COVID in the Classroom

By Alan G. Labouseur, *Marist College*

The COVID-19 pandemic presented a vast array of challenges for colleges and universities all over the world. Many of those challenges remain as we navigate COVID's long tail, its wide-spread and long-lasting effects continuing to be felt. Awful as those effects are, being wide-spread and long-lasting makes them both practical and useful as real-world common ground motivation for teaching topics in computer science. In fact, the author has already done so. In addition to teaching in full-time faculty of the Computer Science department, he was a key member of the COVID-19 testing and screening team at Marist College during the Fall 2020 and Spring 2021 semesters, modeling pooled testing protocols in C++ and Java as well as designing and implementing database systems for generating representative samples of the college population for surveillance testing, results tracking, and compliance monitoring. These experiences prompted new, meaningful, hands-on ways to integrate computer science theory with real-world practice in an immediate and authentic manner as the faculty and students were living a common first-hand experience in real time. Even once this pandemic has receded fully into the past, this experience serves as an example of ways we can incorporate events affecting student and faculty lives into computer science courses. This article describes three of those ways.

INTRODUCTION

For decades students have been observed complaining that some of their programming courses lack "real world" applications. Much ink has been spilled (or, to modernize the idiom: many PDFs have been generated) about presenting computer science topics in a fashion relevant to those we teach. We are always looking for authentic, hands-on ways to reach our students. While the COVID-19 pandemic presented many challenges, it also presented new and meaningful ways to inform computer science theory with real-world practice in an immediate and authentic manner. And it's not a one-time thing. In her New York Times article "Reaching 'Herd Immunity' Is Unlikely in the U.S., Experts Now Believe" of May 3, 2021 [5], science reporter Apoorva Mandavilli explained that herd immunity may not be achievable even with widespread vaccinations, and that virus variants will continue to spread, so that "… rather than making a long-promised exit, the virus will most likely become a manageable threat that will continue to circulate in the United States for years to come…". Since it looks like COVID may persist for a while in some form, we might as well make the best of it.

While the COVID-19 pandemic presented many challenges, it also presented new and meaningful ways to inform computer science theory with real-world practice in an immediate and authentic manner. And it's not a one-time thing.

As a member of the COVID testing and screening team at Marist College during the Fall 2020 and Spring 2021 semesters, I modeled pooled testing protocols in C++ and Java. I also designed and implemented database systems for generating representative samples of the college population for surveillance testing. This database recorded test results and monitored testing compliance as well. (I was among the three people at Marist approved for access to this sensitive data.) This experience in real-world COVID testing and tracking revealed new ways to inform computer science theory with practice. The global and pervasive nature of this pandemic, and that methods of addressing it are both theoretical and practical, aligns with best practices specified in the 2020 ACM Computing Curricula in terms of supporting both student and industry use cases for competency-based curricular dynamics. (See CC2020 chapters 5 and 7 for details.) This article describes three use cases drawn from my COVID testing experience that motivate and illustrate core concepts in undergraduate computer science courses. In the first case we will take a brief look at the multi-process bounded-buffer producer-consumer problem often discussed in Operating Systems courses. The second case goes deeper, illustrating var-

ious aspects of SQL, database design, and stored procedure programming in a Database Systems course. The third case is the most thorough: motivating a semester-long project in an Algorithms course. In all cases these courses were taught in synchronous hybrid mode, consisting of partially in-person and partially online class sessions, and thus are widely applicable to a variety of classroom circumstances.

BACKGROUND

Many aspects of the COVID-19 pandemic could be used to motivate examples for teaching computer science topics.

There are epidemiology issues like stochastic or deterministic disease spread models based on population density, proximity, and infection rate. One could see discussing all manner of graph algorithms like computing network density, maximum cliques, connected components, and clustering coefficients in this context. Then there is logistics: from efficiently distributing testing gear and protective equipment to closing transportation pathways for quarantine. Again, more graph algorithms spring to mind, like calculating shortest paths and betweenness centrality for distribution, and determining minimum edge cuts for quarantine. Also, consider all of the statistics involved: given a certain population size and infection rate, what is the likelihood of an outbreak occurring in the next x days? Those are all excellent topics… for another article. This article focuses on a different topic: pooled testing. Before I could use pooled COVID testing to motivate in-class examples I had to introduce the concept.

