Randomized Algorithms Takehome Exam

Magnus Erik Hvass Pedersen (971055)
March 2005, Daimi, University of Aarhus

Introduction

The purpose of this document is to verify attendance of the author to the *Randomized Algorithms* course, at Department of Computer Science, University of Aarhus.

The document is believed to solve the questions posed in the exam, to the expected degree of completion, with the exception of the last 5% question for Problem 1. The reader is assumed to be familiar with the course litterature and the problem description for this exam in particular.

Problem 1

We are considering m random guesses, where the number m is given by:

$$m = \frac{3}{5}n$$

which corresponds to 60% of the total number of test-questions n. The random value X_i resulting from each of these guesses, is defined as follows, depending on whether the guess was correct or not:

$$X_i = \begin{cases} 1, & \text{if the } i\text{'th guess is correct} \\ 0, & \text{if the } i\text{'th guess is wrong} \end{cases} \tag{1}$$

which is a socalled *indicator* variable, from which we define the sum:

$$X = \sum_{i=1}^{m} X_i$$

Subproblem 1.1

The expected value of X is found by using *linearity of expectations* on the definition of X as follows:

$$E[X] = E\left[\sum_{i=1}^{m} X_i\right] = \sum_{i=1}^{m} E[X_i]$$

Since each question in the test has 4 choices, of which only one choice is correct, each X_i either takes on the value 1 with probablity 1/4, or the value 0 with probability 3/4. We therefore have:

$$E[X_i] = \frac{1}{4} \cdot 1 + \frac{3}{4} \cdot 0 = \frac{1}{4}$$

and hence:

$$E[X] = \sum_{i=1}^{m} E[X_i] = \sum_{i=1}^{m} \frac{1}{4} = \frac{1}{4} \cdot m = \frac{3}{20} n$$

corresponding to our intuition, that one fourth of the m guesses were expected to be correct.

Subproblem 1.2

First define Z_i similarly to Eq.(1), but now with the proper points for right and wrong guesses:

$$Z_i = \begin{cases} 2, & \text{if the } i\text{'th guess is correct} \\ 1, & \text{if the } i\text{'th guess is wrong} \end{cases}$$

for $i \in \{1, \cdots, m\}$. Let Z be as follows:

$$Z = \sum_{i=1}^{m} Z_i$$

Since we clearly have:

$$Z_i = 3 \cdot X_i - 1$$

we find:

$$Z = \sum_{i=1}^{m} Z_i = \sum_{i=1}^{m} (3 \cdot X_i - 1) = 3 \cdot X - m$$

Still assuming 60% of the answers are pure guesses for a challenged student, and the remaining 40% are correct answers, the total score for a challenged student is given by:

$$S = 2 \cdot \frac{2}{5} \cdot n + Z$$

For any student to pass the exam, a score $\geq n$ is required. That is, we require:

$$S \geq n \;\Leftrightarrow\; 2 \cdot \frac{2}{5} \cdot n + 3 \cdot X - m \geq n$$

which means:

$$X \geq \frac{4}{15} n \tag{2}$$

Now looking at the relationship between the expectancy for X and the right-hand side of this equation, we find:

$$\alpha \cdot E[X] = \frac{4}{15} n \;\Leftrightarrow\; \alpha = \frac{\frac{4}{15} n}{\frac{3}{20} n} = \frac{16}{9}$$

and therefore:

$$X \geq \frac{16}{9} E[X]$$

Subproblem 1.3

We now wish to decide the number of questions n, for which a challenged student only passes with a probability of most 5%. In other words, we wish to determine n so that:

$$\Pr\left[X \geq \frac{16}{9} \cdot E[X]\right] \leq 0.05$$

Rename the expectancy as:

$$\mu = E[X] = \frac{3}{20}n$$

and define δ from the above deduction of α:

$$1 + \delta = \frac{16}{9} \;\Leftrightarrow\; \delta = \frac{7}{9}$$

We then have the upper (Chernoff) bound for the desired probability:

$$\Pr[X > (1+\delta)\mu] < F^+(\mu, \delta) = \left[\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right]^\mu = \left[\frac{e^{7/9}}{(16/9)^{(16/9)}}\right]^{n \cdot 3/20}$$

which approximately equals:

$$F^+(\mu, \delta) \simeq 0.9639^n \tag{3}$$

For this bound to be less than or equal to 0.05, we would therefore have:

$$F^+(\mu, \delta) \leq 0.05 \;\Leftrightarrow\; 0.9639^n \leq 0.05 \;\Leftrightarrow\; n \geq \frac{\ln(0.05)}{\ln(0.9639)} \simeq 81.49$$

or as we require n to be an integer, we would need:

$$n \geq 82$$

questions for the test to be good – that is, when using the Chernoff technique to decide n.

Subproblem 1.4

Consider R simulations of the above for different n's, and define the stochastic variable:

$$Y_{n,i} = \begin{cases} 1, & \text{if the } i\text{'th simulated and challenged student passes a test with } n \text{ questions.} \\ 0, & \text{if the student fails the given test.} \end{cases}$$

with $i \in \{1, \cdots, R\}$. Define the probability of success p_n, for a challenged student taking a test with n questions, as follows:

$$p_n = \Pr[Y_{n,i} = 1]$$

which is equivalent to:

$$p_n = \Pr\left[X \geq \frac{16}{9}E[X]\right]$$

from the above, and with n being implicitly given in X. Define the sum of the trials $Y_{n,i}$ as follows:

$$Y_n = \sum_{i=1}^{R} Y_{n,i}$$

whose expectancy is then:

$$E[Y_n] = \cdots = \sum_{i=1}^{R} E[Y_{n,i}]$$

and since:

$$E[Y_{n,i}] = p_n \cdot 1 + (1 - p_n) \cdot 0 = p_n$$

which are independent of the trial i, we have:

$$E[Y_n] = \sum_{i=1}^{R} p_n = R \cdot p_n$$

Subproblem 1.5

The teacher decides to accept the test with n questions as being *good*, if:

$$Y_n \leq 0.04 \cdot R$$

Consider then the probability that a *non-good* test (as defined previously by the relation $p_n > 0.05$), is mis-classified as a *good* test:

$$\Pr[Y_n \leq 0.04R \mid p_n > 0.05]$$

We then wish to decide the number of simulations R, so that this probability is below 0.05, that is:

$$\Pr[Y_n \leq 0.04R \mid p_n > 0.05] \leq 0.05 \tag{4}$$

So let us assume we are considering a non-good test, i.e. $p_n > 0.05$, and then focus on the probability:

$$\Pr[Y_n \leq 0.04R]$$

where we may express $0.04R$ in terms of the expectancy $\mu = E[Y_n] = R \cdot p_n$ by noting:

$$\beta \cdot E[Y_n] = 0.04 \cdot R \;\Leftrightarrow\; \beta = \frac{0.04 \cdot R}{R \cdot p_n} = \frac{0.04}{p_n}$$

and therefore:

$$0.04R = \frac{0.04}{p_n}E[Y_n]$$

Then define:

$$1-\delta = \frac{0.04}{p_n} \;\Leftrightarrow\; \delta = 1 - \frac{0.04}{p_n}$$

which is clearly in the range $(0, 0.8]$, as we assumed $p_n > 0.05$. Using this, we get:

$$\Pr[Y_n \leq 0.04R] = \Pr[Y_n \leq (1-\delta)\mu]$$

which may be bounded by the Chernoff bound $F^-(\mu,\delta)$ as follows:

$$\Pr[Y_n \leq (1-\delta)\mu] < F^-(\mu,\delta)$$

where $F^-(\mu,\delta)$ is given by:

$$F^-(\mu,\delta) = e^{(-\mu\delta^2/2)} = e^{-R\cdot p_n\cdot(1-0.04/p_n)^2/2}$$

which can be rewritten into:

$$F^-(\mu,\delta) = e^{-R\cdot(p_n+0.04^2/p_n-0.08)/2} \tag{5}$$

Back to the question; how many simulations R, do we require in order for this probability to be less than or equal to 0.05. To find this R, we use the Chernoff bound, well knowing, that as it is merely a bound, and not the real value, we might get a much too high R. However, this will merely mean that the actual probability is even lower, so a too large R will not have a negative affect on our confidence in the simulation. We have:

$$F^-(\mu,\delta) < 0.05 \;\Leftrightarrow\; e^{-R\cdot(p_n+0.04^2/p_n-0.08)/2} < 0.05$$

and rewriting this, we obtain:

$$R > -\frac{2\cdot\ln(0.05)}{p_n + 0.04^2/p_n - 0.08}$$

and from our assumption that $p_n > 0.05$, we have:

$$R > -\frac{2\cdot\ln(0.05)}{p_n + 0.04^2/p_n - 0.08} > -1000\cdot\ln(0.05) \simeq 2995.73$$

which means we require 2996 simulations for each n. It should be noted, that the poorer the test, the more likely it is, to be exposed as such during these 2996 simulations.

Problem 2

Implementation of the previously described simulations, is carried out in *MS Visual C++ .NET* and compiles to an *MS Windows* executable. The (pseudo)-random number generator (RNG), is `ran2` from [2, p. 282], which is claimed to have a period greater than 2×10^{18}, and is not known to exhibit any statistical correlation. The class `LYggRandom2` implements this RNG, and the file `ra_exam_problem2.cpp` contains all the other source-code, for the specific project at hand.

Subproblem 2.1

The first function simulates a sequence of m guesses by a challenged student, and returns the corresponding value for X. The function is as follows:

```
int SimulateX(int m)
{
    int X = 0;

    for (int i=0; i<m; i++)
    {
        if (gRandom.RandUni() < 0.25)
            X++;
    }

    return X;
}
```

where the call to `gRandom.RandUni()` returns a uniformly distributed random value in the range $(0, 1)$. This function is called for each of the R simulated students that make up Y_n. The function doing this, is as follows:

```
int SimulateY(int n, int m)
{
    int Yn = 0;

    for (int i=0; i<kR; i++)
    {
        int X = SimulateX(m);
        int score = GetScore(n, m, X);

        if (score >= n)
            Yn++;
    }

    return Yn;
}
```

with n being the number of questions in the test, and m the number of those questions that a challenged student will guess. The function that computes the score for a challenged student with X correct guesses, is given by:

```
int GetScore(int n, int m, int X)
{
    // A correct answer scores 2 points.
    // A wrong answer scores -1 point.
    return 2*(n-m) + 3*X - m;
}
```

where the first term $2 \cdot (n - m)$ is the score for the answers that were known to be correct. These functions are used in the next section.

Subproblem 2.2

To determine the smallest n needed for a *good* test (as defined in the first problem of this exam), we use the function `SimulateY()`, to compute Y_n by simulating R students, for the number of questions n going from 1 and upwards, until we find one for which $Y_n \leq 0.04R$. Although we know that the loop will probabilistically terminate, it may seem appropriate to safe-guard it with a deterministic termination condition. The user is therefore asked to supply a maximum allowed value for n. The loop is then:

```
double ratio = 1;

for (int i=1; i<=gMaxN && ratio>0.04; i++)
{
    int m = std::ceil(3*i/5.0);
    int Yn = SimulateY(i, m);
    ratio = (double)Yn/kR;

    std::cout << "..." << std::endl;
}
```

where the `std::cout` portion is merely there for displaying the progress. Note the calculation of `m`. The theoretical problem as posed in the exam, does not assume m to be an integer, and it is implicitly implied that $m \in \mathbb{R}$. Naturally, we require m to be an integer, and must therefore decide whether to round up or down, that is, should m be determined as:

$$m = \left\lceil \frac{3}{5} n \right\rceil$$

or

$$m = \left\lfloor \frac{3}{5} n \right\rfloor$$

and of course assuming $n \in \mathbb{N}$ also. As can be understood intuitively, there is a difference between the two, which is indeed confirmed in subproblem 2.4 below. The above source-code uses the ceiling (round up) function `std::ceil()`.

