EECE 5643: Simulation and Performance Evaluation
Professor Ningfang Mi

Homework 4

- Assignment Due: 03/02/2023 -

Harrison Sun
Monday, Thursday 11:45 am - 1:25 pm
Completed: March 2, 2023

1 Ex. 4.1.11

Calculate $\bar{x}$ and s by hand, using the two-pass algorithm, the one-pass algorithm, and Welford's algorithm in the following two cases.

(a) Data based on $n = 3$ observations: $x_1 = 1, x_2 = 6, x_3 = 2$.

One Pass Mean:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$
$$\bar{x}_1 = \frac{1}{1}\sum_{i=1}^{1} x_i = 1$$
$$\bar{x}_2 = \frac{1}{2}\sum_{i=1}^{2} x_i = 3.5$$
$$\bar{x}_3 = \frac{1}{3}\sum_{i=1}^{3} x_i = 3$$

Welford's Algorithm Mean:

$$\bar{x}_i = \bar{x}_{i-1} + \frac{1}{i}(x_i - \bar{x}_{i-1})$$
$$\bar{x}_1 = 0 + \frac{1}{1}(1 - 0) = 1$$
$$\bar{x}_2 = 1 + \frac{1}{2}(6 - 1) = 3.5$$
$$\bar{x}_3 = 3.5 + \frac{1}{3}(2 - 3.5) = 3$$

Two-Pass Standard Deviation:

Pass 1: Calculate Mean (From Above)

$$\bar{x}_1 = 1$$
$$\bar{x}_2 = 3.5$$
$$\bar{x}_3 = 3$$

Pass 2: Compute the Squared Differences About $\bar{x}$

$$s = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$
$$s_1 = \sqrt{\frac{1}{1}\sum_{i=1}^{1}(x_i - \bar{x}_1)^2} = 0$$
$$s_2 = \sqrt{\frac{1}{2}\sum_{i=1}^{2}(x_i - \bar{x}_2)^2} = 2.5$$
$$s_3 = \sqrt{\frac{1}{3}\sum_{i=1}^{3}(x_i - \bar{x}_3)^2} = 2.16$$

One-Pass Standard Deviation:

$$s^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i^2) - \bar{x}^2$$
$$s = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i^2) - \bar{x}^2}$$
$$s_1 = \sqrt{\frac{1}{1}\sum_{i=1}^{1}(x_i^2) - \bar{x}_1^2} = 0$$
$$s_2 = \sqrt{\frac{1}{2}\sum_{i=1}^{2}(x_i^2) - \bar{x}_2^2} = 2.5$$
$$s_3 = \sqrt{\frac{1}{3}\sum_{i=1}^{3}(x_i^2) - \bar{x}_1^2} = 2.16$$

Welford's Algorithm Standard Deviation:

$$
\begin{aligned}
v_i &= v_{i-1} + (\frac{i-1}{i})(x_i - \bar{x}_{i-1})^2 \\
v_1 &= v_0 + (\frac{0}{1})(x_1 - \bar{x}_0)^2 = 0 \\
v_2 &= v_1 + (\frac{1}{2})(x_2 - \bar{x}_1)^2 = 12.5 \\
v_3 &= v_2 + (\frac{2}{3})(x_3 - \bar{x}_2)^2 = 14 \\
s_i^2 &= \sqrt{\frac{v_{i-1} + (\frac{i-1}{i})(x_i - \bar{x}_{i-1})^2}{i}} \\
s_1^2 &= \sqrt{\frac{0}{1}} = 0 \\
s_2^2 &= \sqrt{\frac{12.5}{2}} = 2.5 \\
s_3^2 &= \sqrt{\frac{14}{3}} = 2.16
\end{aligned}
$$

(b) The sample path $x(t) = 3$ for $0 < t \leq 2$, and $x(t) = 8$ for $2 < t \leq 5$, over the time interval $0 < t < 5$.

