Dataset:

The dataset I made was quite big but due to lack of time I used some of the data and clash classes to check my algorithms. The Data I used is also attached with the project but I am also pasting it below.

Subject	Day	TimeSlot	Class	Instructor	Room	Capacity	Priority
WE	Monday	8:30-10:30	A	FatimaBashir	E-107	30	Low
TW	Monday	10:30-11:30	A	FazalAli	E-107	30	Low
AILAB	Monday	11:30-2:30	A	TahiraAfridi	E-216	40	High
SRE	Monday	2:30-4:30	A	AmnaBashir	E-206	20	Low
NA	Tuesday	9:30-11:30	A	FasihaIkram	E-107	30	High
AI	Tuesday	12:30-2:30	A	Tariq	E-213	25	High
DS	Wednesday	9:30-11:30	A	Laila	E-107	30	High
WELAB	Wednesday	11:30-2:30	A	Ambreen	E-216	40	Low
SRE	Thursday	8:30-9:30	A	AmnaBashir	E-107	30	Low
NA	Thursday	10:30-11:30	A	FasihaIkram	E-415	35	High
TW	Thursday	11:30-1:30	A	FazalAli	E-213	35	Low
AI	Thursday	1:30-2:30	A	Tariq	E-205	30	High
MAD	Thursday	2:30-4:30	A	FasihaIkram	E-205	30	Low
DSLAB	Friday	8:30-11:30	A	TahiraAfridi	E-219	40	High
MADLAB	Friday	11:30-2:30	A	TahiraAfridi	E-217	40	Low
WE	Monday	8:30-10:30	B	FatimaBashir	E-107	30	Low
AI	Monday	10:30-11:30	B	Tariq	E-205	30	High
TW	Monday	11:30-1:30	B	FazalAli	E-205	30	Low
SRE	Monday	2:30-4:30	B	AmnaBashir	E-206	30	Low
AILAB	Tuesday	8:30-11:30	B	TahiraAfridi	E-216	40	High
NA	Tuesday	11:30-12:30	B	FasihaIkram	E-415	40	High
DS	Tuesday	12:30-2:30	B	Laila	E-310	30	High
TW	Wednesday	8:30-9:30	B	FazalAli	E-107	30	Low
NA	Wednesday	9:30-11:30	B	FasihaIkram	E-106	30	High
WELAB	Wednesday	11:30-2:30	B	Ambreen	E-216	40	Low
SRE	Thursday	8:30-9:30	B	AmnaBashir	E-107	30	Low
AI	Thursday	9:30-11:30	B	Tariq	E-107	30	High
DSLAB	Thursday	11:30-2:30	B	SabaImtiaz	E-217	40	High
MAD	Thursday	2:30-4:30	B	FasihaIkram	E-205	30	Low
MADLAB	Friday	11:30-2:30	B	TahiraAfridi	E-217	40	Low

Algorithms used and Local Search Algorithm+Log:

At first I tried using **Ant Colony Optimization Algorithm** but it was giving an accuracy of **32%,** it maybe because of some errors done by me. But I dropped this idea on **28/5/2023.** Then afterwards I chose the **PYGAD Library** on 30/**5/2023** but again after giving several tries I couldn't make the work by myself, In short I needed some in my group to think but not like me which I was all alone. Then finally I decided on **2/6/2023**, to start coding(this part took long cause I faced numerous errors and I had to tackle them too) the genetic algorithm by myself but with some help fromresearch papers and various internet resources and side by side as to compare the accuracy of **genetic algorithm with another algorithm I chose stimulated annealing as my second algorithm.**

For **Local search exploration** I used two Functions that I saw in a research paper(Link attached below) that were:

1. **ssn (Simple Searching Neighborhood):** Randomly changes the time slot of a class or lab in a given solution.

   Code:

```
def ssn(solution):
    rand_slot = random.choice(slots)
    rand_lt = random.choice(lts)

    a = random.randint(0, len(solution) - 1)

    new_solution = copy.deepcopy(solution)
    new_solution[a] = course_bits(solution[a]) + professor_bits(solution[a])
+\
        group_bits(solution[a]) + rand_slot + lt_bits(solution[a])
```

```
    return [new_solution]
```

2. **swn (Swapping Neighborhood):** Randomly selects two classes and swaps their time slots.