It is important to note, that due to floating point rounding errors, the expression `3*i/5.0` must be computed that way instead of e.g. `3.0*i/5.0`, as the latter will cause the `std::ceil()` function to incorrectly (mathematically speaking), round up in cases where it should instead just truncate the number. For example when `i=10`, the result would be 7 instead of 6. In the expression used here, we first compute an integer for `3*i`, then implicitly promote to a floating point number, and then perform floating point division by 5.0. The latter and unused approach, would start by promoting `i` to a floating point number, then perform floating point multiplication with 3.0, and finally perform floating point division by 5.0.

Also note that R is hard-coded as a constant integer `kR`, but the formulae from subproblem 1.5, could be generalized to allow for other user-supplied probability limits, than the given 0.04, and 0.05, etc.

Subproblem 2.3

Assuming the RNG is of high quality, and its pseudo-random numbers therefore approach true randomness, the confidence in the simulations can be deduced from the answer to subproblem 1.5 above. More specifically, as the `while`-loop simulates Y_n for each n from 1 and up, and terminates when $Y_n \leq 0.04R$ (presently disregarding the upper limit on n), the probability of a *non-good* test to be accepted as *good*, is less than 0.05 as deduced above, because we execute $R = 2996$ simulations.

To understand this, first note that each simulation of Y_n is independent of the previous one. This means the probability of each iteration of the `while`-loop resulting in a *non-good* test appearing as a *good* test, is independent of the previously executed simulations, and is therefore exactly as given in Eq.(4), when R is chosen correctly.

In fact, as mentioned above, when p_n is above 0.05, and we still keep R fixed at 2996, then the probability of a *non-good* test being accepted as a *good* test, is even smaller, and this further strengthens our confidence in this simple `while`-loop finding a good value for n. Take for example $p_n = 0.1$, then the upper Chernoff bound $F^{-}(\mu, \delta)$ in Eq.(5), is less than 4e-24 for such an erroneous acceptance of the *non-good* test.

Subproblem 2.4

Two executions are made of the above program that simulates Y_n, and finds the lowest n that yields a *good* test. The first one rounds m upwards to the nearest integer, and those results are given in table 1. The second run rounds m down instead, and those results are shown in table 2. In both cases, the maximum allowed n is set to 82 – which was the Chernoff bound for n, as discovered in subproblem 1.3.

Clearly the results are different when m is rounded up or down. The reason is of course, that when rounding m upwards, we require one more guess from the challenged student, than when rounding down, and when $n \cdot 3/5$ is not an integer. This difference is clearly seen in the two tables. Another, perhaps more striking feature, which recurs in both tables, is that the pass-ratio does not converge monotonically, as one would expect. Believing the quality of the RNG is beyond reproach, the reason for this inconsistent convergence of Y_n and hence Y_n/R, is also found in the rounding of m.

n	m	Passed (Ratio)
1	1	721 (0.241)
2	2	183 (0.061)
3	2	1287 (0.430)
4	3	448 (0.150)
5	3	458 (0.153)
6	4	789 (0.263)
7	5	320 (0.107)
8	5	294 (0.098)
9	6	503 (0.168)
10	6	522 (0.174)
11	7	194 (0.065)
12	8	336 (0.112)
13	8	355 (0.118)
14	9	135 (0.045)
15	9	545 (0.182)
16	10	230 (0.077)
17	11	103 (0.034)

Table 1: Simulation of Y_n with m rounded upwards to nearest integer by using `std::ceil()`.

Subproblem 2.5

Comparing the theoretical bound of 82 for n as found in subproblem 1.3, with the ones found by experimental simulations in subproblem 2.4, it is clear that whichever choice of rounding m, still results in a significantly lower value of n. The reason for this, is found in the use of the Chernoff bounds, which are just that, bounds. There is no guarantee on the minimality of such bounds, and although the comparison may seem a tad silly, we can liken such a bound to a bound of 1 for the function $f(x) = e^{-x}$ with $x > 1$. Using a numerical approximation of $f(x)$, we would quickly learn that the actual bound is really much lower than 1, although the numerical experiments will only be able to tell us the value up to a certain precision, and we therefore still do not know the minimum upper bound for $f(x)$. With the Chernoff bounds, the function in question is randomized, but the illustrated concept of approximation is much similar to that of simulation, in regards to bounds.

The question remains, which n should the teacher actually decide upon? Obviously, the definition of a challenged student knowing the answers to *exactly* 40% of the questions, is too rigid. Indeed, the problem description uses a more useful formulation: A challenged student knows the answer to *at most* 40% of the questions; which implies m should be rounded up, and hence we should use the result from table 1, meaning the test should have at least $n = 17$ questions, for the test to be *good*.

Subproblem 2.6 (Last 5% Credit)

If we generalize the number of choices to k for each of the n questions in the test, and the score for a correct answer then being $\log k$ (assume the logartihm with base 2: $\log = \log_2$), and the score for a wrong answer still being -1, then how many n are required for a test to be good.

The calculations are similar to those for the case $n = 4$, albeit with expressions of k instead. First off, we clearly have the following expectancy for a single guess X_i:

$$E[X_i] = \frac{1}{k} \cdot 1 + \frac{k-1}{k} \cdot 0 = \frac{1}{k}$$

and as there are still m questions for which the challenged student will make guesses, we have the following expectancy for the sum X of those guesses:

$$E[X] = \cdots = \frac{1}{k} \cdot m = \frac{3}{5k} n$$

The score obtained from these guesses, is then given by:

$$Z = (\log k + 1)X - m$$

and the total score that a challenged student obtains, is given by:

$$S = \log k \cdot \frac{2}{5} n + Z = \log k \cdot \frac{2}{5} n + (\log k + 1)X - m$$

For such a student to pass, we still need half the maximum score, which means:

$$S \geq \log k \cdot \frac{1}{2} n \;\Leftrightarrow\; X \geq n \cdot \frac{\frac{\log k}{10} + 3/5}{\log k + 1}$$

We now wish to determine (a bound for) the following probablity:

$$\Pr[X > (1+\delta)\mu]$$

where $\mu = E[X] = (3n)/(5k)$ and $(1+\delta) = \alpha$ is found as follows:

$$\alpha \cdot E[X] = n \cdot \frac{\frac{\log k}{10} + 3/5}{\log k + 1} \;\Leftrightarrow\; \alpha = \cdots = k \cdot \frac{\frac{\log k}{6} + 1}{\log k + 1}$$