One Pass Mean:

$$
\begin{aligned}
\bar{x} &= \frac{1}{\tau}\int_0^{\tau} x(t)dt \\
\bar{x}_1 &= \frac{1}{2}\int_0^{2} x(t)dt = 3 \\
\bar{x}_2 &= \frac{1}{5}\int_0^{2} x(t)dt + \frac{1}{5}\int_2^{5} x(t)dt = 6
\end{aligned}
$$

Welford's Algorithm Mean:

$$
\begin{aligned}
\bar{x}_i &= \frac{1}{t_i}(x_1\delta_1 + x_2\delta_2 + \ldots + x_i\delta_i) \\
\bar{x}_i &= \bar{x}_{i-1} + \frac{\delta_i}{t_i}(x_i - \bar{x}_{i-1}) \\
\bar{x}_1 &= 0 + \frac{2}{2}(3 - 0) = 3 \\
\bar{x}_2 &= 3 + \frac{3}{5}(8 - 3) = 6
\end{aligned}
$$

Two Pass Standard Deviation:

Pass 1: Calculate Path Mean (From Above)

$$\bar{x}_1 = 3$$
$$\bar{x}_2 = 6$$

Pass 2: Compute the Squared Differences About $\bar{x}$

$$s_i = \sqrt{\frac{1}{\tau}\int_0^{\tau}(x(t) - \bar{x})^2 dt}$$
$$s_1 = \sqrt{\frac{1}{2}\int_0^{2}(x(t) - 3)^2 dt} = 0$$
$$s_2 = \sqrt{\frac{1}{5}\int_0^{2}(x(t) - 6)^2 dt + \frac{1}{5}\int_2^{5}(x(t) - 6)^2 dt} = 2.45$$

One Pass Standard Deviation:

$$s_i = \sqrt{(\frac{1}{t_n}\sum_{i=1}^{n} x_i^2 \delta_i) - \bar{x}^2}$$
$$s_1 = \sqrt{(\frac{1}{2}\sum_{i=1}^{1} 3^2 \times 2) - 3^2} = 0$$
$$s_2 = \sqrt{\frac{1}{5}(3^2 \times 2 + 8^2 \times 3) - 6^2} = 2.45$$

Welford's Algorithm Standard Deviation:

$$v_i = v_{i-1} + \frac{\delta_i t_{i-1}}{t_i}(x_i - \bar{x}_{i-1})^2$$
$$v_1 = 0 + \frac{2 \times 0}{2}(3 - 0)^2 = 0$$
$$v_2 = 0 + \frac{3 \times 2}{5}(8 - 3)^2 = 30$$
$$s_i = \sqrt{\frac{v_i}{t_i}}$$
$$s_1 = \sqrt{\frac{0}{2}} = 0$$
$$s_1 = \sqrt{\frac{30}{5}} = 2.45$$