   Code:

```
def swn(solution):
    a = random.randint(0, len(solution) - 1)
    b = random.randint(0, len(solution) - 1)
    new_solution = copy.deepcopy(solution)
    temp = slot_bits(solution[a])
    new_solution[a] = course_bits(solution[a]) + professor_bits(solution[a])
+\
        group_bits(solution[a]) + slot_bits(solution[b]) +
lt_bits(solution[a])

    new_solution[b] = course_bits(solution[b]) + professor_bits(solution[b])
+\
        group_bits(solution[b]) + temp + lt_bits(solution[b])
    return [new_solution]
```

Algorithms Description +Python Code:

1. Genetic Algorithm:

A genetic algorithm is a heuristic search method used to find solutions for optimization problems. It is based on the Darwinian theory of evolution. This technique involves the ultimate selection of the fittest (best timetable) from a randomly created population (chromosomes) of solutions for the timetabling problem where each individual (chromosome) represents a timetable. The optimality (perfection) of a chromosome is evaluated by a fitness function based on hard and soft constraints . Genetic algorithms begin by creating a random population of timetables followed by their evaluation according to defined criteria to select parents (timetables) for the next generation which is expected to produce better timetables by way of crossovers and mutations. The process is repeated until a satisfactory solution is reached.

Code:

```
def genetic_algorithm():

 generation = 0
 convert_input_to_bin()
 population = init_population(3)

 print("\n------------- Genetic Algorithm --------------\n")
 while True:

     # termination criteria
     if evaluate(max(population, key=evaluate)) == 1 or generation == 500:
         print("Generations:", generation)
         print("Best Chromosome fitness value",
               evaluate(max(population, key=evaluate)))
         print("Best Chromosome: ", max(population, key=evaluate))
         for lec in max(population, key=evaluate):
             print_chromosome(lec)
         break

     # Otherwise continue
     else:
         for _c in range(len(population)):
             crossover(population)
             selection(population, 5)

             # selection(population[_c], len(cpg))
             mutate(population[_c])

     generation = generation + 1
```

Simulated Annealing:

Simulated annealing is a method for solving unconstrained and bound-constrained optimization problems. The method models the physical process of heating a material and then slowly lowering the temperature to decrease defects, thus minimizing the system energy.

Code:

```
def simulated_annealing():

 alpha = 0.9
```

```
    T = 1.0
    T_min = 0.00001

    convert_input_to_bin()
    # as simulated annealing is a single-state method
    population = init_population(1)
    old_cost = cost(population[0])
    for __n in range(500):
        new_solution = swn(population[0])
        new_solution = ssn(population[0])
        new_cost = cost(new_solution[0])
        ap = acceptance_probability(old_cost, new_cost, T)
        if ap > random.random():
            population = new_solution
            old_cost = new_cost
        T = T * alpha

    print("\n------------- Simulated Annealing --------------\n")
    for lec in population[0]:
        print_chromosome(lec)
    print("Score: ", evaluate(population[0]))
```

Constraints:

- checks that a faculty member teaches only one course at a time.
- check that a group member takes only one class at a time.
- checks that a course is assigned to an available classroom.
- checks that the classroom capacity is large enough for the classes that are assigned to that classroom.
- check that room is appropriate for particular class/lab
- check that lab is allocated appropriate time slot

Result + Analysis:

The hard constraints were tested to ensure that all the solutions obtained were valid. The optimized solution for the timetable consisted of the following factors and their values. The population size was 100 with a mutation rate of 0.01%, a crossover rate of 0.9%. With these values the resulting timetable had zero number of clashes with the fitness value of 0.9777 – 1.000. Furthermore; From the Accuracy Point of View The Genetic Algorithm had 93.33333% Accuracy and Stimulated

Annealing had 86.6666% Accuracy. So, this clearly shows that Genetic Algorithm was best in creating cashless Timetable.(Wishing I had coded Genetic Algorithm First I could've saved so much time)

Research Papers Used Along with the Project:

- Microsoft Word - JPCSJ19501065 (iop.org)
- *Genetic Algorithm For University Course Timetabling Problem (olemiss.edu)
- (PDF) Solving Timetable Scheduling Problem by Using Genetic Algorithms | Bagas Kara - Academia.edu
- (2) (PDF) Solving Timetable Problem by Genetic Algorithm and Heuristic Search Case Study: Universitas Pelita Harapan Timetable (researchgate.net)
- (2) (PDF) Class Timetable Scheduling with Genetic Algorithm (researchgate.net)

