Question 12.1

Describe a situation or problem from your job, everyday life, current events, etc., for which a design of experiments approach would be appropriate.

If I am running a hospital and am investigating the causes of emergency room wait times, there is a possibility that there are a lot of different causes (or combinations of causes!) at my specific hospital. Therefore, I could use a design of experiments (DOE) approach to gather data that I can analyze to determine which factors are the most relevant in terms of causing longer wait times. I could then try and make the overall process more efficient by addressing those factors.

Question 12.2

To determine the value of 10 different yes/no features to the market value of a house (large yard, solar roof, etc.), a real estate agent plans to survey 50 potential buyers, showing a fictitious house with different combinations of features. To reduce the survey size, the agent wants to show just 16 fictitious houses. Use R's `FrF2` function (in the `FrF2` package) to find a fractional factorial design for this experiment: what set of features should each of the 16 fictitious houses have? Note: the output of `FrF2` is "1" (include) or "-1" (don't include) for each feature.

The code for this approach can be found in appendix 12.2. This following is one solution that I obtained:

	Feature									
House #	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**
1	-1	-1	-1	1	1	1	1	-1	1	-1
2	1	1	-1	-1	1	-1	-1	-1	1	1
3	-1	1	1	-1	-1	-1	1	1	-1	1
4	-1	-1	-1	-1	1	1	1	1	-1	1
5	1	-1	-1	-1	-1	-1	1	-1	-1	-1
6	1	-1	1	1	-1	1	-1	1	-1	-1
7	1	1	-1	1	1	-1	-1	1	-1	-1
8	-1	1	-1	-1	-1	1	-1	1	1	-1
9	-1	-1	1	1	1	-1	-1	-1	-1	1
10	-1	-1	1	-1	1	-1	-1	1	1	-1
11	-1	1	-1	1	-1	1	-1	-1	-1	1
12	1	-1	-1	1	-1	-1	1	1	1	1
13	1	-1	1	-1	-1	1	-1	-1	1	1
14	-1	1	1	1	-1	-1	1	-1	1	-1
15	1	1	1	1	1	1	1	1	1	1
16	1	1	1	-1	1	1	1	-1	-1	-1

If I were to run the code again with a different seed, I would get a different fractional factorial design:

```
   A  B  C  D  E  F  G  H  J  K
1 -1 -1  1 -1  1 -1 -1  1  1 -1
```

```
2  -1  1  1  1 -1 -1  1 -1  1 -1
3   1 -1  1 -1 -1  1 -1 -1  1  1
4   1 -1  1  1 -1  1 -1  1 -1 -1
5   1  1  1 -1  1  1  1 -1 -1 -1
6  -1 -1 -1 -1  1  1  1  1 -1  1
7  -1  1 -1  1 -1  1 -1 -1 -1  1
8  -1 -1 -1  1  1  1  1 -1  1 -1
9   1 -1 -1 -1 -1 -1  1 -1 -1 -1
10 -1  1 -1 -1 -1  1 -1  1  1 -1
11  1 -1 -1  1 -1 -1  1  1  1  1
12 -1  1  1 -1 -1 -1  1  1 -1  1
13  1  1 -1  1  1 -1 -1  1 -1 -1
14  1  1 -1 -1  1 -1 -1 -1  1  1
15 -1 -1  1  1  1 -1 -1 -1 -1  1
16  1  1  1  1  1  1  1  1  1  1
```

I would assume that these differences are because the process relies on random number generation.

Question 13.1

For each of the following distributions, give an example of data that you would expect to follow this distribution (besides the examples already discussed in class).