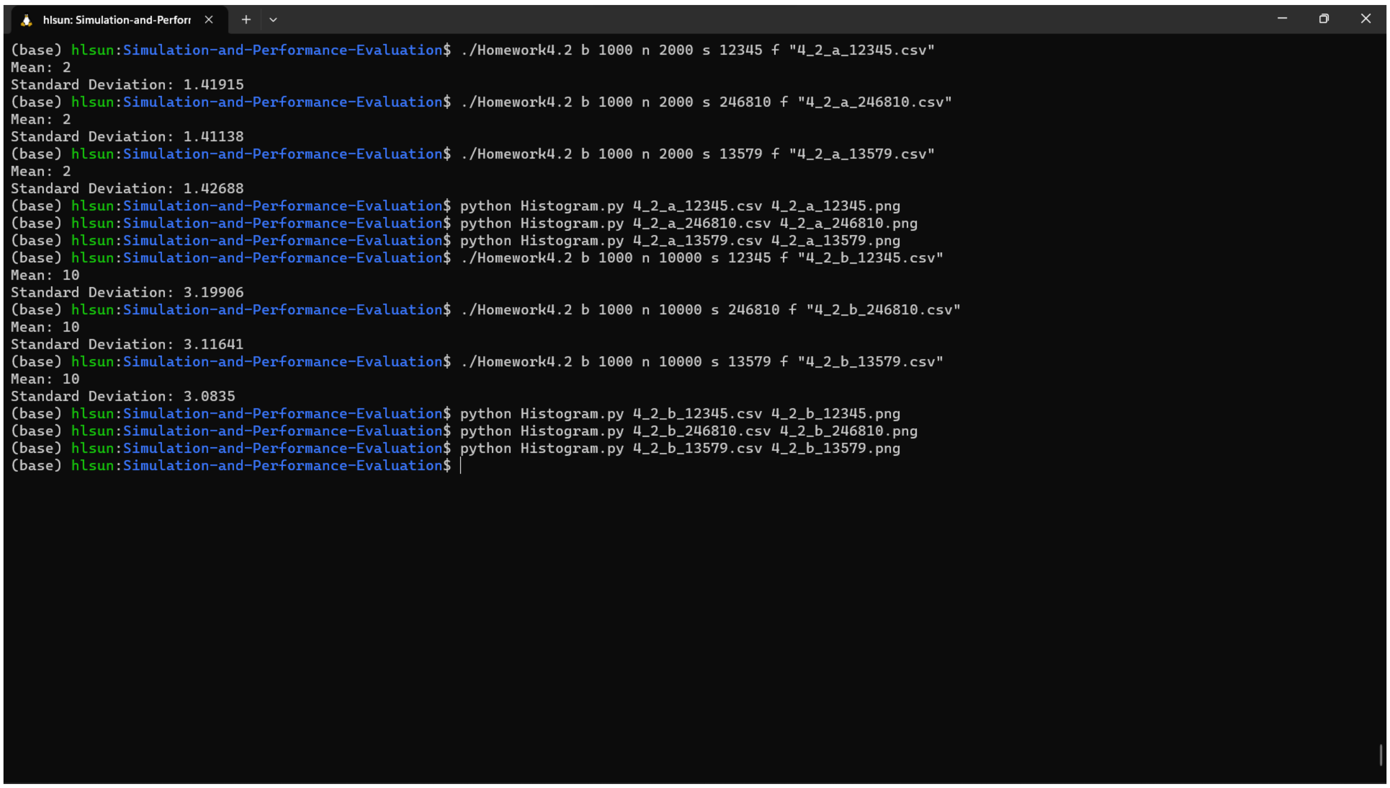
2 Ex. 4.2.2

(a) Generate the 2000-ball histogram in Example 4.2.2.
(b) Generate the corresponding histogram if 10,000 balls are placed, at random, in 1000 boxes.
(c) Calculate the histogram mean (x) and the histogram standard deviation (s) for both the 2000 balls and the 10,000 balls.

2000-ball
x = 2, s = 1.42

10,000-ball
x = 18, s = 3.1

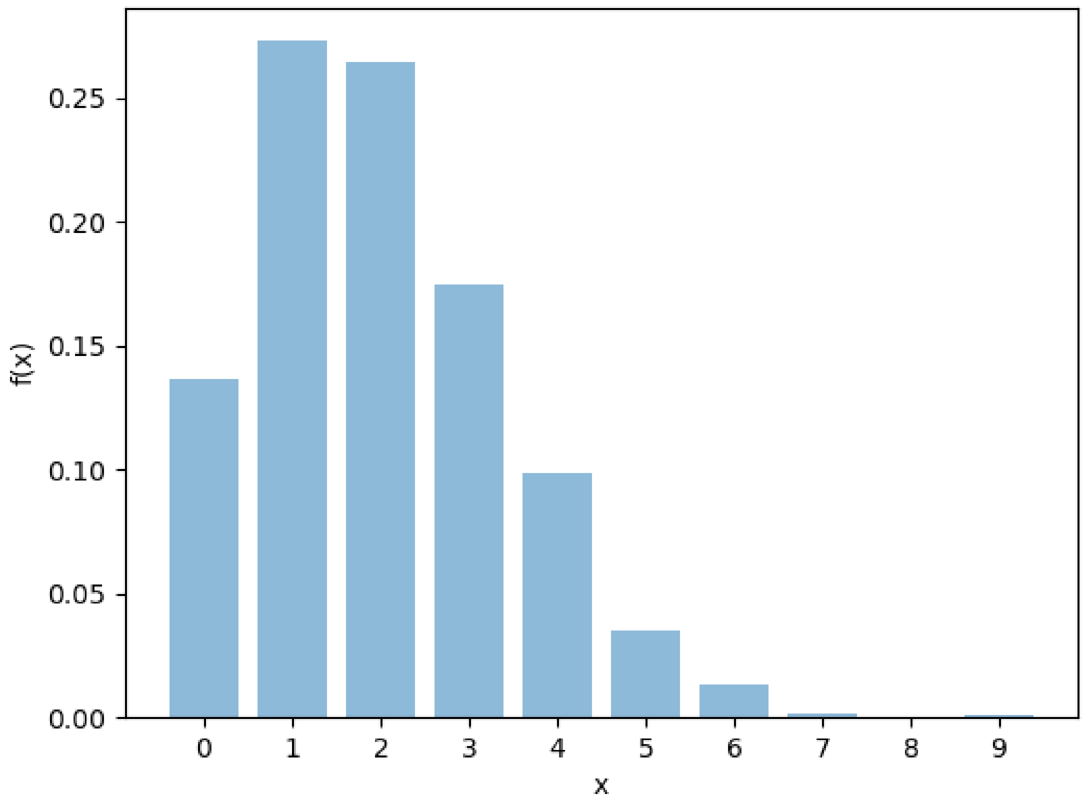
Terminal Output:



