Stage 1:

```
'''
Variables:
    machines_charact
        type: dictionary
        structure: {Machine Name : {"Capacity" : capacity, "Cooldown time" :
cooldown time}}
    machines_list - contains all the names of the machines
        type: list
    parts_processing
        type: dictionary
        structure: {Part Name : {"Item" : no. of items to process, First machine
: time to spend on that machine, Second machine : time to spend on that machine}}
    parts_list - contains all the names of the parts
        type: list
    aux - used as placeholder
        type: string
    index - used for indexing
        type: int
'''


'''
#For testing the code without manually entering the file path
input_file = r'D:\facultate\2023_Internship_Challenge_Software\Input_One.txt'
f = open(input_file, "r")
'''

#For testing with manual entry of the file path
while True:
    try:
        input_file = input("Please insert the file path: ")
        f = open(input_file, "r")
        break
    except FileNotFoundError:
        print("The path given doesn't return any document. Please check the
location of your file or the format of the path and try again")
    except OSError:
        print("Invalid path.")
    except:
        print("Invalid argument.")

machines_charact = {}
machines_list = []
```

```
parts_processing = {}
parts_list = []

for line in f:
    '''
    The loop goes through each line of the file, looking for key phrases that
mark the sections of the document
    '''
    if "Available machines" in line:
        '''
        If "Available machines" is found, then for the next lines extract the
names of the machines and store them in machines_charact
        and machine_lists.
        Based on the formatting in the file, the name of the machine is extracted
by splitting each row at ". ", taking the part after
        the splitting string, eliminating the automatic new line character at the
end and saving it in the placeholder variable aux.
        Then, through aux, each name of the machine is added as a key in the
machines_charact dict and as an element to the
        machines_list list.
        When the splitting string ". " is no longer found in the line, then the
loop exits.
        '''
        for line in f:
            if ". " in line:
                aux = line.split(". ")[1].strip("\n")
                machines_charact[aux] = ""
                machines_list.append(aux)
            else:
                if "#" in line:
                    continue
                else:
                    break
    elif "Machine features" in line:
        '''
        If "Machine features" is found, then from the next line onward the
presence of a feature is checked by the presence of the
        character ":".
        Based on the structure of the document, the machines are put in order, so
using an index starting from 0, the machines_list
        is used to add the characteristics into the machines_charact. The index
increases when only one ":" is found, because all
        machines have only 2 features. When the index is equal to the number of
machines, the loop exits.
```

```
        The features are split from their order number by the splitting string "-
". Then the name of the feature and the value is split
        using the splitting string ": ".
        '''
        index = 0
        for line in f:
            if ":" in line:
                aux = line.split("- ")[1].strip("\n")
                machines_charact[machines_list[index]] = {aux.split(": ")[0] :
aux.split(": ")[1]}
            if line.count(":") == 1:
                index+=1
            if index == len(machines_charact):
                break
    elif "Part list" in line:
        '''
        If "Part list" is found, then from the next line onward if it finds a ".
" then considers that the line contains a part name.
        When ". " is not found and a "#" is not present, the loop breaks
        Aux and key are used as placeholder values in order to contain the number
of items and the name of the item respectively.
        The splitting string ". " is used to separate the number order and " - "
to separate the name from the item.
        The name of the item is saved in both the parts_processing as key and
parts_list, and the number of items is saved for each
        item name key as a dictionary with the structure {"Item": no.}
        '''
        for line in f:
            if ". " in line:
                aux = line.split(". ")[1]
                key = aux.split(" - ")[0]
                value = aux.split(" - ")[1]
                value = value.split(" item")[0].strip("\n")
                dict = "Item"

                parts_processing[key] = {dict : value}
                parts_list.append(key)
            else:
                if "#" in line:
                    continue
                else:
                    break
    elif "Part operations" in line:
        '''
```

```
        If the phrase "Part operations" is found, then the next lines checked for
the piece and its operations.