- Binomial
    - You could use a binomial distribution to model the yearly probability of a river overflowing a certain number of times due to increased rainfall. Then, using this distribution, you can tell how many times you might need to be ready for overflow during the year
- Geometric
    - If you flip a bottle, the number of trials before the bottle lands upright could be described as a geometric distribution
- Poisson
    - You could use a Poisson distribution to describe the expected number of customers that might arrive at a restaurant in a day
- Exponential
    - You could use an exponential distribution to model the time, perhaps in hours, between earthquakes in a certain location
- Weibull
    - You could use a Weibull distribution to model the time it takes for a very expensive computer screen to fail

Question 13.2

In this problem you, can simulate a simplified airport security system at a busy airport. Passengers arrive according to a Poisson distribution with $\lambda_1 = 5$ per minute (i.e., mean interarrival rate $\mu_1 = 0.2$ minutes) to the ID/boarding-pass check queue, where there are several servers who each have exponential service time with mean rate $\mu_2 = 0.75$ minutes. [Hint: model them as one block that has more than one resource.] After that, the passengers are assigned to the shortest of the several personal-check queues, where they go through the personal scanner (time is uniformly distributed between 0.5 minutes and 1 minute).

Use the Arena software (PC users) or Python with SimPy (PC or Mac users) to build a simulation of the system, and then vary the number of ID/boarding-pass checkers and personal-check queues to determine how many are needed to keep average wait times below 15 minutes. [If you're using SimPy, or if you have access to a non-student version of Arena, you can use λ_1 = 50 to simulate a busier airport.]

The code for this approach can be found in appendix 13.2. For all investigations, I used 100 replications and set the simulation runtime to 840 minutes (14 hours). Once I had built my model, I started investigating the ideal number of scanners and ID/boarding-pass checkers. I first did this investigation for the case where λ_1 = 5 passengers per minute and began by setting the number of scanners and ID/boarding-pass checkers to 2. I then continued to vary the number of scanners and ID/boarding-pass checkers until I felt that I had found an optimal number. The results are displayed in the following table:

Scanners	ID/Boarding-Pass Checkers	Average System Time (minutes)	Average Wait Time (minutes)
2	2	203.51	202.01
2	3	195.98	194.48
3	2	197.32	195.82
3	3	92.86	91.35
3	4	85.64	84.14
4	3	86.77	85.27
4	4	5.51	4.01
5	4	4.44	2.94
4	5	3.41	1.90
5	5	2.29	0.79
6	5	2.23	0.73
5	6	2.09	0.59
6	6	2.02	0.52

I didn't put average check times and average scan times in the table because they ended up being the same for all runs at a value of 0.75 minutes. In this situation, we see that with a passenger arrival rate of 5 passengers/minute, the number of scanners and ID/boarding-pass checkers plays a huge role in overall wait times. If there are two scanners and ID/boarding-pass checkers, the wait times are very long at 202.01 minutes per passenger. However, if we expand to three of each, wait times decrease quite a bit to 91.35 minutes per passenger. Yet, that is still a pretty long time to be waiting. So, if we increase to four scanners and ID/boarding-pass checkers, we can reduce wait times below 15 minutes to 4.01 minutes per passenger. This is a much better wait time. If we go ahead and expand to five scanners and ID/boarding-pass checkers, we can actually get to an average of 0.79 minutes of wait time per passenger. This is great and should be implemented assuming the airport has the capacity to do so.

I also decided to investigate the case where $\lambda_1 = 50$ passengers per minute. Because using my original parameters of 100 replications and simulation runtime of 840 minutes (14 hours) took far too long, I decided to use only 30 replications. I also set the runtime to 240 minutes (4 hours). The following table shows the best few attempts in terms of the ideal number of scanners and ID/boarding-pass checkers:

Scanners	ID/Boarding-Pass Checkers	Average Wait Times (minutes)
5	5	104.06
20	20	56.34
30	30	25.37
33	33	15.71
34	34	12.57
35	35	9.81
37	37	3.74
38	38	2.04
39	39	1.13
40	40	0.88

Therefore, looking at the results here, it seems that if we want to decrease wait times below 15 minutes, we should aim to have at least 34 scanners and ID/boarding-pass checkers. However, if we continue to add more until we get to 40 of each, we can even reduce wait times to an average of 0.88 minutes per passenger, which is great.