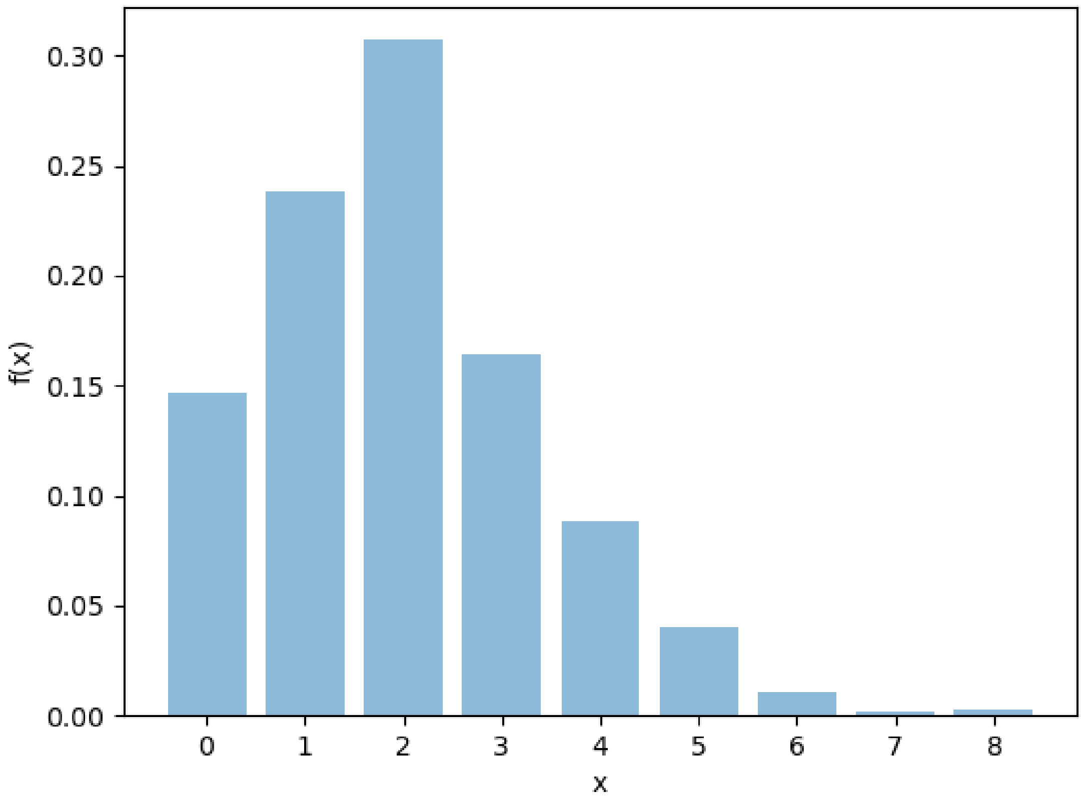
```
(base) hlsun:Simulation-and-Performance-Evaluation$ ./Homework4.2 b 1000 n 2000 s 12345 f "4_2_a_12345.csv"
Mean: 2
Standard Deviation: 1.41915
(base) hlsun:Simulation-and-Performance-Evaluation$ ./Homework4.2 b 1000 n 2000 s 246810 f "4_2_a_246810.csv"
Mean: 2
Standard Deviation: 1.41138
(base) hlsun:Simulation-and-Performance-Evaluation$ ./Homework4.2 b 1000 n 2000 s 13579 f "4_2_a_13579.csv"
Mean: 2
Standard Deviation: 1.42688
(base) hlsun:Simulation-and-Performance-Evaluation$ python Histogram.py 4_2_a_12345.csv 4_2_a_12345.png
(base) hlsun:Simulation-and-Performance-Evaluation$ python Histogram.py 4_2_a_246810.csv 4_2_a_246810.png
(base) hlsun:Simulation-and-Performance-Evaluation$ python Histogram.py 4_2_a_13579.csv 4_2_a_13579.png
(base) hlsun:Simulation-and-Performance-Evaluation$ ./Homework4.2 b 1000 n 10000 s 12345 f "4_2_b_12345.csv"
Mean: 10
Standard Deviation: 3.19906
(base) hlsun:Simulation-and-Performance-Evaluation$ ./Homework4.2 b 1000 n 10000 s 246810 f "4_2_b_246810.csv"
Mean: 10
Standard Deviation: 3.11641
(base) hlsun:Simulation-and-Performance-Evaluation$ ./Homework4.2 b 1000 n 10000 s 13579 f "4_2_b_13579.csv"
Mean: 10
Standard Deviation: 3.0835
(base) hlsun:Simulation-and-Performance-Evaluation$ python Histogram.py 4_2_b_12345.csv 4_2_b_12345.png
(base) hlsun:Simulation-and-Performance-Evaluation$ python Histogram.py 4_2_b_246810.csv 4_2_b_246810.png
(base) hlsun:Simulation-and-Performance-Evaluation$ python Histogram.py 4_2_b_13579.csv 4_2_b_13579.png
(base) hlsun:Simulation-and-Performance-Evaluation$
```