        The start of the operations list for each part is identified by looking
for two ":". When it's found, the index marks
        the number of the piece the current line is at (which is why index starts
from -1). Then, for each part, it goes throught
        the list of machines and if it is found, then it is added to the
dictionary with its processing time.
        '''
        index_part = -1
        for line in f:
            if line.count(":") == 2:
                index_part += 1
            for machine in machines_list:
                if machine in line:
                    parts_processing[parts_list[index_part]][machine] =
line.split(machine)[1].strip("\n").strip(": ")

'''
Formatting the console output
'''
print("Parts")
for part in parts_processing.keys():
    print(part)
    for key in parts_processing[part].keys():
        print('{:<10}{:<20}{}'.format("", key, parts_processing[part][key]))
```

Stage 2:

```
def get_int(line):
    i = 0
    for char in line:
        if char.isdigit():
            i = i * 10 + int(char)
    return i

def time_format(seconds):
    result = "{:02d}:{:02d}:{:02d}".format(seconds // 3600, (seconds % 3600) //
60, (seconds % 60))
    return result

#For testing the code without manually entering the file path
def auto_input():
    input_file = r'D:\facultate\2023_Internship_Challenge_Software\Input_One.txt'
    f = open(input_file, "r")
    return f


#For testing with manual entry of the file path
def manual_input():
    while True:
        try:
            input_file = input("Please insert the file path: ")
            f = open(input_file, "r")
            break
        except FileNotFoundError:
            print("The path given doesn't return any document. Please check the
location of your file or the format of the path and try again")
        except OSError:
            print("Invalid path.")
        except:
            print("Invalid argument.")
    return f

f = manual_input()

'''
We assume that all the part operations are written in the order that the machines
were presented.
The order in which the parts are machined is first based on the longest machining
time, then the number
of machines in part operations. This order is saved as a list in parts_order.
```

```
Variables:
machines_charact
        type: dictionary
        structure: {Machine Name : {"Capacity" : capacity, "Cooldown time" :
cooldown time}}
machines_list - contains all the names of the machines
    type: list
parts_processing
    type: dictionary
    structure: {Part Name : {"Item" : no. of items to process, First machine :
time to spend on that machine, Second machine : time to spend on that machine}}
parts_list - contains all the names of the parts
    type: list
c_part_schedule - the machining schedule of the part currenty being processed
    type: dictionary
    structure: {Machine name : [machining start, machining end]}
parts_schedule - dictionary in which the entire machining schedule is saved
    typr: dict()
    structure: {Part name : {Machine name : [machining start, machining end]}}
aux - used as placeholder
    type: string
index - used for indexing
    type: int
'''

machines_charact = {}
machines_list = []

parts_processing = {}
parts_list = []

c_part_schedule = dict(zip(machines_list, map(lambda x: [], machines_list)))
parts_schedule = dict()
cooldown_times = dict()

for line in f:
    '''
    Variables:
    c_machine = current machine
    c_part = current part
    '''
    if "Available machines" in line:
        for line in f:
            if ". " in line:
                aux = line.split(". ")[1].strip("\n")
```

```
                machines_charact[aux] = ""
                machines_list.append(aux)
            else:
                if "#" in line:
                    continue
                else:
                    break
    elif "Machine features" in line:
        index = 0
        for line in f:
            if line.count(":") == 2:
                c_machine = machines_list[index]
                machines_charact[c_machine] = []
            if ":" in line:
                if " one " in line:
                    machines_charact[c_machine].append(1)
                elif "second" in line:
                    sec = get_int(line)
                    machines_charact[c_machine].append(sec)
                elif "none" in line:
                    machines_charact[c_machine].append(0)