Appendix

Question 12.2

```
#Load Libraries
library(FrF2)

## Loading required package: DoE.base

## Loading required package: grid

## Loading required package: conf.design

## Registered S3 method overwritten by 'DoE.base':
##   method           from
##   factorize.factor conf.design

##
## Attaching package: 'DoE.base'

## The following objects are masked from 'package:stats':
##
##     aov, lm

## The following object is masked from 'package:graphics':
##
##     plot.design

## The following object is masked from 'package:base':
##
##     lengths

set.seed(1)

frac_design <- FrF2(nruns = 16, nfactors = 10)
summary(frac_design)

## Call:
## FrF2(nruns = 16, nfactors = 10)
##
## Experimental design of type  FrF2
## 16  runs
##
## Factor settings (scale ends):
##    A  B  C  D  E  F  G  H  J  K
## 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
## 2  1  1  1  1  1  1  1  1  1  1
##
## Design generating information:
## $legend
##  [1] A=A B=B C=C D=D E=E F=F G=G H=H J=J K=K
##
```

```
## $generators
## [1] E=AB   F=AC   G=BC   H=AD   J=BCD  K=ABCD
##
##
## Alias structure:
## $main
##  [1] A=BE=CF=DH=JK B=AE=CG       C=AF=BG       D=AH=GJ       E=AB=FG
##  [6] F=AC=EG       G=BC=DJ=EF=HK H=AD=GK       J=AK=DG       K=AJ=GH
##
## $fi2
## [1] AG=BF=CE=DK=HJ BD=CJ=EH=FK    BH=CK=DE=FJ    BJ=CD=EK=FH    BK=CH=DF=E
J
##
##
## The design itself:
##     A  B  C  D  E  F  G  H  J  K
## 1  -1 -1 -1  1  1  1  1 -1  1 -1
## 2   1  1 -1 -1  1 -1 -1 -1  1  1
## 3  -1  1  1 -1 -1 -1  1  1 -1  1
## 4  -1 -1 -1 -1  1  1  1  1 -1  1
## 5   1 -1 -1 -1 -1 -1  1 -1 -1 -1
## 6   1 -1  1  1 -1  1 -1  1 -1 -1
## 7   1  1 -1  1  1 -1 -1  1 -1 -1
## 8  -1  1 -1 -1 -1  1 -1  1  1 -1
## 9  -1 -1  1  1  1 -1 -1 -1 -1  1
## 10 -1 -1  1 -1  1 -1 -1  1  1 -1
## 11 -1  1 -1  1 -1  1 -1 -1 -1  1
## 12  1 -1 -1  1 -1 -1  1  1  1  1
## 13  1 -1  1 -1 -1  1 -1 -1  1  1
## 14 -1  1  1  1 -1 -1  1 -1  1 -1
## 15  1  1  1  1  1  1  1  1  1  1
## 16  1  1  1 -1  1  1  1 -1 -1 -1
## class=design, type= FrF2
```

```
frac_design
```

```
##     A  B  C  D  E  F  G  H  J  K
## 1  -1 -1 -1  1  1  1  1 -1  1 -1
## 2   1  1 -1 -1  1 -1 -1 -1  1  1
## 3  -1  1  1 -1 -1 -1  1  1 -1  1
## 4  -1 -1 -1 -1  1  1  1  1 -1  1
## 5   1 -1 -1 -1 -1 -1  1 -1 -1 -1
## 6   1 -1  1  1 -1  1 -1  1 -1 -1
## 7   1  1 -1  1  1 -1 -1  1 -1 -1
## 8  -1  1 -1 -1 -1  1 -1  1  1 -1
## 9  -1 -1  1  1  1 -1 -1 -1 -1  1
## 10 -1 -1  1 -1  1 -1 -1  1  1 -1
## 11 -1  1 -1  1 -1  1 -1 -1 -1  1
## 12  1 -1 -1  1 -1 -1  1  1  1  1
## 13  1 -1  1 -1 -1  1 -1 -1  1  1
## 14 -1  1  1  1 -1 -1  1 -1  1 -1
## 15  1  1  1  1  1  1  1  1  1  1
## 16  1  1  1 -1  1  1  1 -1 -1 -1
## class=design, type= FrF2
```