INTRODUCING POOLED TESTING

In his New York Times column of May 7, 2020 [3], Jordan Ellenberg wrote about a World War II era technique called pooled (or "group") testing. It can be used to reduce the number of tests needed to screen a large population by simultaneously testing multiple samples. Back then there was a different kind of social virus causing trouble: syphilis. US Army economists Robert Dorfman and David Rosenblatt created a group testing method for detecting (and rejecting) syphilitic draft candidates. This old technique has renewed importance in our modern environment of COVID testing.

Many aspects of the COVID-19 pandemic could be used to motivate examples for teaching computer science topics. There are epidemiology issues like stochastic or deterministic disease spread models based on population density, proximity, and infection rate.

Assuming we have an unbiased and uniform population to test, and assuming the test is sufficiently accurate, sensitive, and specific to signal the presence of a COVID virus, combining biomarkers (i.e., pooling samples) from multiple patients into single test can reduce the overall number of tests needed. It's based on a core principle of computer science: divide and conquer. Here's how it works: divide the population into small groups and test (or "conquer" in this metaphor) each group for infection. If the test comes out negative, then nobody in the group is infected. If the test comes out positive then one or more members of the group are infected, but we don't know which one(s) so we must continue by testing everybody in the group or subdividing into smaller groups and repeating the process.

There are three possibilities to consider when testing a pool of samples.

1. There are no infected samples.
2. there is exactly one infected sample.
3. there are two or more infected samples.

Given these three cases, we apply our divide and conquer approach by testing groups of 8 (for example) and then testing subgroups of 4 if any infection is found. The best-case scenario is that we determine all 8 are infection-free with 1 test. See Case (1) in Figure 1. The second case occurs when we find 1 infection with 7 tests: 1 test for the group of eight, 2 tests for subgroups of four, and 4 individual tests for the (single) infected subgroup.

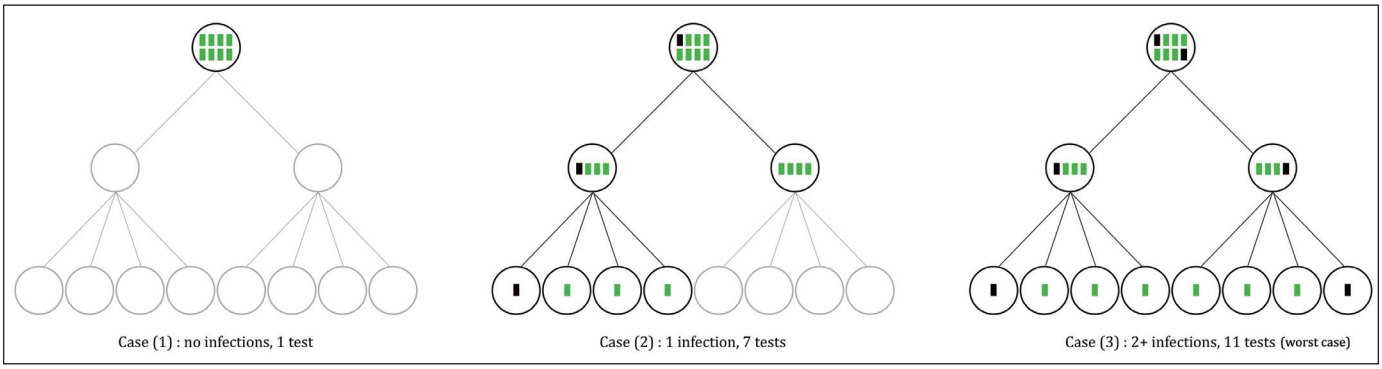


Figure 1: Pooled Testing Cases

See Case (2) in Figure 1. The worst-case scenario occurs when there are two or more infected samples in the group and we divide them into subgroups of 4 and end up using 11 tests: 1 test for the group of eight, 2 tests for the groups of four, and 8 individual tests. See Case (3) in Figure 1. Thankfully, the worst-case scenario is rare when the infection rate is low, but that changes along with the infection rate. (Note that there is overlap between cases (2) and (3) because of the possibility of 2, 3, or 4 infections being found in the **same** subgroup, which is a case (2) situation requiring 7 tests rather than a case (3) situation requiring 11 tests. For now, let's make the simplifying assumption that 2+ infections is always case (3), the worst case. We will revisit this later.)