Which means:

$$\delta = \cdots = k \cdot \frac{\frac{\log k}{6} + 1}{\log k + 1} - 1$$

Now consider the Chernoff bound:

$$F^{+}(\mu, \delta) = \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}} \right]^{\mu}$$

Because of the rather large expressions for δ and μ, we shall not presently write this out, but simply keep in mind that δ and μ are actually functions of k. We furthermore define:

$$\gamma = \frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}$$

which is therefore also a function of k, but allows us to write the bound as:

$$F^+(\mu, \delta) = \gamma^\mu$$

For this bound to be at most 0.05, we have the usual:

$$F^+(\mu, \delta) \leq 0.05 \;\Leftrightarrow\; \gamma^\mu \leq 0.05$$

taking the logarithm on both sides and rearranging, we get:

$$\mu \geq \frac{\ln 0.05}{\ln \gamma}$$

where $\leq$ becomes $\geq$ because $\ln \gamma < 0$. This means:

$$\frac{3}{5k} \cdot n \geq \frac{\ln 0.05}{\ln \gamma}$$

which is equivalent to:

$$n \geq \frac{5k}{3} \cdot \frac{\ln 0.05}{\ln \gamma}$$

assuming $k > 0$.

Simulation

The previous implementation of the simulation was done for the special case $k = 4$. That source-code is now found in `ra_exam_problem2_org.cpp`, and `ra_exam_problem2.cpp` has been extended as follows. First off, the user now provides the number of choices k, and the following function computes the Chernoff bound from that:

```
double Chernoff()
{
    double deltaPlusOne = gK * (gScore/6 + 1) / (gScore + 1);
    double delta = deltaPlusOne - 1;
    double gamma = std::exp(delta) / std::pow(deltaPlusOne,
                                              deltaPlusOne);

    return (5 * gK / 3.0) * std::log(0.05) / std::log(gamma);
}
```

where `gScore` is $\log_2 k$, and `gK` is k as provided by the user. Since we now compute the Chernoff bound, there is no need for the user to input the maximum allowed value of n. The function for simulating X uses the general probablity $1/k$ instead of the fixed $1/4$:

```
int SimulateX(int m)
{
    int X = 0;
    for (int i=0; i<m; i++)
    {
        if (gRandom.RandUni() < gKReciprocal)
            X++;
    }

    return X;
}
```

where `gKReciprocal` holds the value $1/k$. Since $\log_2 k$ is not guaranteed to be an integer, the function for computing the score, now returns a `double` instead of an integer:

```
double GetScore(int n, int m, int X)
{
    // A correct answer scores gScore points.
    // A wrong answer scores -1 point.
    return gScore*(n-m) + (gScore+1)*X - m;
}
```

and the simulation of Y_n also takes this into account:

```
int SimulateY(int n, int m)
{
    int Yn = 0;
    double passScore = gScore * 0.5 * n;

    for (int i=0; i<kR; i++)
    {
        int X = SimulateX(m);
        double score = GetScore(n, m, X);

        if (score >= passScore)
            Yn++;
    }

    return Yn;
}
```

We ignore any floating point rounding errors that may cause `score >= passScore` to fail, even though it would be mathematically expected to be true. This does not seem to affect our experiments. Also note that the computation of the number of simulations R in subproblem 1.5, is independent of both k and the score, and we may there continue to use $R = 2996$.

The results of running the simulation with k from 2 up to 7, are shown in table 3, where the Chernoff bounds for n are also presented. As can be seen, the Chernoff bounds for n are consistently much higher than the experimental values found by simulation. Note that for $k = 8$, the probability of guessing the correct answer to a single question, is $1/8 = 0.125$. This means the test only needs two questions, where the student will of course need to guess both (if he knew the answer to one of them, he would have 50% right, and pass the test by definition, thus ceasing to be a challenged student). The probability for a student to guess both questions is $1/8^2 < 0.05$, which means a test with $k \geq 8$ is *good* simply by the number of choices k a question has, and not how many questions n the quiz has. For $k = 20$ and upwards, the test only needs a single question.

Problem 3

Given two multisets X and Y, both of size n and containing the words x_i and y_i respectively, we are to study two fingerprint-based methods for deciding whether X and Y are equal. The methods are allowed to fail (i.e. classify the multisets as being equal, when in fact they are not), with probability $\leq 2^{-k}$ for some $k \in \mathbb{N}$.

Subproblem 3.1

The first method considers the polynomials f and g defined in factorial form by:

$$f(z) = (z - x_1) \cdots (z - x_n)$$

and:

$$g(z) = (z - y_1) \cdots (z - y_n)$$

where z is a word from the field of words $\mathbb{F}$. Without loss of generality, we shall assume that words are simply bitstrings of length $\leq C$. The arithmetic operations on bitstrings are welldefined, they may be interpreted as integers instead. Clearly what we wish to decide, is whether the polynomium:

$$Q(z) = f(z) - g(z)$$

is identically zero:

$$Q(z) \equiv 0$$

regardless of $z \in \mathbb{F}$. Which would mean that the following statements are equivalent:

- $Q(z) \equiv 0$.
- $f(z) = g(z)$ for all $z \in \mathbb{F}$.
- $X = Y$.

If $X = Y$ then f equals g for all z, because the x_i's and y_i's are merely permutations of each other. The converse is also trivially true, so bi-implication between the last two statements has been established. Bi-implication between the first two statements is also trivial, and follows directly from the definition of Q. We are of course primarily concerned with the equivalence between $Q(z) \equiv 0$ and $X = Y$.

Subproblem 3.2

Randomly deciding whether f and g are equal, can be done by selecting a word r randomly from $\mathbb{F}$, and then computing $Q(r)$. Clearly if $Q(r) \neq 0$, then we may immediately deduce that f and g are different, and hence $X \neq Y$. On the other hand, if $Q(r) = 0$, then we are interested in deciding, the probability that $Q(z) = 0$ for all $z \in \mathbb{F}$, or equivalently, the probability that $Q(z) \not\equiv 0$ even though we had a sample $r \in \mathbb{F}$ for which $Q(r) = 0$.