```
set.seed(2)
frac_design2 <- FrF2(nruns = 16, nfactors = 10)
summary(frac_design2)
```

```
## Call:
## FrF2(nruns = 16, nfactors = 10)
##
## Experimental design of type  FrF2
## 16  runs
##
## Factor settings (scale ends):
##    A  B  C  D  E  F  G  H  J  K
## 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
## 2  1  1  1  1  1  1  1  1  1  1
##
## Design generating information:
## $legend
##  [1] A=A B=B C=C D=D E=E F=F G=G H=H J=J K=K
##
## $generators
## [1] E=AB   F=AC   G=BC   H=AD   J=BCD  K=ABCD
##
##
## Alias structure:
## $main
##  [1] A=BE=CF=DH=JK B=AE=CG       C=AF=BG       D=AH=GJ       E=AB=FG
##  [6] F=AC=EG       G=BC=DJ=EF=HK H=AD=GK       J=AK=DG       K=AJ=GH
##
## $fi2
## [1] AG=BF=CE=DK=HJ BD=CJ=EH=FK    BH=CK=DE=FJ    BJ=CD=EK=FH    BK=CH=DF=E
J
##
##
## The design itself:
##     A  B  C  D  E  F  G  H  J  K
## 1  -1 -1  1 -1  1 -1 -1  1  1 -1
## 2  -1  1  1  1 -1 -1  1 -1  1 -1
## 3   1 -1  1 -1 -1  1 -1 -1  1  1
## 4   1 -1  1  1 -1  1 -1  1 -1 -1
## 5   1  1  1 -1  1  1  1 -1 -1 -1
## 6  -1 -1 -1 -1  1  1  1  1 -1  1
## 7  -1  1 -1  1 -1  1 -1 -1 -1  1
## 8  -1 -1 -1  1  1  1  1 -1  1 -1
## 9   1 -1 -1 -1 -1 -1  1 -1 -1 -1
## 10 -1  1 -1 -1 -1  1 -1  1  1 -1
## 11  1 -1 -1  1 -1 -1  1  1  1  1
## 12 -1  1  1 -1 -1 -1  1  1 -1  1
## 13  1  1 -1  1  1 -1 -1  1 -1 -1
## 14  1  1 -1 -1  1 -1 -1 -1  1  1
## 15 -1 -1  1  1  1 -1 -1 -1 -1  1
## 16  1  1  1  1  1  1  1  1  1  1
## class=design, type= FrF2
```

```
frac_design2
```

```
##     A  B  C  D  E  F  G  H  J  K
## 1  -1 -1  1 -1  1 -1 -1  1  1 -1
## 2  -1  1  1  1 -1 -1  1 -1  1 -1
## 3   1 -1  1 -1 -1  1 -1 -1  1  1
## 4   1 -1  1  1 -1  1 -1  1 -1 -1
## 5   1  1  1 -1  1  1  1 -1 -1 -1
## 6  -1 -1 -1 -1  1  1  1  1 -1  1
## 7  -1  1 -1  1 -1  1 -1 -1 -1  1
## 8  -1 -1 -1  1  1  1  1 -1  1 -1
## 9   1 -1 -1 -1 -1 -1  1 -1 -1 -1
## 10 -1  1 -1 -1 -1  1 -1  1  1 -1
## 11  1 -1 -1  1 -1 -1  1  1  1  1
## 12 -1  1  1 -1 -1 -1  1  1 -1  1
## 13  1  1 -1  1  1 -1 -1  1 -1 -1
## 14  1  1 -1 -1  1 -1 -1 -1  1  1
## 15 -1 -1  1  1  1 -1 -1 -1 -1  1
## 16  1  1  1  1  1  1  1  1  1  1
## class=design, type= FrF2
```