This high-level overview of pooled testing is all we need to teach our students before moving on to other pedagogical opportunities at the heart of this article. For further details on pooled testing and its applications see Du and Hwang's 2000 book *Combinatorial Group Testing and Its Applications* [2] and Aldridge, Johnson, and Scarlett's 2019 paper *Group Testing: An Information Theory Perspective* [1].

COVID SCREENING ON CAMPUS

A team of healthcare professionals, faculty, staff, and administrators conducted COVID screening via pooled surveillance testing during the Fall 2020 and Spring 2021 semesters. As part of that team, I incorporated three aspects of this endeavor into my classes: the physical logistics of testing; selecting members of the population for testing via representative sampling and then monitoring their compliance; and modeling the pooled testing protocol.

Physical Logistics

Hundreds of people from our community were pseudo-randomly selected for surveillance testing every day. We invited them to report to a specific campus location in a particular window of time so our health care professionals could collect and pool saliva samples for screening. Managing the physical logistics of all of those people flowing through our testing locations provided a timely example of queues and bounded-buffer producer-consumer problems, as we will see in the next section.

Selection and Compliance Monitoring

Generating daily representative samples of hundreds of people from our community and then tracking their compliance—because believe it or not, every day there were people who skipped their test—provided a vivid landscape for discussing SQL queries and views, relational database design, and stored procedure programming, which we will explore in the Database Systems section.

Modeling the Protocol

Programming a model of our pooled testing protocol and statistically determining the expected test usage provided an ideal opportunity to explore data structures and discuss statistics, which we will see in the Algorithms section.

OPERATING SYSTEMS

Deep into the semester, once material required for the semester-long project [4] had been covered, my students and I turned our attention to some classic problems of managing multiple processes: locking, blocking, deadlock, semaphores, critical sections, and the bounded-buffer producer-consumer problem (see chapter 5.7 in the 9th edition of *Operating System Concepts* by Silberschatz, Galvin, and Gagne) as specified in the "OS/Concurrency Core-Tier2" component of the ACM Computer Science Curricula 2013 Body of Knowledge. For binary semaphores, selling concert tickets online makes an excellent example of mutual exclusion and the need for critical sections. (See chapters 4.2 and 4.3 in *Operating Systems and Middleware* by Max Hailperin.) For bounded-buffer producer-consumer problems, one typically presents a fast-food restaurant environment where a chef produces food items into a bounded-buffer holding area while customers consume items from that buffer by purchasing them from a cashier. The food item buffer (a queue) is bounded by the number of items it can hold so we can use counting semaphores to avoid overflow or underflow. After covering these examples, I introduced the COVID context and presented a screening environment where people in our community enter a bounded-buffer holding area (a queue) while health care professionals "consume" people from that buffer by administering their COVID tests. The people buffer is bounded by the number of people it can hold so we can use counting semaphores to avoid overflow or underflow.

By the time this came up in the semester most of my students had been through this screening process several times. It was easy for them to identify with this scenario and visualize themselves "in the buffer," seeing the problems of overflow (they would have to wait outside) and underflow (the health care professionals would have nothing to do). Not only does this scenario illustrate similar points as more traditional examples in an immediate and authentic-to-real-life way, but it also lends itself to additional complexity by varying the buffer capacity based on changing social distancing guidelines.

DATABASE SYSTEMS

At the beginning of each semester, I was given a snapshot of student and employee data from *Banner*, our higher education ERP system, which I imported into the PostgreSQL object-relational database in a table called *People*, shown in Figure 2. A month or two into surveillance testing it became clear that more attributes were needed, which I added, as noted in Figure 2c.

With the People data in place—and purposefully ignoring any design issues until later—my students and I began discussions of elementary SQL. There is a plethora of obvious and simple `select - from - where` queries to use here, e.g., on-campus computer science majors:

```
select lastName, firstName
from People
where isStudent
  and livesOnCampus
  and major = 'Computer Science'
```

To illustrate the utility of views, we created them for *OnCampusStudents*, *OffCampusStudents*, and *Employees*. Then I had my students look at some of the same queries from earlier with instructions to use the new views instead of base tables:

```
select lastName, firstName
from OnCampusStudents
where major = 'Computer Science'
```

This led to a discussion of view definitions being stored in the system catalog and an illustration of query rewriting. We would revisit this table in the context of advanced SQL when discussing stored procedures later in the semester. But before we could do that, we would have to take up relational database design and address some of the pressing deficiencies of the People table.