            if line.count(":") == 1:
                index+=1
            if index == len(machines_charact):
                break
    elif "Part list" in line:
        for line in f:
            if ". " in line:
                aux = line.split(". ")[1]
                c_part = aux.split(" - ")[0]
                value = aux.split(" - ")[1]
                value = value.split(" item")[0].strip("\n")

                parts_processing[c_part] = list()
                parts_processing[c_part].append(get_int(value))
                parts_list.append(c_part)
            else:
                if "#" in line:
                    continue
                else:
                    break
    elif "Part operations" in line:
        index_part = -1
        index_machine = 0
        for line in f:
```

```
            if line.count(":") == 2:
                index_part += 1
                index_machine = 0
            for machine in machines_list:
                if machine in line:
                    c_part = parts_list[index_part]
                    aux = get_int(line.split("-")[1])
                    parts_processing[c_part].append(aux)
                    line = f.readline()
                else:
                    parts_processing[c_part].append(0)


parts_processing_copy = parts_processing.copy()

#For each part, we check how many items there need to be and add them as
different parts to part_order
for key in parts_processing_copy.keys():
    if parts_processing[key][0] > 1:
        new_key = key + "+"
        parts_processing[new_key] = parts_processing[key][1:]
    parts_processing[key] = parts_processing[key][1:]

parts_order = sorted(parts_processing.values(), key = lambda x: (0 in x, max(x)),
reverse=True)

for list in parts_order:
    for key, value in parts_processing.items():
        if value == list:
            parts_schedule[key] = None

#adding to memory all the machines with cooldown times and the cooldown time
for machine in machines_charact.keys():
    if machines_charact[machine][1] > 0:
        cooldown_times[machine] = machines_charact[machine][1]

machine_schedule = dict((key, None) for key in machines_list)

for part in parts_order:
    '''
    Based on the order in parts_order, each part is taken, its machining interval
for each machine
    is computed and added to part_schedule. For each part, the list of machines
machine_list is
    iterated.
```

```
    Computing the machining time is done by
    Variable:
    c_part_schedule - holds the machining schedule for the current part
        type: dict
        structure: {Machine name : [start processing time, end processing time]}
    last_time - holds the value of the time at which the last machining operation
ended for the
                current part
        type: int
    '''
    last_time = 0
    c_part_schedule = dict(zip(machines_list, map(lambda x: [0, 0],
machines_list)))
    if parts_order.index(part) == 0:
        '''
        For the first machining of the first part in order, the start time is 0
and the end time
        is the machining time for the first machining operation, and after the
start time is the end time
        of the previus machining operation and the end time is the start time +
the machining time
        last_time takes the value of the end time of this machining operation
        '''
        for machine, parts_time in zip(c_part_schedule.keys(), part):
            if parts_time != 0:
                c_part_schedule[machine][0] = last_time    #machining start time
                c_part_schedule[machine][1] = parts_time + last_time   #machining
end time
                last_time = c_part_schedule[machine][1]
    else:
        '''
        For all the other parts, for each machine for which the machining time is
not 0 we compute an intitial interval by taking the
        time at which the previous machining ended and adding to it the machining
time.
        If its the first machining, then the starting time is 0.
        After that, we check for each machining time if its outside the working
intervals of its machine. If it is, then it is saved in
        parts_schedule. If not, then all the intervals of that part are pushed up
by 50, as to not lose continuity, and the check is done again
        until all the intervals are in a free slot.
        After adding the parts schedule to parts_schedule, the cooling time is
added to the intervals.