Question 13.2

```
import simpy
import random

# Constants and Global Variables
NUM_BP_CHECK = 5 # Number of our ID/Boarding-Pass Checkers
NUM_SCAN = 5 # Number of our scanners
ARRIVAL_RATE = 5 # Arrival rate based on question prompt
CHECK_RATE = 0.75 # BP check rate
MIN_SCAN = 0.5 # Min time for scanner, unif dist.
MAX_SCAN = 1.0 # Max time for scanner, unif dist.
RUN_TIME = 840 # Run time in minutes/simulation
REP = 100 # Number of replications

# List to store results from each replication
AVG_CHECK_TIME = []
AVG_SCAN_TIME = []
AVG_WAIT_TIME = []
AVG_SYSTEM_TIME = []
```

```
# Generate the model
class System:
    def __init__(self, env, num_bp_check, num_scan):
        self.env = env
        self.checker = simpy.Resource(env, num_bp_check)
        self.scanner = []
        for i in range(num_scan):
            self.scanner.append(simpy.Resource(env,1))

    def check(self, passenger):
        yield self.env.timeout(random.expovariate(1.0/CHECK_RATE))

    def scan(self, passenger):
        yield self.env.timeout(random.uniform(MIN_SCAN, MAX_SCAN))

def passenger(env, name, sys):
    time_arrive = env.now

    with sys.checker.request() as request:
        yield request
        t_in = env.now
        yield env.process(sys.check(name))
        t_out = env.now
        check_time.append(t_out - t_in)

    minq = 0
    for i in range(1, NUM_SCAN):
        if (len(sys.scanner[i].queue) < len(sys.scanner[minq].queue)):
            minq = i

    with sys.scanner[minq].request() as request:
        yield request
        t_in = env.now
        yield env.process(sys.scan(name))
        t_out = env.now
        scan_time.append(t_out - t_in)

    time_leave = env.now
    sys_time.append(time_leave - time_arrive)
    tot_throughput.append(1)

def setup(env, arrival_rate, num_bp_check, num_scan):
    i = 0
    sys = System(env, num_bp_check, num_scan)
    while True:
        yield env.timeout(random.expovariate(arrival_rate))
        i += 1
        env.process(passenger(env, 'Passenger %d' % i, sys))

# Run the Model
for i in range(REP):
    random.seed(i)
    env = simpy.Environment()

    tot_throughput = []
    check_time = []
    scan_time = []
    sys_time = []

    env.process(setup(env, ARRIVAL_RATE, NUM_BP_CHECK, NUM_SCAN))
    env.run(until=RUN_TIME)

    avg_system_time = sum(sys_time[1:len(tot_throughput)]) / len(tot_throughput)
    avg_check_time = sum(check_time[1:len(tot_throughput)]) / len(tot_throughput)
    avg_scan_time = sum(scan_time[1:len(tot_throughput)]) / len(tot_throughput)
    avg_wait_time = avg_system_time - avg_check_time - avg_scan_time

    AVG_SYSTEM_TIME.append(avg_system_time)
    AVG_CHECK_TIME.append(avg_check_time)
    AVG_SCAN_TIME.append(avg_scan_time)
    AVG_WAIT_TIME.append(avg_wait_time)


print('Average system time = %.2f' % (sum(AVG_SYSTEM_TIME)/REP))
print('Average check time = %.2f' % (sum(AVG_CHECK_TIME)/REP))
print('Average scan time = %.2f' % (sum(AVG_SCAN_TIME)/REP))
print('Average wait time = %.2f' % (sum(AVG_WAIT_TIME)/REP))
```