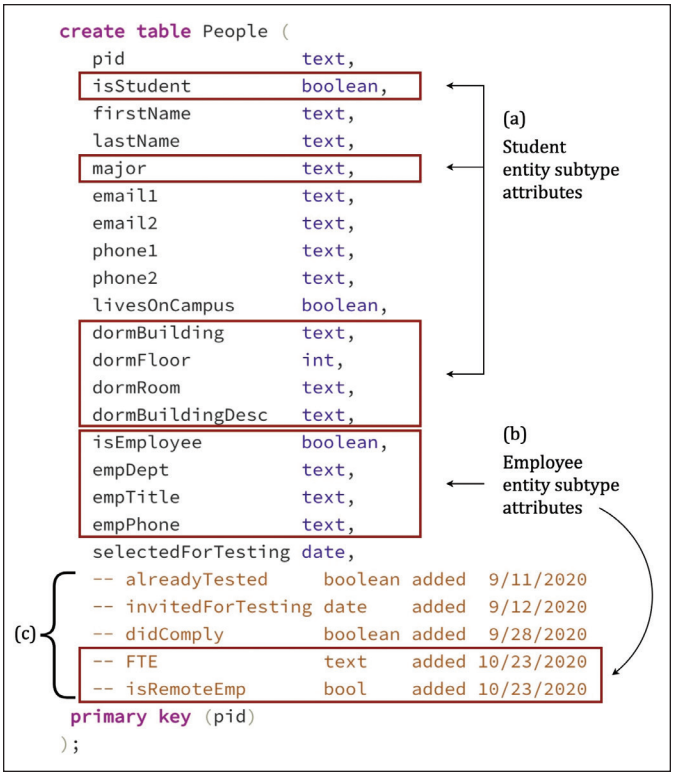


Figure 2: The People table

Relational Database Design

The deficiencies of the People table were made clear by the presence of NULLs in the data due to some attributes being relevant only to certain rows. With this in mind we began our database design discussions. After covering the normal forms and developing an appreciation for Codd and his rules, we looked to revise the **People** table. It was apparent by this point that there are multiple (sub)types of people and that we have a few different attributes for each. The fact that student-only and employee-only views eased query writing provided an excellent opportunity to introduce entity subtypes and their implementation as one-to-one relationships with optional participation on the subtype side. (This also proved to be a good time to note the use of a primary key simultaneously as a foreign key in the subtype tables.)

We factored out the student and employee attributes from the **People** table as noted in Figures 2a and 2b and into subtype entities. Having done this, we noted that we no longer needed the **`isStudent`** and **`isEmployee`** attributes, that functionality now being accomplished via **`pid`** membership in the new subtype tables. This led to a discussion of logical data independence, demonstrated by rewriting the *OnCampusStudents*, *OffCampusStudents*, and *Employees* views to make use of the new subtype tables and then executing unchanged our earlier queries against those revised views.