To this end, we use the socalled *Schwartz-Zippel* theorem from [1, p. 165], on the *univariate* polynomium $Q(z)$. Both f and g are of degree n, so Q is at most of degree n. The theorem then states:

$$\Pr[Q(r) = 0 \mid Q(z) \not\equiv 0] \leq \frac{n}{|\mathbb{S}|}$$

where $\mathbb{S}$ is a finite subset of $\mathbb{F}$. Since $\mathbb{F}$ is itself finite, let us simply use that, noting it has size $|\mathbb{F}| = 2^C$. We therefore obtain:

$$\Pr[Q(r) = 0 \mid Q(z) \not\equiv 0] \leq \frac{n}{|\mathbb{S}|} = \frac{n}{2^c}$$

Performing m iterations of the algorithm, is done by picking m words r_i independently, uniformly and randomly from $\mathbb{F}$, and computing $Q(r_i)$ until it is nonzero for some r_i, or otherwise if $Q(r_i) = 0$ for all r_i, then conclude the probability that we have observed all these $Q(r_i) = 0$ even though $Q(z) \not\equiv 0$, is:

$$\Pr[Q(r_i) = 0 \text{ for all } r_i \mid Q(z) \not\equiv 0] \leq \left(\frac{n}{2^c}\right)^m$$

Which is due to the independence of the individual runs of the method. The question is then, how big should m be, in order for this probability to be $\leq 2^{-k}$. We have:

$$\left(\frac{n}{2^c}\right)^m \leq 2^{-k} \;\Leftrightarrow\; m \geq \frac{\log_2(2^{-k})}{\log_2\left(\frac{n}{2^C}\right)}$$

since $n < 2^C$. And this can be reduced to:

$$m \geq \frac{-k}{\log_2(n) - C}$$

So for example, if we were considering multisets of size $n = 2^{14} = 16384$, containing words of at most $C = 32$ bits, and we would like a probability of error $\leq 2^{-8} \simeq 0.0039$, then we require:

$$m \geq \frac{-8}{\log_2(16384) - 32} = \frac{8}{18}$$

So we only need a single iteration of the algorithm in this case.

Concerning the worst-case running time for this algorithm, we shall once more consider bitstrings of max length C. The problem-description suggests that the characters in a word have 8 bits each, but the deduction is slightly simpler if we continue to consider bitstrings of max length C, instead of $8 \cdot C$ or $16 \cdot C$.

Let us start by assuming a random $r \in \mathbb{F} = \{0,1\}^C$ can be picked in time $O(C)$, and also ignore the possibility of overflows in the following. Then we need to compute $f(r)$ and $g(r)$, each of which uses n subtractions and $n-1$ multiplications. As each of these arithmetic operation are assumed to take time $O(b^2)$, when performed on bitstrings of length b, the computation of $f(r)$ and $g(r)$ can be done in time $O(n \cdot C^2)$. Then a single subtraction of the values for $f(r)$ and $g(r)$ is needed to compute the value of $Q(r)$, but as this is just a single operation, and does not add to the time complexity. This procedure is to be repeated $\lceil m \rceil$ times in order to achieve the desired probability of failure. The total time complexity of the algorithm is therefore:

$$O(n \cdot C^2 \cdot \lceil m \rceil) \tag{6}$$

As C is a fixed bound on the length of a word (here, bitstring), we may excluded that from our time-complexity as a constant factor. Furthermore, for most reasonably sized multisets X and Y, we would get a very low value of m, most likely $m = 1$. Therefore we may say that this randomized algorithm approaches linear execution time.

Note that the maximum length C for a word $z \in \mathbb{F}$, can be determined from the multisets at hand. That is, in time $O(n)$ we can run through X first and then Y, finding the word of greatest length, and then set C to this. The random word r is then generated so as to have at most, the number of characters for the longest word in either X or Y. We could also choose a smaller C, as long as $n < \mathbb{S} = 2^C$. This means we could generate less random characters (here, bits), but with the proper choice of C, we could still guarantee the probability bound of 2^{-k}.

Subproblem 3.3

In the second fingerprint-based approach, we seek a congruence map of a multiset X to a single (small) number, which may be used for comparison instead of X itself. To achieve this, start by defining the polynomial sum over a multiset X as follows:

$$P_X(m) = \sum_{i=1}^{n} m^{x_i}$$

where $m = n + 1$. Clearly if $X = Y$, then their polynomial sums are also equal:

$$X = Y \ \Rightarrow\ P_X(m) = P_Y(m)$$

So far however, we have not achieved any reduction in the amount of data to be compared, which is where modulo arithmetics come into play. Because of reflexivity of congruence, we have:

$$P_X(m) \equiv P_Y(m) \mod p$$

If we define:

$$F_p(X) = P_X(m) \mod p$$

and similarly for Y, then we have shown:

$$X = Y \;\Rightarrow\; F_p(X) = F_p(Y)$$

Naturally, the inverse does not hold in general, which is why $F_p(X)$ is called a *fingerprint*. Two issues are of interest now, first is the selection of p, second is the efficient calculation of $F_p(X)$. These issues will be resolved in the next section.

Subproblem 3.4

As in subproblem 3.2, we now consider the probability that our modulo-based multiset equality method fails, that is, the probability that the method yields $F_p(X) = F_p(Y)$ even though $X \neq Y$. The probability can be written as:

$$\Pr[F_p(X) = F_p(Y) \mid X \neq Y]$$

Randomization is introduced by selecting p at random. The question is then, how big should p be so that this probability is small?

Once more, and still without loss of generality, we assume the words in X and Y are bitstrings, and then let C be the length of the longest word in either of X and Y. This means all x_i and y_i can be interpreted as numbers $\leq 2^C$. The maximum value for $P_X(m)$ and $P_Y(m)$, is then:

$$P_X(m),\, P_Y(m) \;\leq\; n \cdot m^{2^C} \tag{7}$$

If we let d be the difference between $P_X(m)$ and $P_Y(m)$, then $|d|$ must clearly be similarly bounded:

$$d = P_X(m) - P_Y(m),\, |d| \leq n \cdot m^{2^C}$$

From the definition of congruence, $d \mod p = 0$ means that p divides d, which is equivalent to:

$$d \mod p = 0 \;\Leftrightarrow\; P_X(m) \mod p - P_Y(m) \mod p = 0 \;\Leftrightarrow\; F_p(X) = F_p(Y)$$

So assuming $X \neq Y$ so that $P_X(m) \neq P_Y(m)$ and hence $d \neq 0$, the only way for $F_p(X)$ to equal $F_p(Y)$, is if p divides d, which is equivalent to p dividing $|d|$. How many different choices of p can divide $|d|$?