        '''
        wrong = 1
```

```
        max = 0
        for key in parts_schedule.keys():
            if parts_schedule[key] != None:
                m = machines_list[0]
                if max < parts_schedule[key][m][1]:
                    max = parts_schedule[key][m][1]
        while wrong == 1:
            wrong = 0
            for machine, parts_time in zip(c_part_schedule.keys(), part):
                if parts_time != 0:
                    if part.index(parts_time) == 0:
                        c_part_schedule[machine][0] = max
                        c_part_schedule[machine][1] = max + parts_time
                    else:
                        c_part_schedule[machine][1] = parts_time + last_time
                        c_part_schedule[machine][0] = last_time
                    last_time = c_part_schedule[machine][1]
            for key in parts_list:
                for machine in machines_list:
                    if parts_schedule[key] != None and
parts_schedule[key][machine] != [0, 0] and c_part_schedule[machine] != [0, 0]:
                        if parts_schedule[key][machine][0] >=
c_part_schedule[machine][1] or parts_schedule[key][machine][1] <=
c_part_schedule[machine][0]:
                            continue
                        else:
                            wrong = 1
                            max += 50
                            break
                if wrong == 1:
                    break
    for key, val in parts_processing.items():
        if val == part:
            parts_schedule[key] = c_part_schedule
            break
    del parts_processing[key]
    if cooldown_times:
        for machine in cooldown_times.keys():
            c_part_schedule[machine][1] += cooldown_times[machine]


#transforming all the intervals into the 00:00:00 format
for value in parts_schedule.values():
    for interval in value.values():
        interval[0] = time_format(interval[0])
```

```
        interval[1] = time_format(interval[1])

final_time = 0

#printing final schedule
for part_name in parts_list:
    print('{:<10}{}'.format("", part_name))
    index = 1
    for part, process in parts_schedule.items():
        if part_name in part:
            print(index, ".")
            for machine, interval in process.items():
                print('{:<30}{}'.format(machine, interval))
                final_time = interval[1]
            index += 1

print("Total process time", final_time)
```

Stage 3:

```
from library import Machine
from library import Part
import library as l

f = l.manual_input()

l.processingTextFile(f)

'''
Since continuity is no longer a concern, first we schedule the first machining
operation
based on the longest machining time of each part. After this, we loop the parts
order and
schedule the part with the lowest machining time each time. When a machining
operation is done, it's
deleted from the part's operations list.
'''

for part in Part.getPartList():
    '''
    For each part, we check how many items there need to be and make new objects
of class Part
    '''
    if part.getItems() > 1:
        part.makeCopy(part)

#ordering all the parts by the longest machining time
Part.orderPartList()

last_time = 0

parts_list = Part.copyPartList()

for part in parts_list:
    '''
    Scheduling the first operation for each part. It is taken into consideration
that each part has the same first machining operation.
    '''
    first_machine = Machine.getMachineList()[0]
    first_machine_name = first_machine.getName()
    free_machines = first_machine.getCapacity()
    machining_time = part.getOperation(first_machine_name)
    interval = [last_time, last_time + machining_time]
```

```
    part.setSchedule(first_machine_name, interval)
    first_machine_busy_times = first_machine.getBusyTimes()
    if [last_time, last_time + machining_time] not in first_machine_busy_times:
        cooldown = first_machine.getCooldown()
        period = [last_time, last_time + machining_time + cooldown]
        first_machine.addToBusyTimes(period)
    last_time += machining_time
    part.delOperation(first_machine_name)

#Sorting the part list by the longest machining time of each machine
parts_list = sorted(Part.getPartList(), key = lambda part: part.longestOpTime(),
reverse = True)

while len(parts_list) != 0:
    '''
    Scheduling all the other operations in the order of which they are finishing
their last operation.
    The initial interval for the next machining is starting from the finish time
of the last operation and
    ending by adding the machining time.
    Then the interval is checked against the busy times of the machine. If it
doesn't intersect with an already occupied
    interval then it's added to the parts schedule and the machine's busy times.
It it does, we add 10 to both ends of the
    interval and compare again.
    Completed operations are deleted from the list.
    Finished parts are deleted from the list.