Stored Procedure Programming

With SQL queries, views, and design issues behind us, we considered how to go about generating representative samples of our community for COVID screening. Figure 3 shows (most of) the PL/pgSQL code for pseudo-randomly selecting on-campus students who had not been recently tested or recently selected for testing, grouped by dormitory clusters. (The code for selecting off-campus students and employees is substantially similar and excluded in the interest of space and anti-redundancy.) This code was used to demonstrate many features of stored procedures: local variable declaration and usage, date functions, (nested) **`for`** loops, advanced SQL inside of control structures, table manipulation, and the need for good formatting and comments to make sense of it. My students felt strongly connected to this because, by this time in the semester, all of them had been selected for COVID screening by this very code.

```
-- Asymptomatic surveillance COVID screening
-- The PostgreSQL random() function is based on the POSIX-standard erand48() family.
-- It's not good enough for cryptography, but it's plenty good for us.
create or replace function choosePeopleToTest()
   returns table (
        ...
    )
language plpgsql
as
$$
declare
   currentCluster       text;
   clusterRow           record;
   dormRow              record;
   goBackAtLeastThisFar date;
begin
   -- Establish the date prior to which we'll consider re-inviting people for testing.
   goBackAtLeastThisFar := (current_date - interval '13 days');
   for clusterRow in ( select distinct dayNameToDayNumber(d.cluster), d.cluster
                       from Dorms d
                       order by dayNameToDayNumber(d.cluster) ) loop
      currentCluster := clusterRow.cluster;
      for dormRow in (select building, numToTest
                      from Dorms d
                      where d.cluster = currentCluster
                      order by d.building) loop
         insert into workTable
         select p.pid,
                ...
         from People p inner join Dorms d on p.dormBuilding = d.building
         where p.livesOnCampus
           and p.isStudent
           and ( (p.selectedForTesting is null)
                 or
                (p.selectedForTesting < goBackAtLeastThisFar) )
           and p.dormBuilding = dormRow.building
         order by previouslyTested ASC, random()
         limit dormRow.numToTest;
      end loop; -- for dormRow
      -- This completes one cluster. Loop to the next.
   end loop; -- for clusterRow
   -- That's it. We're done. Return the results.
   return query (select * from workTable);
end;
$$
```

Figure 3: Stored Procedure

All told, this database material addressed many areas of the "IM/Database Systems and IM/Data Modeling Core-Tier2" component of the ACM Computer Science Curricula 2013 Body of Knowledge in a practical way.

The other student, an Applied Mathematics and Data Science double major, noticed this issue right away and documented it in her write up. Future versions of this project will include an extra credit portion addressing this, to be completed once everything else is perfect.

ALGORITHMS

The Algorithms course at Marist could more accurately be called Algorithms and Data Structures. Given that the entire school was undergoing surveillance testing using a pooled testing protocol, it was natural to focus on examples and scenarios from that realm when exploring algorithms utilizing data structures such as lists, stacks, queues, and trees. After discussing the physical logistics of surveillance testing in terms of students *queueing* in the holding area while health care professionals administered COVID tests from their *stack* of test kits, I wanted to introduce the semester-long project. But first I would need to cover some statistics.

Statistics

After explaining our pooled testing protocol (remember, there are three possibilities to consider: no infected samples, exactly one infected sample, and two or more infected samples) as described earlier and shown in Figure 1, we discussed the expected test usage for a given population and a given infection rate under that protocol. Here are some highlights of that discussion:

> Why do we think this works? We can determine the likelihood of each testing possibility based on the number of samples we pool into each group and the infection rate of the disease. For example, when testing groups of 8 for a disease with an infection rate of 2% using the protocol described here…
>
> **Case (1)** is expected to happen 85% of the time. This is because a 2% infection rate means that, on average, 98% of the population is uninfected. The likelihood of randomly choosing 8 uninfected people is 0.988, which is 0.8507 or roughly 85%. When this occurs only one test is needed.
>
> **Case (3)** happens slightly less than 0.04% of the time because the likelihood of randomly choosing two infected samples is 0.02^2 or 0.0004 or 0.04%. (It's actually less, but it's safe to err on the side of an upper bound. Also, the likelihood of randomly choosing more than two infected samples is even lower, so again we are safe with this upper bound.) In this case, at most 11 tests are needed.
>
> **Case (2)** is the only other possibility, which happens the rest of the time, which is 14.96%, and 7 tests are needed.
>
> So, for 1000 people where 20 of them (2%) are infected and 980 are infection-free, we could make 125 pools of 8 samples each and work out what we expect based on the percentages we just calculated:

```
Case (1): 125 x 0.8507 = 106.33 instances
          requiring 107 tests (no partials)
Case (2): 125 x 0.1496 = 18.70 instances
          requiring 131 tests
Case (3): 125 x 0.0004 = 0.05 round to 1 instance
          requiring 11 tests
-------------------------------------------------
That's 249 tests to screen a population of 1000
people for a disease with an infection rate of 2
```

> *Note:* This assumes 100% testing accuracy. Since tests are rarely perfect, it would be wise to incorporate test reliability into the model by introducing conditional probability and Bayes' theorem. But that's for another class.