Since the smallest prime is 2, then $|d|$ can at most have $\log_2(|d|)$ prime divisors. Because of the above bound on $|d|$, then $|d|$ has at the most, the following number of prime divisors:

$$\log_2(|d|) = \log_2\left(n \cdot m^{2^C}\right)$$

The idea is then to select a random prime p_r less than some τ, where the total number of primes less than τ is given by $\pi(\tau)$. The actual value of τ, is found below. Now having the total number of primes to select the random p_r from, and the maximum number of primes which may cause the fingerprint to misjudge the multiset equality, the probability of failure is therefore given by:

$$\Pr[F_p(X) = F_p(Y) \mid X \neq Y] \leq \frac{\log_2\left(n \cdot m^{2^C}\right)}{\pi(\tau)}$$

Using the hint from the problem description, we know $\pi(\tau) > \tau/\ln\tau$ for $\tau \geq 17$, so we have:

$$\Pr[F_p(X) = F_p(Y) \mid X \neq Y] < \frac{\ln\tau}{\tau} \cdot \log_2\left(n \cdot m^{2^C}\right)$$

when assuming $\tau \geq 17$. If we let:

$$\tau = tm \cdot \log_2\left(n \cdot m^{2^C}\right)$$

for some adequately large t (e.g. $t = m$), we have:

$$\frac{\ln\tau}{\tau} \cdot \log_2\left(n \cdot m^{2^C}\right) = \frac{\ln\left(tm \cdot \log_2\left(n \cdot m^{2^C}\right)\right)}{tm}$$

so we have:

$$\Pr[F_p(X) = F_p(Y) \mid X \neq Y] < \frac{\ln\left(tm \cdot \log_2\left(n \cdot m^{2^C}\right)\right)}{tm}$$

Then select a sequence p_i of l independent random primes less than τ, and continue to compute and compare $F_{p_i}(X)$ and $F_{p_i}(Y)$ until they are either different, or until all p_i's have been used. In the latter case, the probability that $X \neq Y$ although $F_{p_i}(X) = F_{p_i}(Y)$ for all p_i, is given by:

$$\Pr[F_{p_i}(X) = F_{p_i}(Y) \text{ for all } p_i \mid X \neq Y] < \left(\frac{\ln\left(tm \cdot \log_2\left(n \cdot m^{2^C}\right)\right)}{tm}\right)^l$$

So if we want the desired probability bound of 2^{-k}, we would require:

$$\left(\frac{\ln\left(tm \cdot \log_2\left(n \cdot m^{2^C}\right)\right)}{tm}\right)^l \leq 2^{-k} \;\Leftrightarrow\; l \geq \frac{-k}{\log_2\left(\frac{\ln\left(tm \cdot \log_2\left(n \cdot m^{2^C}\right)\right)}{tm}\right)}$$

If for example we take $C = 32$, $n = 2^{14} = 16384$, $t = m = n + 1$, and $k = 8$, we get:

$$l \geq 0.35$$

so in this case, a single iteration of the method is needed to provide the given bound on probability of failure. We need to choose the random primes p_i less than:

$$\tau \simeq 1.614 \times 10^{19}$$

which is of course rather large, but still, $P_X(m)$ and $P_Y(m)$ are vastly larger, as we know from Eq.(7):

$$P_X(m),\, P_Y(m) \ \leq \ n \cdot m^{2^C} = 16384 \cdot 16385^{4294967296}$$

for these choices of C, n, t, and m.

Algorithm Overview

First find C by running through X and Y, letting C be the length of the longest bitstring in either of the two multisets. Then let $t = m = n + 1$, and compute τ and l and draw $\lceil l \rceil$ random primes $p_i < \tau$. Then compute $F_{p_i}(X)$ and $F_{p_i}(Y)$, and if they are different, halt the algorithm and report the multisets are not equal, otherwise continue until all the random primes p_i have been used.

Time Complexity

Concerning the time complexity, first we need to decide how many bits are needed for our random primes. A prime is less than τ, so we need at most $\log_2 \tau$ bits:

$$\log_2 \tau = \log_2 \left(tm \cdot \log_2 \left(n \cdot m^{2^C} \right) \right) = \log_2(tm) + \log_2 \left(\log_2 \left(n \cdot m^{2^C} \right) \right)$$

which is approximately:

$$\log_2 \tau \simeq \log_2(tm) + C$$

which we denote by b:

$$b = \log_2(tm) + C$$

Finding C can be done in time $O(n)$ by going through the multisets X and Y. Computing τ and l both take $O(1)$ time. A random b-bit prime is assumed to be provided at a time-cost of $O(b^4)$, and we need n of these. Using socalled *repeated squaring* to compute each m^{x_i} takes time:

$$O(\log_2(2^C) \cdot b^2) = O(C \cdot b^2)$$

under the assumption that a single multiplication of b-bit numbers takes time $O(b^2)$. To see this, note that repeated squaring has $\log_2(2^C) = C$ iterations, because $x_i \leq 2^C$, and each iteration requires multiplication and modulo, both performed on b-bit numbers, and therefore assumed to take $O(b^2)$ time each. This procedure must be repeated n times; once for each x_i, and then we need $n-1$ additions to compute $F_{p_i}(X)$. Note that during the summation in $F_{p_i}(X)$, we also compute modulo p on the partial results, to keep the numbers small (i.e.

b-bit). This means $F_{p_i}(X)$ is computed in time $O(n \cdot (b^2 + b^4 + C \cdot b^2))$ which equals:

$$O(n \cdot b^4) = O(n \cdot (\log_2(tm) + C)^4) \simeq O(n \cdot C^4)$$

for sufficiently large C (so that C dominates over $\log(tm)$). Naturally we wish to perform $\lceil l \rceil$ iterations, so our total time complexity becomes:

$$O(n \cdot C^4 \cdot \lceil l \rceil) \tag{8}$$

and as was the case with the previous polynomial-based method for comparing multisets X and Y, we usually have that the number of required iterations is very low, e.g. $\lceil l \rceil = 1$, and since C may be fixed, we can say that this congruence-based and randomized fingerprint method, executes in time linear to the size of X and Y. However, the effect of C will be discussed in subproblem 3.5 below.