    '''
    min = 9999999999999999
    max = 0
    for part in parts_list:
        for interval in part.getSchedule():
            machining_end = interval[1]
            if max < machining_end:
                max = machining_end
        if min > max:
            min = max
            c_part = part
    c_machine_name = c_part.getOperationsMachines()[0]
    next_machine = next((machine for machine in Machine.getMachineList() if
machine.getName() == c_machine_name), None)
    next_machine_name = next_machine.getName()
    t_start = min #previous end time
    t_working = c_part.getOperation(next_machine_name)
    t_end = t_start + t_working
```

```
    inc = 0
    wrong = 1
    while wrong == 1:
        wrong = 0
        t_start += inc
        t_end += inc
        next_machine.sortBusyTimes()
        if next_machine.getBusyTimesLen() > 0:
            free_machines = next_machine.getCapacity()
            busy_times = next_machine.getBusyTimes()
            for interval in busy_times:
                if t_start in range(interval[0], interval[1]) or t_end in
range(interval[0], interval[1]):
                    if free_machines == 1:
                        wrong = 1
                        inc = 50
                        break
                    else:
                        free_machines -= 1
    busy_times = next_machine.getBusyTimes()
    if [t_start, t_end] not in busy_times:
        cooldown = next_machine.getCooldown()
        period = [t_start, t_end + cooldown]
        next_machine.addToBusyTimes(period)
    c_part.setSchedule(next_machine_name, [t_start, t_end])
    c_part.delOperation(next_machine_name)
    for item in parts_list:
        if item.noOfOperations() == 0:
            del parts_list[parts_list.index(item)]


#Printing schedule
all_parts = Part.getPartList()
max = 0 #part final time
all_max = 0 #overall final one
for part in all_parts:
    print('{:<10}{}'.format("", part.getName()))
    max = 0
    for interval in part.getSchedule():
        machining_end = interval[1]
        if max < machining_end:
            max = machining_end
    part_schedule = part.getSchedule()
    for interval in part_schedule:
        interval[0] = l.time_format(interval[0])
```

```
        interval[1] = l.time_format(interval[1])
    print('\n'.join(map(lambda item1, item2: f'{item1}\n    {item2}',
part.getScheduleMachines(), part_schedule)))
    if all_max < max:
        all_max = max

print("Total time")
print(l.time_format(all_max))
```

Library.py:

```
class Machine:
    '''
    A class for creating machine objects. All machine objects have a name,
capacity and a cooldown time.
    Also, each machine object has a list that keeps track of all the intervals in
which the machine is
    occupied.
        Syntax:
        self.busy_times = [[t1, t2]]
    '''
    all_machines = []
    '''
    A list of all the Machine class objects. Every time an object is created,
it's added
    to this list.
    '''

    def __init__(self, name):
        self.name = name
        self.capacity = 0
        self.cooldown = 0
        self.busy_times = []
        Machine.all_machines.append(self)

    def getName(self):
        return self.name

    def setCapacity(self, capacity):
        self.capacity = capacity

    def getCapacity(self):
```

```
        return self.capacity

    def setCooldown(self, cooldown):
        self.cooldown = cooldown

    def getCooldown(self):
        return self.cooldown

    def addToBusyTimes(self, interval):
        self.busy_times.append(interval)

    def getBusyTimes(self):
        return self.busy_times

    def sortBusyTimes(self):
        self.busy_times.sort(key = lambda x: x[-1])

    def getBusyTimesLen(self):
        return len(self.busy_times)

    def getMachineList():
        return Machine.all_machines


class Part:
    '''
    A class for creating part objects. All part objects have a name, a number of
items, a dictionary
    for all the operations for that part and one for the machining schedule of
each part.
    Syntax:
        self.operations = {Machine: machining time}
        self.schedule = {Machine: [start time, end time]}
    '''
    all_parts = []
    '''
    A list of all the Part class objects. Every time an object is created, it's
added
    to this list.
