6: 246
7: 260
8: 239
9: 258
------ ITERATION 10 ------
0: 226
1: 233
2: 234
3: 236
4: 259
5: 229
6: 242
7: 256
8: 244
9: 246
------ ITERATION 11 ------
0: 226
1: 229
2: 233
3: 234
4: 253
5: 252
6: 262
7: 242
8: 233
9: 247
------ ITERATION 12 ------
0: 226
1: 229
2: 233
```

```
3: 233
4: 259
5: 231
6: 236
7: 238
8: 253
9: 243
------ ITERATION 13 ------
0: 226
1: 229
2: 231
3: 233
4: 237
5: 243
6: 248
7: 248
8: 256
9: 258
------ ITERATION 14 ------
0: 226
1: 229
2: 231
3: 233
4: 230
5: 276
6: 247
7: 251
8: 238
9: 258
------ ITERATION 15 ------
0: 226
1: 229
2: 230
3: 231
4: 258
5: 236
6: 254
7: 248
8: 251
9: 248
------ ITERATION 16 ------
0: 226
```

```
1: 229
2: 230
3: 231
4: 257
5: 273
6: 233
7: 255
8: 258
9: 242
------ ITERATION 17 ------
0: 226
1: 229
2: 230
3: 231
4: 258
5: 260
6: 246
7: 247
8: 249
9: 250
------ ITERATION 18 ------
0: 226
1: 229
2: 230
3: 231
4: 238
5: 247
6: 257
7: 249
8: 253
9: 245

Best time: 226
The best sequence is: [23  8  1  9 12  4 11 14 17 27  5 15 20 24  6 21 19 18  7 28 25  2 10  0
 29  3 22 26 16 13]
```

Tabu algorithm

In [264]:
```
# to be added
```

Particle swarm optimisation

In [263]:
```
# to be added
```

5. Create a mathematical programming model for the problem and try to optimally solve it. If completed, compare the solution to the previous sections.

In [265]:
```
# to be added
```

In [ ]:
```

```