As noted earlier in the section introducing pooled testing, there is overlap between cases (2) and (3) because of the possibility of 2, 3, or 4 infections being found in the same subgroup. When that happens, we have a case (2) situat**ion r**equiring 7 tests rather than a case (3) situation requiring 11 tests. We made the simplifying assumption that 2+ infections would always require 11 tests. This makes sense in an Algorithms class where we exclude constant factors and evaluate worst-case scenarios when talking about asymptotic performance, and I did not address this with my students. In the end, two students noticed this issue. One noticed a discrepancy between the predicted and actual values produced by his simulation when he ran his code on populations of 100,000 or 1M people. The other student, an Applied Mathematics and Data Science double major, noticed this issue right away and documented it in her write up. Future versions of this project will include an extra credit portion addressing this, to be completed once everything else is perfect.

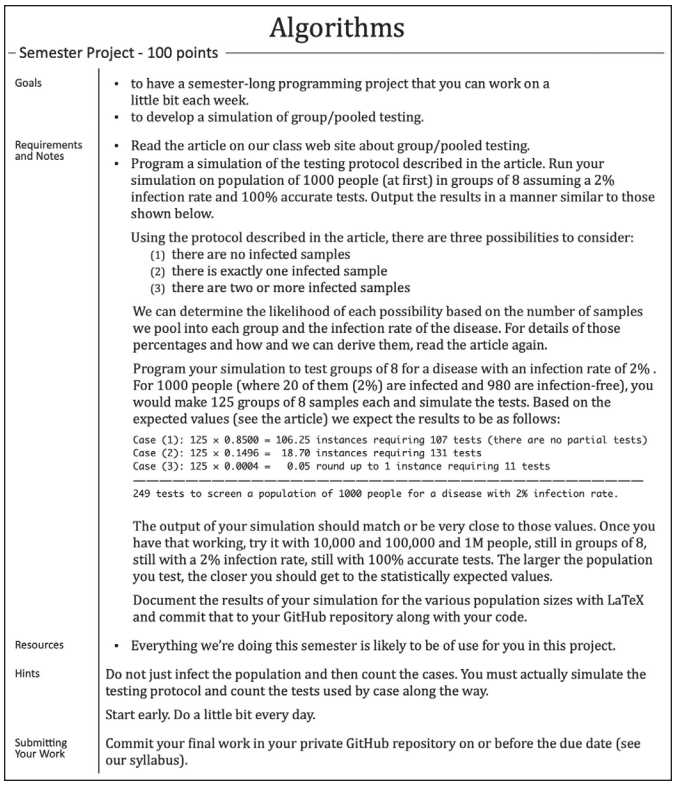
Another thing I did not address with my students is the subtle fudge in the expected values for test usage. The expected values we calculated were based on selection **with** replacement (binomial distribution), which is not the case when creating a representative population sample. Once a person is selected for testing, they cannot be selected again in the same sample. To be more accurate I could have done the calculations based on selection **without** replacement (hypergeometric distribution). For example, the likelihood of selecting 8 healthy people in a population of 100 given a 2% infection rate is slightly less than 0.98^8 (which is 0.8507). It's actually

$$\frac{98}{100} * \frac{97}{99} * \frac{96}{98} * \frac{95}{97} * \frac{94}{96} * \frac{93}{95} * \frac{92}{94} * \frac{91}{93} = 0.8456$$

To be even more accurate I could have used our actual population size of 6000 because, while we don't expect exactly two positives in every group of 100, we do expect 120 positives in 6000. In the end, no students discovered this (not even our Applied Mathematics and Data Science major), which is a little disappointing. This too seems like a nice extra credit enhancement for future instances of the semester-long project.

The Semester-long Project

With the statistics out of the way, we could move on to our semester-long project simulating a pooled testing protocol in Java or C++. (Most students chose Java, but a few chose C++.) This assignment, the details of which can be seen in Figure 4, was given early in the semester and the students were invited to submit questions and comments before they got started. I collected my students' questions and posted answers to the entire class. Here are some highlights:



Algorithms

– Semester Project - 100 points

Goals

- to have a semester-long programming project that you can work on a little bit each week.
- to develop a simulation of group/pooled testing.

Requirements and Notes

- Read the article on our class web site about group/pooled testing.
- Program a simulation of the testing protocol described in the article. Run your simulation on population of 1000 people (at first) in groups of 8 assuming a 2% infection rate and 100% accurate tests. Output the results in a manner similar to those shown below.