Part A:

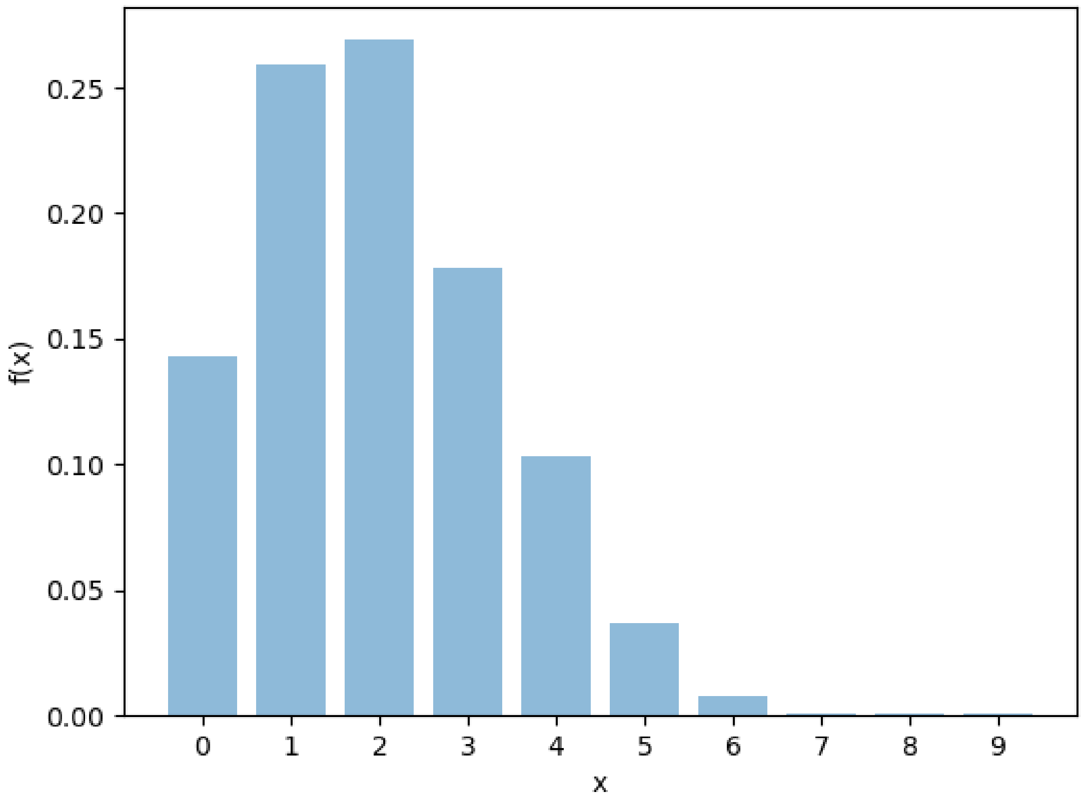
Seed: 12345



Seed: 13579