OUTPUT BELOW⬇

Output:

```
------------- Genetic Algorithm --------------

Generations: 500
Best Chromosome fitness value 0.9333333333333333
Best Chromosome:  ['00000000000010', '00100000100110', '0100
0100001110', '01000100100001', '10001100010011', '1011000000
0110', '0b101000001001', '01101100110010']
CourseClass: AIA  |  Professor: Tariq  |  Group: BSCSA, Size
: 30  |  Slot: 08:30-10:00 Day: Mon  |  Room: E213 Size: 60
CourseClass: AIB  |  Professor: Tariq  |  Group: BSCSB, Size
: 35  |  Slot: 10:15-11:45 Day: Mon  |  Room: E213 Size: 60
CourseClass: NA  |  Professor: Fasiha  |  Group: BSCSA, Size
: 30  |  Slot: 08:30-10:00 Day: Tue  |  Room: E213 Size: 60
CourseClass: NA  |  Professor: Fasiha  |  Group: BSCSB, Size
: 35  |  Slot: 08:30-10:00 Day: Mon  |  Room: E-412 Size: 40
CourseClass: AILAB  |  Professor: Tahira  |  Group: BSCSA, S
ize: 30  |  Slot: 08:30-11:30 Day: Mon  |  Room: E-219(LAB)
Size: 40
CourseClass: DS  |  Professor: Laila  |  Group: BSCSA, Size:
 30  |  Slot: 10:15-11:45 Day: Mon  |  Room: E213 Size: 60
CourseClass: AIB  |  Professor: Fazal  |  Group: BSCSA, Size
: 30  |  Slot: 12:00-13:30 Day: Mon  |  Room: E-412 Size: 40
CourseClass: MADLAB  |  Professor: Tahira  |  Group: BSCSB,
Size: 35  |  Slot: 08:30-11:30 Day: Mon  |  Room: E213 Size:
 60
```

```
------------- Simulated Annealing --------------

CourseClass: AIA  |  Professor: Tariq  |  Group: BSCSA, Size
: 30  |  Slot: 08:30-10:00 Day: Tue  |  Room: E-219(LAB) Siz
e: 40
CourseClass: AIB  |  Professor: Tariq  |  Group: BSCSB, Size
: 35  |  Slot: 12:00-13:30 Day: Mon  |  Room: E-219(LAB) Siz
e: 40
CourseClass: NA  |  Professor: Fasiha  |  Group: BSCSA, Size
: 30  |  Slot: 08:30-10:00 Day: Mon  |  Room: E213 Size: 60
CourseClass: NA  |  Professor: Fasiha  |  Group: BSCSB, Size
: 35  |  Slot: 08:30-10:00 Day: Tue  |  Room: E213 Size: 60
CourseClass: AILAB  |  Professor: Tahira  |  Group: BSCSA, S
ize: 30  |  Slot: 08:30-11:30 Day: Mon  |  Room: E-219(LAB)
Size: 40
CourseClass: DS  |  Professor: Laila  |  Group: BSCSA, Size:
 30  |  Slot: 12:00-13:30 Day: Mon  |  Room: E-102 Size: 20
CourseClass: AIB  |  Professor: Fazal  |  Group: BSCSA, Size
: 30  |  Slot: 10:15-11:45 Day: Mon  |  Room: E-219(LAB) Siz
e: 40
CourseClass: MADLAB  |  Professor: Tahira  |  Group: BSCSB,
Size: 35  |  Slot: 10:15-11:45 Day: Mon  |  Room: E213 Size:
 60
Score:  0.8666666666666667
PS C:\Users\hamma\OneDrive\Desktop\AI-CCP>
```

Whole Code:

Trial-2.py:

```
import random

import copy

from Classes import *

from math import ceil, log2

import math

Group.groups = [Group("BSCSA", 30), Group("BSCSB", 35), Group(

    "BSCSC", 30), Group("BSCSD", 50), Group("BSCSE", 30)]
```

```
Professor.professors = [Professor("Tariq"), Professor("Fasiha"), Professor("Fazal"),
                        Professor("Tahira"), Professor("Laila")]

CourseClass.classes = [CourseClass("AIA"), CourseClass("AIB"), CourseClass("NA"),
                       CourseClass("MADLAB",is_lab=True), CourseClass(
                           "AILAB", is_lab=True),
                       CourseClass("DS"), CourseClass("TW"), CourseClass("SRE")]

Room.rooms = [Room("E-102", 20), Room("E-412", 40), Room(
    "E213", 60), Room("E-219(LAB)", 40, is_lab=True)]

Slot.slots = [Slot("08:30", "10:00", "Mon"), Slot("10:15", "11:45", "Mon"),
              Slot("12:00", "13:30", "Mon"), Slot("08:30", "10:00", "Tue"), Slot("08:30", "11:30", "Mon",
True)]


max_score = None


cpg = []
lts = []
slots = []
bits_needed_backup_store = {}  # to improve performance
```

```
def bits_needed(x):
    global bits_needed_backup_store
    r = bits_needed_backup_store.get(id(x))
    if r is None:
        r = int(ceil(log2(len(x))))
        bits_needed_backup_store[id(x)] = r
    return max(r, 1)


def join_cpg_pair(_cpg):
    res = []
    for i in range(0, len(_cpg), 3):
        res.append(_cpg[i] + _cpg[i + 1] + _cpg[i + 2])
    return res


def convert_input_to_bin():
    global cpg, lts, slots, max_score

    cpg = [CourseClass.find("AIA"), Professor.find("Tariq"), Group.find("BSCSA"),
           CourseClass.find("AIB"), Professor.find(
               "Tariq"), Group.find("BSCSB"),
           CourseClass.find("NA"), Professor.find(
```

```
        "Fasiha"), Group.find("BSCSA"),
    CourseClass.find("NA"), Professor.find(
        "Fasiha"), Group.find("BSCSB"),
    CourseClass.find("AILAB"), Professor.find(
        "Tahira"), Group.find("BSCSA"),
    CourseClass.find("DS"), Professor.find(
        "Laila"), Group.find("BSCSA"),
    CourseClass.find("cs101"), Professor.find(
        "Fazal"), Group.find("BSCSA"),
    CourseClass.find("MADLAB"), Professor.find(
        "Tahira"), Group.find("BSCSB")
    ]
for _c in range(len(cpg)):
    if _c % 3:  # CourseClass
        cpg[_c] = (bin(cpg[_c])[2:]).rjust(
            bits_needed(CourseClass.classes), '0')
    elif _c % 3 == 1:  # Professor
        cpg[_c] = (bin(cpg[_c])[2:]).rjust(
            bits_needed(Professor.professors), '0')
    else:  # Group
        cpg[_c] = (bin(cpg[_c])[2:]).rjust(bits_needed(Group.groups), '0')

cpg = join_cpg_pair(cpg)
for r in range(len(Room.rooms)):
```

```
        lts.append((bin(r)[2:]).rjust(bits_needed(Room.rooms), '0'))

    for t in range(len(Slot.slots)):
        slots.append((bin(t)[2:]).rjust(bits_needed(Slot.slots), '0'))

    # print(cpg)
    max_score = (len(cpg) - 1) * 3 + len(cpg) * 3


def course_bits(chromosome):
    i = 0

    return chromosome[i:i + bits_needed(CourseClass.classes)]


def professor_bits(chromosome):
    i = bits_needed(CourseClass.classes)

    return chromosome[i: i + bits_needed(Professor.professors)]


def group_bits(chromosome):
    i = bits_needed(CourseClass.classes) + bits_needed(Professor.professors)
```

```
    return chromosome[i:i + bits_needed(Group.groups)]


def slot_bits(chromosome):
    i = bits_needed(CourseClass.classes) + bits_needed(Professor.professors) + \
        bits_needed(Group.groups)

    return chromosome[i:i + bits_needed(Slot.slots)]


def lt_bits(chromosome):
    i = bits_needed(CourseClass.classes) + bits_needed(Professor.professors) + \
        bits_needed(Group.groups) + bits_needed(Slot.slots)

    return chromosome[i: i + bits_needed(Room.rooms)]


def slot_clash(a, b):
    if slot_bits(a) == slot_bits(b):
        return 1
    return 0


# checks that a faculty member teaches only one course at a time.
```

```
def faculty_member_one_class(chromosome):
    scores = 0
    for i in range(len(chromosome) - 1):  # select one cpg pair
        clash = False
        for j in range(i + 1, len(chromosome)):  # check it with all other cpg pairs
            if slot_clash(chromosome[i], chromosome[j])\
                    and professor_bits(chromosome[i]) == professor_bits(chromosome[j]):
                clash = True
        if not clash:
            scores = scores + 1
    return scores


# check that a group member takes only one class at a time.
def group_member_one_class(chromosomes):
    scores = 0

    for i in range(len(chromosomes) - 1):
        clash = False
        for j in range(i + 1, len(chromosomes)):
            if slot_clash(chromosomes[i], chromosomes[j]) and\
                    group_bits(chromosomes[i]) == group_bits(chromosomes[j]):
                clash = True
                break
```

```
        if not clash:
            scores = scores + 1
    return scores


# checks that a course is assigned to an available classroom.
def use_spare_classroom(chromosome):
    scores = 0
    for i in range(len(chromosome) - 1):  # select one cpg pair
        clash = False
        for j in range(i + 1, len(chromosome)):  # check it with all other cpg pairs
            if slot_clash(chromosome[i], chromosome[j]) and lt_bits(chromosome[i]) ==
lt_bits(chromosome[j]):
                clash = True
        if not clash:
            scores = scores + 1
    return scores


# checks that the classroom capacity is large enough for the classes that
# are assigned to that classroom.
def classroom_size(chromosomes):
    scores = 0
    for _c in chromosomes:
```

```