    '''

    def __init__(self, name):
        self.name = name
        self.items = 0
        self.operations = dict()
```

```
        self.schedule = dict()
        Part.all_parts.append(self)

    def longestSchTime(self):
        max = 0
        for interval in self.schedule.values():
            machining_end = interval[1]
            if max < machining_end:
                max = machining_end
        return max

    def longestOpTime(self):
        max = 0
        for machine in self.operations.keys():
            time = self.operations[machine]
            if max < time:
                max = time
        return max

    def makeCopy(self, part):
        for i in range(1, part.items):
            new_part_name = part.name + " " + str(i)
            new_part = Part(new_part_name)
            new_part.items = 0
            new_part.operations = part.operations.copy()
            new_part.schedule = part.schedule.copy()

    def getName(self):
        return self.name

    def setItems(self):
        return self.items

    def getItems(self):
        return self.items

    def setOperation(self, machine, value):
        self.operations[machine] = value

    def getOperation(self, machine):
        return self.operations[machine]

    def getOperationsMachines(self):
        return list(self.operations.keys())
```

```
    def noOfOperations(self):
        return len(self.operations.keys())

    def delOperation(self, machine):
        del self.operations[machine]

    def setSchedule(self, machine, interval):
        self.schedule[machine] = interval

    def getSchedule(self):
        return self.schedule.values()

    def getScheduleMachines(self):
        return self.schedule.keys()

    def getPartList():
        return Part.all_parts

    def copyPartList():
        return Part.all_parts.copy()

    def orderPartList():
        Part.all_parts.sort(key = lambda part: part.longestOpTime(), reverse =
True)

    def orderStageTwo():
        Part.all_parts.sort(key = lambda x: (0 in x, max(x)), reverse=True)

def get_int(line):
    '''
    Extracting numbers from lines.
    '''
    i = 0
    for char in line:
        if char.isdigit():
            i = i * 10 + int(char)
    return i

def time_format(seconds):
    '''
    Formatting seconds into hours:minutes:seconds format
    '''
    result = "{:02d}:{:02d}:{:02d}".format(seconds // 3600, (seconds % 3600) //
60, (seconds % 60))
    return result
```

```
def auto_input():
    '''
    For testing the code without manually entering the file path
    '''
    input_file = r'D:\facultate\2023_Internship_Challenge_Software\Input_Two.txt'
    f = open(input_file, "r")
    return f

def manual_input():
    '''
    For testing with manual entry of the file path.
    '''
    while True:
        try:
            input_file = input("Please insert the file path: ")
            f = open(input_file, "r")
            break
        except FileNotFoundError:
            print("The path given doesn't return any document. Please check the
location of your file or the format of the path and try again")
        except OSError:
            print("Invalid path.")
        except:
            print("Invalid argument.")
    return f

def spelling_check(control, line):
    '''
    In order to take spelling errors into consideration, first both the machine
name saved
    in machine_list and the one from the line are each converted to lowercase and
saved in a set.
    Then, we take the number of different characters from each set, turn it into
a percentage
    by dividing it with then length of the set machine_name and the lower the
percentage,
    the closer the 2 values are. 0.1 is chosen as treshold to take into
consideration matches.
    When a match is found, it's added to parts_processing.
    This method compares all the characters in the 2 names only once, and deduces
a similarity percentage.
    The loop iterates through all the machine names until the match is found.
    '''
    control = set(control.lower())
```

```
    line = set(line[6:].split(":")[0].lower())
    match = control.symmetric_difference(line)
    match_pct = len(match) / len(control)
    if match_pct < 0.1:
        return True
    else:
        return False

def processingTextFile(f):
    for line in f:
        '''
        The loop goes through each line of the file, looking for key phrases that
mark the sections of the document
        '''
        if "Available machines" in line:
            '''
            If "Available machines" is found, then for the next lines extract the
names of the machines and create Machine class objects.
            Based on the formatting in the file, the name of the machine is
extracted by splitting each row at ". ", taking the part after
            the splitting string, eliminating the automatic new line character at
the end and saving it in the placeholder variable aux.
            Then, through aux, each name of the machine is added as a key in the
machines_charact dict and as an element to the
            machines_list list.
            When the splitting string ". " is no longer found in the line, then
the loop exits.