Using the protocol described in the article, there are three possibilities to consider:
(1) there are no infected samples
(2) there is exactly one infected sample
(3) there are two or more infected samples

We can determine the likelihood of each possibility based on the number of samples we pool into each group and the infection rate of the disease. For details of those percentages and how and we can derive them, read the article again.

Program your simulation to test groups of 8 for a disease with an infection rate of 2% . For 1000 people (where 20 of them (2%) are infected and 980 are infection-free), you would make 125 groups of 8 samples each and simulate the tests. Based on the expected values (see the article) we expect the results to be as follows:

```
Case (1): 125 × 0.8500 = 106.25 instances requiring 107 tests (there are no partial tests)
Case (2): 125 × 0.1496 =  18.70 instances requiring 131 tests
Case (3): 125 × 0.0004 =   0.05 round up to 1 instance requiring 11 tests
————————————————————————————————————————————————————————
249 tests to screen a population of 1000 people for a disease with 2% infection rate.
```

The output of your simulation should match or be very close to those values. Once you have that working, try it with 10,000 and 100,000 and 1M people, still in groups of 8, still with a 2% infection rate, still with 100% accurate tests. The larger the population you test, the closer you should get to the statistically expected values.

Document the results of your simulation for the various population sizes with LaTeX and commit that to your GitHub repository along with your code.

Resources

- Everything we're doing this semester is likely to be of use for you in this project.

Hints

Do not just infect the population and then count the cases. You must actually simulate the testing protocol and count the tests used by case along the way.

Start early. Do a little bit every day.

Submitting Your Work

Commit your final work in your private GitHub repository on or before the due date (see our syllabus).

Figure 4: Semester Project

Q: *What data structure should I use to store groups of 8?*

A: An array or a list of some sort would be good. Actually, you could hold the entire population to be tested in an array or a list or a vector.

Q: *Does our program have to come up with the likelihood of each case occurring (0.85, 0.1496, 0.0004) or can we set them as constants?*

A: Your program/simulation should produce those numbers as output by counting the number of occurrences of each case. You'll know you've got it right when your output gets close to the statistically expected values.

Q: *Should the total number of people being tested be input by the user or set in the code?*

A: To avoid complicating things with I/O, take a command-line parameter for the population size. If the program is called "sim" then …

```
> sim 1000
```

…would run the simulation on 1000 people, while…

```
> sim 1000000
```

… would run the simulation on 1 million people.

Q: *Will randomness be a part of our program?*

A: Yes. You'll need to use a random() function. There is probably one built into your programming language.

Q: *How will individual positive or negative people be represented? (Ex. 1s and 0s ?)*

A: It's up to you, but I like 1s and 0s.

Q: *What's the best way for those 1000 people to be broken up into groups of 8 and then inserted into a data structure?*

A: Randomly. Kinda. It's actually not all that important. Let's say you are running the simulation on 1000 people, take the first 8 as a group, then the next 8 as a group, and so on through the 125th group.

Q: *If the theoretical approach to the pooled testing assumes 100% testing accuracy, would the entire nature of the simulation fall apart due to the inaccuracies of actual testing?*

A: No. A simulation that accounts for test accuracy would be more accurate than one that assumes 100% accuracy. But we've got to start somewhere. For now, assume 100% test accuracy. Once you've got that simulation working, consider taking into account test accuracy as a refinement later.

Q: *I am under the impression that we should be using a binary tree to isolate the infected individuals. I feel like this is also how you described it. However, at certain points during the discussion I got the impression that we should split the initial population into 125 groupings of 8. Which is it?*

A: t's a subtle point that I **could** have made better. We could build a binary tree and use its nodes as containers for the tests, as seen in [Figure 1]. But we don't need to. I used a binary tree as a conceptual representation to explain our divide and conquer approach. If we test a group of 8 and there's an infection then we'll test groups of 4 and 4, and when there is one or more infection(s) from those tests (as there must be) then we'll do individual tests. The process is conceptually similar a binary tree, but as a practical matter of code, we can test `population[0] - population[7]` as one test, and if it's positive, use `population[0] - population[3]` and `population[4] - population[7]` for the next two tests.