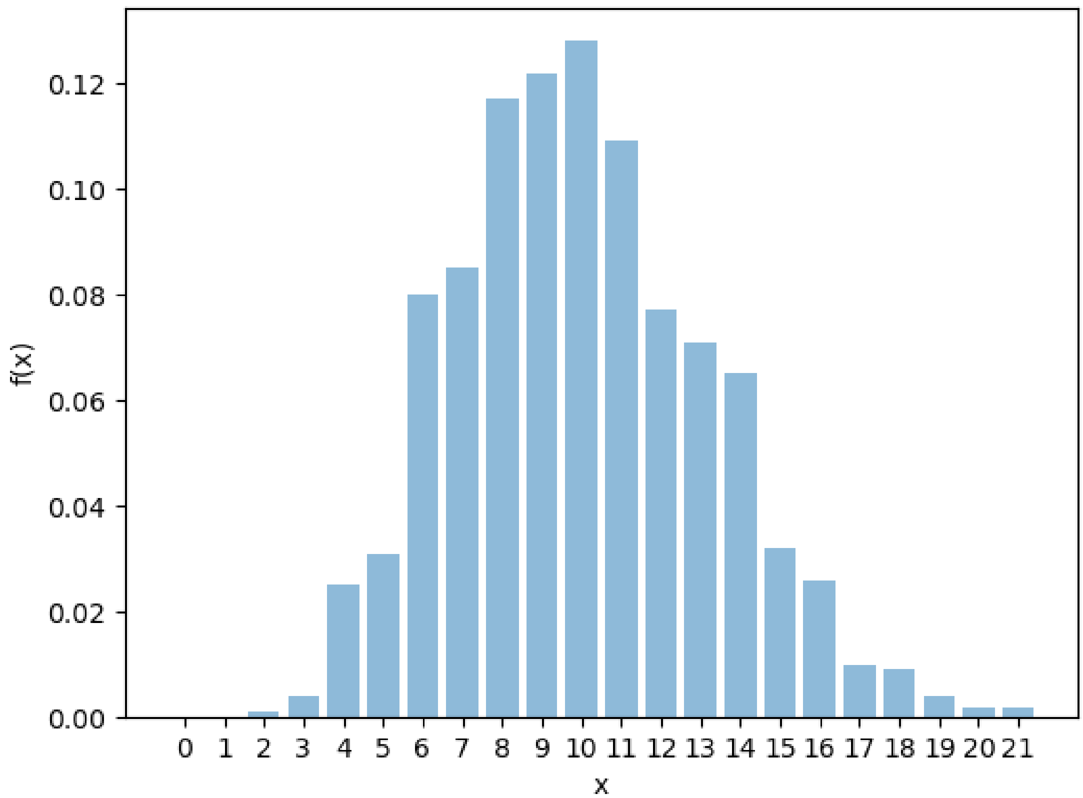


Seed: 246810

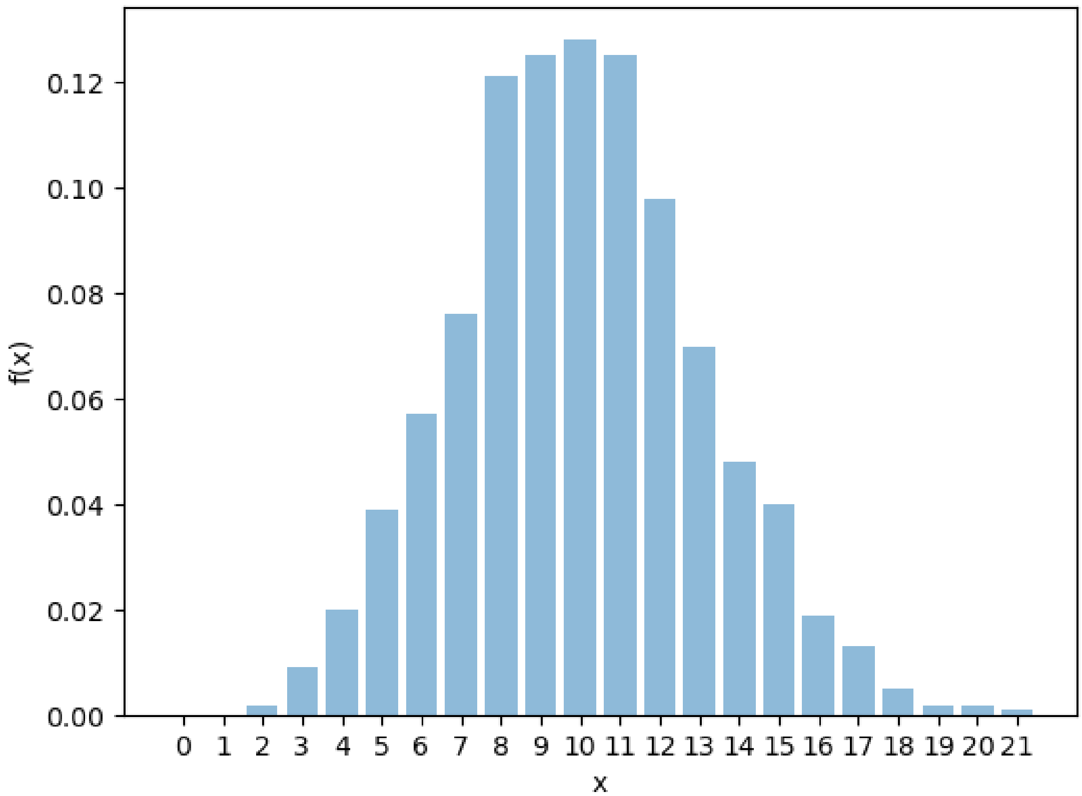


Part B:

Seed: 12345

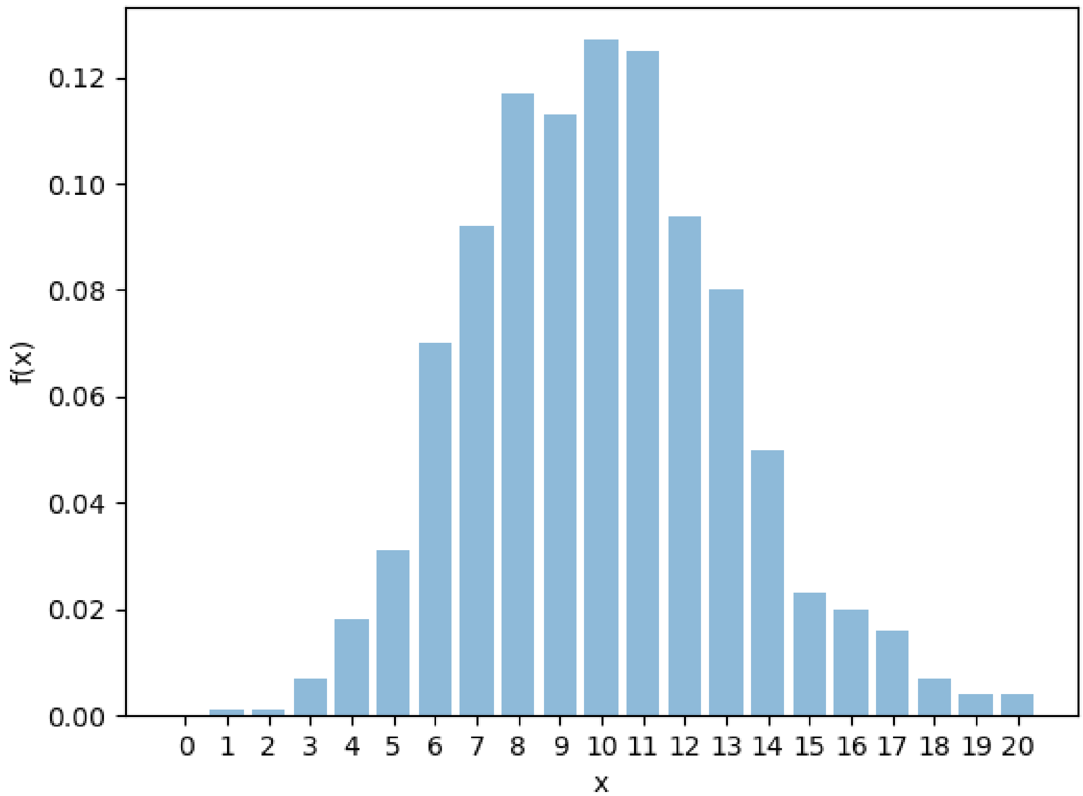


Seed: 13579



0.00
0.02
0.04
0.06
0.08
0.10
0.12
f(x)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
x