            '''
            for line in f:
                if ". " in line:
                    aux = line.split(". ")[1].strip("\n")
                    machine = Machine(aux)
                else:
                    if "#" in line:
                        continue
                    else:
                        break
        elif "Machine features" in line:
            '''
            If "Machine features" is found, then from the next line onward the
presence of a feature is checked by the presence of the
            character ":".
            '''
            index = 0
            for machine in Machine.getMachineList():
```

```
                for line in f:
                    if ":" in line:
                        if " one " in line:
                            machine.setCapacity(1)
                        elif " two " in line:
                            machine.setCapacity(2)
                        elif "no limit" in line:
                            machine.setCapacity(-1)
                        elif "second" in line:
                            sec = get_int(line)
                            machine.setCooldown(sec)
                        elif "none" in line:
                            machine.setCooldown(0)
                    else:
                        break
        elif "Part list" in line:
            '''
        If "Part list" is found, then from the next line onward if it finds a ". " then considers that the line contains a part name.
        When ". " is not found and a "#" is not present, the loop breaks
        The splitting string ". " is used to separate the number order and " - " to separate the name from the item.
        A Part class object is created for each new part. The number of items is saved in the item attribute.
        '''
            for line in f:
                if ". " in line:
                    aux = line.split(". ")[1]
                    name = aux.split(" - ")[0]
                    items = aux.split(" - ")[1]
                    items = items.split(" item")[0].strip("\n")

                    part = Part(name)
                    #Part.part_types.append(name)
                    part.items = get_int(items)
                else:
                    if "#" in line:
                        continue
                    else:
                        break
        elif "Part operations" in line:
            '''
            If the phrase "Part operations" is found, then the next lines checked for the piece and its operations.
```

```
            The start of the operations list for each part is identified by
looking for ":".
            '''
            for part in Part.all_parts:
                for line in f:
                    if ":" in line:
                        machine_list = Machine.getMachineList()
                        for machine in machine_list:
                            machine_name = machine.getName()
                            if line != "\n":
                                if spelling_check(machine_name, line):
                                    aux = get_int(line.split(" - ")[1])
                                    part.setOperation(machine_name, aux)
                                    break
                        else:
                            break
```

Stage 4:

```
from library import Machine
from library import Part
import library as l
import getinput as gui
import tkinter as tk

#f = l.auto_input()

f = gui.f
l.processingTextFile(f)

'''
Since continuity is no longer a concern, first we schedule the first machining
operation
based on the longest machining time of each part. After this, we loop the parts
order and
schedule the part with the lowest machining time each time. When a machining
operation is done, it's
deleted from the part's operations list.
'''

for part in Part.getPartList():
    '''
    For each part, we check how many items there need to be and make new objects
of class Part
    '''
    if part.getItems() > 1:
        part.makeCopy(part)

#ordering all the parts by the longest machining time
Part.orderPartList()

last_time = 0

parts_list = Part.copyPartList()

for part in parts_list:
    '''
    Scheduling the first operation for each part. It is taken into consideration
that each part has the same first machining operation.
    '''
    first_machine = Machine.getMachineList()[0]
    first_machine_name = first_machine.getName()
```

```
    free_machines = first_machine.getCapacity()
    machining_time = part.getOperation(first_machine_name)
    interval = [last_time, last_time + machining_time]
    part.setSchedule(first_machine_name, interval)
    first_machine_busy_times = first_machine.getBusyTimes()
    if [last_time, last_time + machining_time] not in first_machine_busy_times:
        cooldown = first_machine.getCooldown()
        period = [last_time, last_time + machining_time + cooldown]
        first_machine.addToBusyTimes(period)
    last_time += machining_time
    part.delOperation(first_machine_name)

#Sorting the part list by the longest machining time of each machine
parts_list = sorted(Part.getPartList(), key = lambda part: part.longestOpTime(),
reverse = True)

while len(parts_list) != 0:
    '''
    Scheduling all the other operations in the order of which they are finishing
their last operation.