        if Group.groups[int(group_bits(_c), 2)].size <= Room.rooms[int(lt_bits(_c), 2)].size:
            scores = scores + 1
    return scores


# check that room is appropriate for particular class/lab
def appropriate_room(chromosomes):
    scores = 0
    for _c in chromosomes:
        if CourseClass.classes[int(course_bits(_c), 2)].is_lab == Room.rooms[int(lt_bits(_c), 2)].is_lab:
            scores = scores + 1
    return scores


# check that lab is allocated appropriate time slot
def appropriate_timeslot(chromosomes):
    scores = 0
    for _c in chromosomes:
        if CourseClass.classes[int(course_bits(_c), 2)].is_lab == Slot.slots[int(slot_bits(_c),
2)].is_lab_slot:
            scores = scores + 1
    return scores
```

```
def evaluate(chromosomes):
    global max_score
    score = 0
    score = score + use_spare_classroom(chromosomes)
    score = score + faculty_member_one_class(chromosomes)
    score = score + classroom_size(chromosomes)
    score = score + group_member_one_class(chromosomes)
    score = score + appropriate_room(chromosomes)
    score = score + appropriate_timeslot(chromosomes)
    return score / max_score


def cost(solution):
    return 1 / float(evaluate(solution))


def init_population(n):
    global cpg, lts, slots
    chromosomes = []
    for _n in range(n):
        chromosome = []
        for _c in cpg:
            chromosome.append(_c + random.choice(slots) + random.choice(lts))
        chromosomes.append(chromosome)
```

```
    return chromosomes


def mutate(chromosome):
    rand_slot = random.choice(slots)
    rand_lt = random.choice(lts)

    a = random.randint(0, len(chromosome) - 1)

    chromosome[a] = course_bits(chromosome[a]) + professor_bits(chromosome[a]) +\
        group_bits(chromosome[a]) + rand_slot + rand_lt

    # print("After mutation: ", end="")
    # printChromosome(chromosome)


def crossover(population):
    a = random.randint(0, len(population) - 1)
    b = random.randint(0, len(population) - 1)
    # assume all chromosome are of same len
    cut = random.randint(0, len(population[0]))
    population.append(population[a][:cut] + population[b][cut:])


def selection(population, n):
```

```
    population.sort(key=evaluate, reverse=True)
    while len(population) > n:
        population.pop()


def print_chromosome(chromosome):
    print(CourseClass.classes[int(course_bits(chromosome), 2)], " | ",
          Professor.professors[int(professor_bits(chromosome), 2)], " | ",
          Group.groups[int(group_bits(chromosome), 2)], " | ",
          Slot.slots[int(slot_bits(chromosome), 2)], " | ",
          Room.rooms[int(lt_bits(chromosome), 2)])

# Simple Searching Neighborhood
# It randomly changes timeslot of a class/lab
def ssn(solution):
    rand_slot = random.choice(slots)
    rand_lt = random.choice(lts)

    a = random.randint(0, len(solution) - 1)

    new_solution = copy.deepcopy(solution)
    new_solution[a] = course_bits(solution[a]) + professor_bits(solution[a]) +\
        group_bits(solution[a]) + rand_slot + lt_bits(solution[a])
    return [new_solution]
```

```
# Swapping Neighborhoods

# It randomy selects two classes and swap their time slots

def swn(solution):

    a = random.randint(0, len(solution) - 1)

    b = random.randint(0, len(solution) - 1)

    new_solution = copy.deepcopy(solution)

    temp = slot_bits(solution[a])

    new_solution[a] = course_bits(solution[a]) + professor_bits(solution[a]) +\

        group_bits(solution[a]) + slot_bits(solution[b]) + lt_bits(solution[a])


    new_solution[b] = course_bits(solution[b]) + professor_bits(solution[b]) +\

        group_bits(solution[b]) + temp + lt_bits(solution[b])

    return [new_solution]




def acceptance_probability(old_cost, new_cost, temperature):

    if new_cost < old_cost:

        return 1.0

    else:

        return math.exp((old_cost - new_cost) / temperature)




def simulated_annealing():
```

```
alpha = 0.9

T = 1.0

T_min = 0.00001


convert_input_to_bin()

# as simulated annealing is a single-state method

population = init_population(1)

old_cost = cost(population[0])

for __n in range(500):

    new_solution = swn(population[0])

    new_solution = ssn(population[0])

    new_cost = cost(new_solution[0])

    ap = acceptance_probability(old_cost, new_cost, T)

    if ap > random.random():

        population = new_solution

        old_cost = new_cost

    T = T * alpha


print("\n------------- Simulated Annealing --------------\n")

for lec in population[0]:

    print_chromosome(lec)

print("Score: ", evaluate(population[0]))
```

```
def genetic_algorithm():
    generation = 0
    convert_input_to_bin()
    population = init_population(3)

    print("\n------------- Genetic Algorithm --------------\n")
    while True:

        # termination criteria
        if evaluate(max(population, key=evaluate)) == 1 or generation == 500:
            print("Generations:", generation)
            print("Best Chromosome fitness value",
                  evaluate(max(population, key=evaluate)))
            print("Best Chromosome: ", max(population, key=evaluate))
            for lec in max(population, key=evaluate):
                print_chromosome(lec)
            break

        # Otherwise continue
        else:
            for _c in range(len(population)):
                crossover(population)
                selection(population, 5)
```

```
            # selection(population[_c], len(cpg))
            mutate(population[_c])

        generation = generation + 1


def main():
    random.seed()
    genetic_algorithm()
    simulated_annealing()


main()
```