Based on the penultimate question I will likely add a third extra-credit option to future projects—an option that considers test accuracy and tracking false positives and false negatives. If one wants to go even further along these lines, this seems like a good place to introduce Bayes' theorem.

This material, presented throughout the Algorithms class, addressed many areas of the "AL. Algorithms and Complexity Core-Tier1" component of the ACM Computer Science Curricula 2013 Body of Knowledge.

CONCLUSION AND FUTURE ACTIVITIES

It is no surprise that circumstances affecting all of us around the world should be well suited to motivate various topics and assignments in our computer science classes. After only two semesters, my experience bears this out, and it aligns with the opinions of my students. This quote from a student at the end of the Fall 2020 semester puts it nicely.

> *I wanted to thank you for an interesting and academically engaging semester. This class really sharpened a lot of my core software development skills that I felt I had been missing. But most of all I really appreciated how you connected the material in the course to the pandemic instead of just pretending it was a regular semester. The connection of CS courses to the real world has been something I have wanted to see more of in my classes so I really enjoyed that factor of the semester project.* - Maria

SOURCE CODE

Much of the SQL and PL/pgSQL source code (but none of the data) for the database is available on GitHub at `https://github.com/Labouseur/CovidInTheClassroom`, as is the entirety of my C++ code for the pooled testing simulation as well as a Java version from one of my students. You are welcome and encouraged to use it… and to improve it.

FUTURE ACTIVITIES

In the near-term it makes sense to analyze vaccination distribution plans using graph analytics like clustering coefficient and maximum cliques so that those who are most connected—the "social butterfly super-spreaders"—are among those with the easiest access to vaccines. Looking further into the future, the wide-spread and long-lasting effects of the COVID-19 pandemic will be with us throughout its long tail. This unfortunate fact means that the pandemic will remain useful as "real world" common ground motivation for teaching subjects in computer science for quite some time. In addition to the topics in operating systems, database systems, and algorithms mentioned here, there are many other areas to consider: teaching graph analytics with examples from disease spread models or network density; illustrating pathfinding for protective equipment distribution with the algorithms of Dijkstra, Kruskal, Prim, and those of Bellman-Ford; reasoning about statistics by modeling transmission and infection, and more. While this "silver lining" is not worth the exorbitant cost of the COVID "cloud," we can make the best of it. ❖

It is no surprise that circumstances affecting all of us around the world should be well suited to motivate various topics and assignments in our computer science classes. After only two semesters, my experience bears this out, and it aligns with the opinions of my students.

Acknowledgements

I sincerely thank Dean Roger Norton of the School of Computer Science and Mathematics for suggesting the use of my COVID testing simulation as the basis for this paper. Heartfelt thanks also go out to Dean Alicia Slater of the School of Science for trusting me with all that sensitive data, answering every question I had, and allowing me to be her sidekick in fighting this pandemic. Thanks also to Hope Neveux from my Algorithms class, who contributed her Java-based semester project to the GitHub repository noted earlier. And finally, a thank you to the Inroads reviewers for their thoughtful comments.

References

1. Aldridge, M., Johnson, O., and Scarlett, J. Group testing: An information theory perspective. *Foundations and Trends in Communications and Information Theory* 15, 3-4 (2019), 196–392.
2. Du, D., and Hwang, F. *Combinatorial Group Testing and Its Applications*. Series on Applied Mathematics, vol 3. World Scientific, 2000.
3. Ellenberg, J. Five people. One test. This is how you get there. *New York Times*, May 7, 2020. https://www.nytimes.com/2020/05/07/opinion/coronavirus-group-testing.html. Accessed 2021 May 6.
4. Labouseur, A. A Browser-based Operating Systems Project: JavaScript Adventures in Dinosaur Slaying. *SIGCSE Bull*. 41, 4 (2009), 71–75.
5. Mandavilli, A. Reaching 'Herd Immunity' Is Unlikely in the U.S., Experts Now Believe. *New York Times*, May 3, 2021; https://www.nytimes.com/2021/05/03/health/covid-herd-immunity-vaccine.html. Accessed 2021 May 6.

Alan G. Labouseur
Marist College
School of Computer Science and Mathematics
3399 North Road, Poughkeepsie, NY 12601 USA
Alan.Labouseur@Marist.edu

DOI: 10.1145/3477055