    The initial interval for the next machining is starting from the finish time
of the last operation and
    ending by adding the machining time.
    Then the interval is checked against the busy times of the machine. If it
doesn't intersect with an already occupied
    interval then it's added to the parts schedule and the machine's busy times.
It it does, we add 10 to both ends of the
    interval and compare again.
    Completed operations are deleted from the list.
    Finished parts are deleted from the list.
    '''
    min = 9999999999999999
    max = 0
    for part in parts_list:
        for interval in part.getSchedule():
            machining_end = interval[1]
            if max < machining_end:
                max = machining_end
        if min > max:
            min = max
            c_part = part
    c_machine_name = c_part.getOperationsMachines()[0]
    next_machine = next((machine for machine in Machine.getMachineList() if
machine.getName() == c_machine_name), None)
    next_machine_name = next_machine.getName()
```

```
    t_start = min #previous end time
    t_working = c_part.getOperation(next_machine_name)
    t_end = t_start + t_working
    inc = 0
    wrong = 1
    while wrong == 1:
        wrong = 0
        t_start += inc
        t_end += inc
        next_machine.sortBusyTimes()
        if next_machine.getBusyTimesLen() > 0:
            free_machines = next_machine.getCapacity()
            busy_times = next_machine.getBusyTimes()
            for interval in busy_times:
                if t_start in range(interval[0], interval[1]) or t_end in
range(interval[0], interval[1]):
                    if free_machines == 1:
                        wrong = 1
                        inc = 50
                        break
                    else:
                        free_machines -= 1
    busy_times = next_machine.getBusyTimes()
    if [t_start, t_end] not in busy_times:
        cooldown = next_machine.getCooldown()
        period = [t_start, t_end + cooldown]
        next_machine.addToBusyTimes(period)
    c_part.setSchedule(next_machine_name, [t_start, t_end])
    c_part.delOperation(next_machine_name)
    for item in parts_list:
        if item.noOfOperations() == 0:
            del parts_list[parts_list.index(item)]

window = tk.Tk()

scrollbar = tk.Scrollbar(window)
scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

results = tk.Text(window, yscrollcommand=scrollbar.set)
results.pack(side=tk.LEFT, fill=tk.BOTH)

#Printing schedule
all_parts = Part.getPartList()
max = 0 #part final time
all_max = 0 #overall final one
```

```
for part in all_parts:
    results.insert(tk.END, '{:<10}{}'.format("", part.getName()))
    max = 0
    for interval in part.getSchedule():
        machining_end = interval[1]
        if max < machining_end:
            max = machining_end
    part_schedule = part.getSchedule()
    for interval in part_schedule:
        interval[0] = l.time_format(interval[0])
        interval[1] = l.time_format(interval[1])
    results.insert(tk.END, '\n'.join(map(lambda item:
f'\n{item[0]}\n    {item[1]}\n', zip(part.getScheduleMachines(),
part_schedule))))
    if all_max < max:
        all_max = max

results.insert(tk.END, "Total time:")
results.insert(tk.END, l.time_format(all_max))

width = 50

results.config(width=width)

window.mainloop()
```

getinput.py:

```
import tkinter as tk

def open_input_window():
    global f
    try:
        entered_text = entry.get()
        f = open(entered_text, "r")
    except FileNotFoundError:
        label = tk.Label(window, text="The path given doesn't return any
document. Please check the location of your file or the format of the path and
try again")
```

```
        label.pack()
    except OSError:
        label = tk.Label(window, text="Invalid path.")
        label.pack()
    except:
        label = tk.Label(window, text="Invalid argument.")
        label.pack()
    else:
        close_current_window()

def close_current_window():
    window.destroy()

# Create the main window
window = tk.Tk()

# Create an Entry widget
entry = tk.Entry(window)
entry.pack()

# Create a button widget
button = tk.Button(window, text="Submit", command=open_input_window)
button.pack()

# Start the GUI event loop
window.mainloop()
```