Deterministic Algorithm

A simple deterministic approach for comparing the multisets X and Y, would be to consider their elements x_i and y_i as sequences, and sort them. Deterministic sorting can be done in time $O(n \log n)$, and the ensuing comparison of the sorted sequences, in time $O(n)$. Thus, this deterministic algorithm would solve the multiset equality problem, in time complexity $O(n \log n)$. Note that depending on the sorting algorithm (whether it is inplace or not, and also depending on how the datasets for X and Y are stored), we may require additional memory proportional to n, in order for this algorithm to work. Such additional memory is not required in the randomized methods.

Comparison (Subproblem 3.5)

A number of assumptions were made in order to achieve the simple forms of Eqs.(6) and (8). For the polynomial-based fingerprint method, we assumed that the arithmetic operations would not overflow. This assumption is also implicit for the congruence-based method, although its use of modulo arithmetics makes this less important. In reality, the polynomial-based method will require more than C bits (depending on n), and hence increase the cost of the arithmetic operations, as they are considered functions of C.

For the congruence-based method, the random selection of the primes p_i having b bits, was assumed to have time complexity $O(b^4)$, which is a major contribution to the approximate time complexity of $O(n \cdot C^4 \cdot \lceil l \rceil)$, which has a C^4 factor instead of the C^2 factor for the polynomial-based method.

Although we should not expect to see many documents (i.e. real-world multisets) for which $\log_2(tm)$ dominates C, i.e. $tm > 2^C$. But if this happens, then the time complexity would be more like $O(n \cdot \log(tm)) = O(n \cdot \log(n))$. Note that this is similar to the deterimnistic algorithm, but is for very small C.

The question remains, which algorithm should be chosen for an implementation? In both cases we will need an auxiliary library for carrying out integer arithmetics on a high number of bits. The time complexity of the polynomial-based is likely higher (perhaps $O(n \cdot C^3 \cdot \lceil m \rceil)$) because of the chance of significant overflows. Another advantage of the congruence-based method, is that it only requires positive numbers and partial results, which means we can save the sign-bit. Since the exact number of operations on the exact number of bits have not been calculated though, and the asymptotical time complexities are only approximates, we will need real-world experimentation to decide which method is really the fastest.

Problem 4

The issues that are of interest to testing, are primarily the correctness of algorithms, probability-bounds, and running time. So in testing the fingerprint-based methods for solving the Multiset Equality problem, we first need to select a pool of multisets. The problem was originally posed as one of comparing documents, and we would therefore require an adequate amount of text documents to perform such testing. Furthermore, we need an auxiliary library that can perform arithmetic operations on large numbers (e.g. 80-bit integers for 10 character words, each character being 8-bit; and these may overflow significantly, requiring even more bits). And for the congruence-based method, we also require an efficient prime-generator.

In constructing the tests for running time, we are both interested in running time as a function of the number of words n in the multisets, but also as a function of the maximum stringlength C. So, the documents used for test-multisets, should be selected with this in mind.

It is beyond the scope of this document to construct such a rigorous testing, in particular because of the lack of a variable-bit arithmetic library and the prime-generator. Instead, we shall implement a simplified version which uses 32-bit integers as the basic datatype. This severely restricts the size of the multisets n and the wordlengths C, if we are to avoid overflowing in the intermediate arithmetic results – in particular for the polynomial-based fingerprints. So for this reason, we are unable to really test any of the desired issues, and we shall therefore suffice with a description of the implementation, thus merely demonstrating the concepts, and then provide a few testruns to indicate correctness of the implementation.

The user starts by entering the number of words in the multisets, and how many of these should be identical in X and Y. The integer value of the words range between 0 and $2^C - 1$, and the words are chosen at random. The words that are not identical in X and Y, are simply set to:

$$y_i = 2^C - x_i - 1$$

The basic function for computing the polynomial fingerprint of a multiset `X` is given by:

```
T MultisetPolynomial(T *X, T z, int n)
```

```
{
    T prod = 1;

    for (int i=0; i<n; i++)
    {
        prod *= (z - X[i]);
    }

    return prod;
}
```

where the datatype `T` may easily be defined to be e.g. a higher-precision bit-string, provided it implements the appropriate arithmetic operations. For the congruence-based fingerprint, We use the following prime-generating polynomial from [3], however noting that it only produces 45 primes, ranging from 2753 up to 36809:

$$p_i = 36i^2 - 810i + 2753$$

for $i \in \{0, \cdots, 44\}$. This prime-generator is implemented as follows:

```
int GetPrime()
{
    int prime = 36 * gPrimeCount*gPrimeCount - 810 * gPrimeCount
                + 2753;

    gPrimeCount++;
    gPrimeCount %= kPrimeCount;

    return prime;
}
```

where `gPrimeCount` is a global variable ensuring that we only compute the primes that are valid for this particular prime-generating polynomial. The congruence-based fingerprint for a multiset `X`, is computed as follows:

```
T MultisetCongruence(T *X, int n, int p)
{
    int m = n+1;
    T sum = 0;

    for (int i=0; i<n; i++)
    {
        sum += RepeatedSquaring(m % p, X[i], p);
        sum %= p;
    }

    return sum;
}
```

Where the auxiliary function for computing $m^{x_i} \mod p$ is implemented recursively:

```
T RepeatedSquaring(const int x, const T k, const T p)
{
    T retVal;

    if (k == 0)
    {
        retVal = 1;
    }
    else if (k % 2 == 1)
    {
        retVal = (x * RepeatedSquaring(x, k-1, p)) % p;
    }
    else
    {
        T t = RepeatedSquaring(x, k>>1, p) % p;
        retVal = (t * t) % p;
    }

    return retVal;
}
```