<u>Classes.py:</u>

```
class Group:
    groups = None

    def __init__(self, name, size):
        self.name = name
        self.size = size

    @staticmethod
    def find(name):
```

```
        for i in range(len(Group.groups)):
            if Group.groups[i].name == name:
                return i
        return -1

    def __repr__(self):
        return "Group: " + self.name + ", Size: " + str(self.size)


class Professor:
    professors = None

    def __init__(self, name):
        self.name = name

    @staticmethod
    def find(name):
        for i in range(len(Professor.professors)):
            if Professor.professors[i].name == name:
                return i
        return -1

    def __repr__(self):
        return "Professor: " + self.name
```

```
class CourseClass:
    classes = None

    def __init__(self, code, is_lab=False):
        self.code = code
        self.is_lab = is_lab

    @staticmethod
    def find(code):
        for i in range(len(CourseClass.classes)):
            if CourseClass.classes[i].code == code:
                return i
        return -1

    def __repr__(self):
        return "CourseClass: " + self.code


class Room:
    rooms = None

    def __init__(self, name, size, is_lab=False):
```

```
        self.name = name
        self.size = size
        self.is_lab = is_lab

    @staticmethod
    def find(name):
        for i in range(len(Room.rooms)):
            if Room.rooms[i].name == name:
                return i
        return -1

    def __repr__(self):
        return "Room: " + self.name + " Size: " + str(self.size)


class Slot:
    slots = None

    def __init__(self, start, end, day, is_lab_slot=False):
        self.start = start
        self.end = end
        self.day = day
        self.is_lab_slot = is_lab_slot
```

```
def __repr__(self):
    return "Slot: " + self.start + "-" + self.end + " Day: " + self.day
```

Thank You! 😊