1. 100- the total number of artificial errors seeded
   75-number of artificial faults founded
   Let r be the ratio of the number of artificial faults found to the number of total artificial faults. r=75/100=0.75. This shows as we left with 25 faults. If we uncovered that many real errors during testing, we should not stop at this point because we still have errors that may affect the quality of results of the program and would apply another testing method for better efficiency.

2. Error seeding is the process of intentionally introducing error within a program to check whether the test cases can capture the seeded error or. Error seeding can be applied iteratively by creating artificial faults at each testing phase, but it is not valuable because the accuracy of the measurements depends on how faults are introduced, which result in being uncertain how they are equivalent to inherent faults in difficulty of detection. Artificial faults are manually planted; therefore, it would be difficult and time taking to implement them iteratively since at each testing phase we need to introduce artificial faults with increasing their complexity gradually. Therefore, it could be better if we use Mutation testing since it introduces faults into a program more systematically.

3. One of the weaknesses of error seeding is that the accuracy of measuring the quality of software testing is dependent on how faults are introduced. Since artificial faults are manually planted, they are not easy to implement in practice, and It is not easy to introduce artificial faults that are equivalent to inherent faults in difficulty of detection. Even though it requires less time and economical to perform, error seeding is a less efficient error testing technique.

   - I would use an error seeding technique at the early stage of testing when creating artificial faults are not that complicated. However, over time the system gets more complicated, and we would be exposed to more advanced testing levels where inherent faults also get more complicated. Therefore, it would not be easy to introduce artificial faults that are equivalent to them. This can be considered as exhausting and time-consuming, and it can lead to making these artificial faults difficult to be detected.

4. In the given scenario, there are 10000 mutants, and based on our assumption, we have 20 elements in the test set. In each test case, the testing system executes the original program and each mutant and compares their output together. This will give results in 10000 * 20 execution time. Then each mutant found the output to be different from the output of the original program that needs to be marked dead. When finishing execution, the tester examines mutants that are not marked dead. The tester would then declare these mutants to be equivalent to the original program or produce additional test data. Also, the tester should supply the program to be tested, and

chooses the levels of mutation analysis to be performed as well as the tester supplies the test set.

5. The two assumption behind the mutation testing are explained below.

   5.1 "The coupling effect hypothesis assumes that simple and complex errors are coupled, and hence test data that cause simple non-equivalent mutants to die will usually also cause complex mutants to die".

   - I believe the assumption might be valid, but not always the case. In some scenarios, there are systems written by languages that have no garbage collector. In this scenario, the system would experience exhaustion and might be run out of resource which can cause the system to stop working. To solve this issue the programmer must include deallocation of these resources in the code. But if he/she forgot to do so, it will lead to errors.

   5.2 The other assumption is a competent programmer hypothesis. Here the hypothesis assumes that a competent programmer writes the program to be tested. The programmer creates programs that are close to being correct. This means that the mutants have a small deviation from the original program only to be considered mutation analysis. These mutants are created systematically and mechanically by applying a set of transformations to the tested program. These mutation operators would ideally model programming errors made by programmers. This assumption might not be correct always. In a team of programmers, all the team members will not have the same magnitude. We can't guarantee that all the programmers will write the same way and correctly.

6. I would use mutation testing only for small projects. Since mutation testing iterates through each line of implemented code, and running unit tests in which, this consumes both time and space, I would not prefer to use it for large projects. In addition to this, it requires large human resources for examining many mutants for possible equivalence. Therefore, I prefer to use this testing technique only for a small project, not for large projects.

Reference

1. Zhu, H., Hall, P. A. V., & May, J. H. R. (1997). Software unit test coverage and adequacy. ACM Computing Surveys, 29(4), 366–427. doi: 10.1145/267580.267590