Comparing the fingerprints for the polynomial-based method is done as follows:

```
bool MultisetEqualityPolynomial()
{
    T z = gRandom.RandIndex(gMaxWordValue);
    T f = MultisetPolynomial(gX, z, gN);
    T g = MultisetPolynomial(gY, z, gN);

    return f == g;
}
```

where `z` is chosen in the range 0 to `gMaxWordValue` minus one. And in a similar fashion for the congruence-based fingerprints, we have:

```
bool MultisetEqualityCongruence()
{
    T p = GetPrime();
    T FpX = MultisetCongruence(gX, gN, p);
    T FpY = MultisetCongruence(gY, gN, p);

    return  FpX == FpY;
}
```

which simply cycles through the 45 available primes, drawing a new on each call of the function. These are wrapped in a function which performs the kind of fingerprint comparison of the user's choice:

```
bool MultisetEquality()
{
    bool retVal;

    switch (gFingerprintType)
    {
    case ePolynomial:   retVal = MultisetEqualityPolynomial();
                        break;
    case eCongruence:
    default:            retVal = MultisetEqualityCongruence();
    }

    return retVal;
}
```

which could have been implemented as sub-classing in a more general setting. Finally, this is iterated a number of times, while counting the number of times the fingerprints were equal:

```
    int numTrue = 0;

    for (int i=0; i<gNumIterations; i++)
    {
        if (MultisetEquality())
            numTrue++;
    }
```

after which the result is printed to standard output, for the user to read.

Experiments

As mentioned above, the polynomial fingerprint quickly overflows its intermediate calculations. Take for example $n = 32$ and the words having 8 bits each, which means x_i and y_i range between 0 and 255. Then $(z - x_i)$ also requires 8 bits (and an additional sign-bit), and in the worst case, multiplying these for all x_i's requires significantly more than 32 bits, which is all we have at our disposal here.

However, if we try running the program with $n = 32$ anyway, and where $x_i \neq y_i$ for all i, and we allow the x_i's and y_i's to range between 0 and 2047 (both inclusive), then 1000 iterations of the polynomial fingerprint, each with a new random $z \in \{0, \cdots, 2047\}$, results in 253 iterations having equal fingerprints. Although our method of generating the multisets does not guarantee that $X \neq Y$ even though $x_i \neq y_i$ for all i, we should still not place much confidence in this result, simply because of the rapid overflowing. In an identical run, but with each $x_i = y_i$, we expect all fingerprints to be equal, which is also the case. If we let 31 of the 32 words in X and Y be identical, and the last word generated as $y_i = 2^C - x_i - 1$, then we find 409 out of 1000 fingerprints were equal. Again, due to overflow, this does not reveal much in regards to correctness or accuracy of the method or this implementation, but we may boldly use it as an indication, that the more elements in the two multisets that are identical, the higher is the likelihood that the fingerprints will be identical too.

Repeating these experiments with the congruence-based fingerprint, we find that when $x_i \neq y_i$, then one of the 45[1] iterations have identical fingerprints. If instead we let $x_i = y_i$, then all 45 iterations yield identical fingerprints, as was expected. When only 31 out of the 32 elements in the multisets are equal, and the last is generated as described above, then 0 out of 45 iterations yield identical fingerprints. Since our multisets are rather artificial, in particular in regards to the generation of inequal elements $x_i \neq y_i$, and furthermore that we use such a simple and limited prime-generator, then we can not place much confidence in this algorithm either, just from these few results. However, because of the use of *repeated squaring* and modulo arithmetics in the summation also, we are much less likely to run into overflow-problems, in particular when the primes p are as small as here. Therefore, the present congruence-based implementation can be said to be of some use, even when it has only 32-bit numbers. For example, if we extend the multisets to have 2048 elements each, and let 2047 of these be identical, and the last generated as above. And if we then let the words take on values between 0 and $2^{14} - 1 = 16383$ (both inclusive), then in two runs none of the iterations of the congruence-based method resulted in equal fingerprints, while another run did result in a single fingerprint match. This tendency for either zero or just a few fingerprint matches, corresponds well with intuition of the congruence-based method.

Even though we can not conclude much from these experiments, it is evident that the congruence-based fingerprints have a clear advantage in regards to overflowing its arithmetics. When deciding upon an algorithm, this advantage should be weighted against the difficulty of selecting random primes, as well as the increased time expenditure of the congruence-based method over the polynomial-based. However, without a library for performing arithmetics on arbitrarily long bitstrings, we are forced to use the congruence-based method, for all but the most trivial cases of multiset equality.

References

[1] Rajeev Motwani and Prabhakar Raghavan: Randomized Algorithms, Cambridge University Press, ISBN 0-521-47465-5

[2] William H. Press et. al. Numerical Recipes in C, Cambridge University Press 1992, ISBN 0-521-43108-5

[3] Sloane, N. J. A., Sequence A050268 in "An On-Line Version of the Encyclopedia of Integer Sequences.", `http://www.research.att.com/~njas/sequences/eisonline.html`.

[1]Recall that we only have 45 primes at our disposal.

n	m	Passed (Ratio)
1	0	2996 (1.000)
2	1	728 (0.243)
3	1	2996 (1.000)
4	2	1298 (0.433)
5	3	486 (0.162)
6	3	1718 (0.573)
7	4	775 (0.259)
8	4	784 (0.262)
9	5	1061 (0.354)
10	6	547 (0.183)
11	6	532 (0.178)
12	7	728 (0.243)
13	7	704 (0.235)
14	8	341 (0.114)
15	9	477 (0.159)
16	9	500 (0.167)
17	10	222 (0.074)
18	10	623 (0.208)
19	11	354 (0.118)
20	12	168 (0.056)
21	12	472 (0.158)
22	13	256 (0.085)
23	13	237 (0.079)
24	14	341 (0.114)
25	15	174 (0.058)
26	15	164 (0.055)
27	16	270 (0.090)
28	16	240 (0.080)
29	17	118 (0.039)

Table 2: Simulation of Y_n with m rounded downwards to nearest integer by using `std::floor()`.

k	Chernoff n	Simulation n
2	758	162
3	158	29
4	82	17
5	56	11
6	43	2
7	36	2

Table 3: Simulation of Y_n, with different number of choices k for the questions in the test.