Seed: 246810



0.12
0.10
0.08
0.06
0.04
0.02
0.00
f(x)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
x

```
/**
 * Harrison Sun (sun.har@northeastern.edu)
 * EECE 5643 - Simulation and Performance Evaluation
 * Homework 4.2
 */

#include <cstdlib>
#include <cstring>
#include <fstream>
#include <stdio.h>
#include <exception>
#include <iostream>
#include <list>
#include <math.h>
#include <string>
#include <vector>
#include "c_lib/rvgs.h"
#include "c_lib/rngs.h"
#include "checkarg/checkarg.h"

#define DEFAULT_N_BOX     1000
#define DEFAULT_N_BALLS    2000
#define DEFAULT_SEED      12345
#define DEFAULT_OFILE    "output.txt"

/**
 * int main() - The main function
 *
 * @param int argc - the number of arguments
 * @param char* argv[] - the arguments
 *
 * @return 0 if the program runs successfully
 */

int main(int argc, char* argv[])
{
  // Variable Declarations
  int nBoxes{};
  int nBalls{};
  std::string outputFileName{};

  // doubly linked list <box number, number of balls in box>
  std::list<std::pair<long, long>> boxCount;

    // Set the seed
    for (int i = 0; i < argc; ++i)
    {
        if (*argv[i] == 's' && checkArg(argv[i + 1]))
        {
            PutSeed(std::stol(argv[i + 1]));
            break;
        }
        else
        {
            PutSeed(DEFAULT_SEED);
        }
    }

  // Set the number of boxes
  for (int i = 0; i < argc; ++i)
  {
    if (*argv[i] == 'b' && checkArg(argv[i + 1]))
    {
      nBoxes = std::stol(argv[i + 1]);
      break;
```

```
    }
    else
    {
      nBoxes = DEFAULT_N_BOX;
    }
  }

  // Set the number of balls
  for (int i = 0; i < argc; ++i)
  {
    if (*argv[i] == 'n' && checkArg(argv[i + 1]))
    {
      nBalls = std::stol(argv[i + 1]);
      break;
    }
    else
    {
      nBalls = DEFAULT_N_BALLS;
    }
  }

  // Set the filestream
  for (int i = 0; i < argc; ++i)
  {
    if (*argv[i] == 'f')
    {
      outputFileName = argv[i + 1];
      break;
    }
    else
    {
      outputFileName = DEFAULT_OFILE;
    }
  }

  /* Generate balls and determine which box they go in. */
  for (int i = 0; i < nBalls; ++i)
  {
    // Equilikely distribution between box 0 and box nBoxes - 1
    int box = Equilikely(0, nBoxes - 1);

    // Iterate through the linked list looking for the box number. If it is found, increment the count.
    // If the box is not in the list, add it to the list and set the count to 1.
    // The box is sorted in ascending order.
    bool found = false;
    for (auto it = boxCount.begin(); it != boxCount.end(); ++it)
    {
      // The box is in the list
      if (it->first == box)
      {
        it->second++;
        found = true;
        break;
      }

      // The box is not in the list and a greater box value is found
      else if (it->first > box)
      {
        boxCount.insert(it, std::make_pair(box, 1));
        found = true;
        break;
      }
    }

```

```
    // The box is not in the list and is greater than all boxes currently in the list
    if (!found)
    {
      boxCount.push_back(std::make_pair(box, 1));
    }
  }

  // Traverse the linked list nBalls times and determine how many boxes contain n number of balls.
  std::vector<long> boxCountVector;
  for (int i = 0; i < nBalls; ++i)
  {
    long count = 0;
    for (auto it = boxCount.begin(); it != boxCount.end(); ++it)
    {
      if (it->second == i)
      {
        count++;
      }
    }
    boxCountVector.push_back(count);
  }

  // Remove trailing zeros
  for (auto iter = boxCountVector.end() - 1; iter != boxCountVector.begin(); --iter)
  {
    if (*iter == 0)
    {
      boxCountVector.pop_back();
    }
    else
    {
      break;
    }
  }

  // Set the number of boxes with zero balls to nBoxes subtracted by the sum of the number of boxes that have balls in them
  long sum{ 0 };
  for (auto iter = boxCountVector.begin(); iter != boxCountVector.end(); ++iter)
  {
    sum += *iter;
  }
  boxCountVector.at(0) =  (nBoxes - sum);

  // Store boxCountVector as a csv file for plotting in Python
  std::ofstream outputFile;
  outputFile.open(outputFileName);
  long boxNum{ 0 };
  for (auto iter = boxCountVector.begin(); iter != boxCountVector.end(); ++iter)
  {
    // Output the probability of a box containing n number of balls
    outputFile << boxNum++ << ", " << (double)  *iter / nBoxes << std::endl;
  }
  outputFile.close();

  // Calculate the mean and standard deviation
  double mean{ 0 };
  double stdDev{ 0 };
  boxNum = 0;

  for (auto iter = boxCountVector.begin(); iter != boxCountVector.end(); ++iter)
  {
    mean += (double) * iter * boxNum / nBoxes;
    ++boxNum;
```

```
  }

  boxNum = 0;

  for (auto iter = boxCountVector.begin(); iter != boxCountVector.end(); ++iter)
  {
    stdDev += (double) * iter * pow((boxNum - mean), 2);
    ++boxNum;
  }

  stdDev = sqrt(stdDev / nBoxes);

  std::cout << "Mean: " << mean << std::endl;
  std::cout << "Standard Deviation: " << stdDev << std::endl;

  return 0;
}
```

3 Ex. 4.2.11

A test is compiled by selecting 12 different questions, at random and without replacement, from a well-published list of 120 questions. After studying this list you are able to classify all 120 questions into two classes, I and II. Class I Questions are those about which you feel confident; the remaining questions define class II. Assume that your grade probability conditioned on the class of the problem, is

	A	B	C	D	E
Class I	0.6	0.3	0.1	0.0	0.0
Class II	0.0	0.1	0.4	0.4	0.1

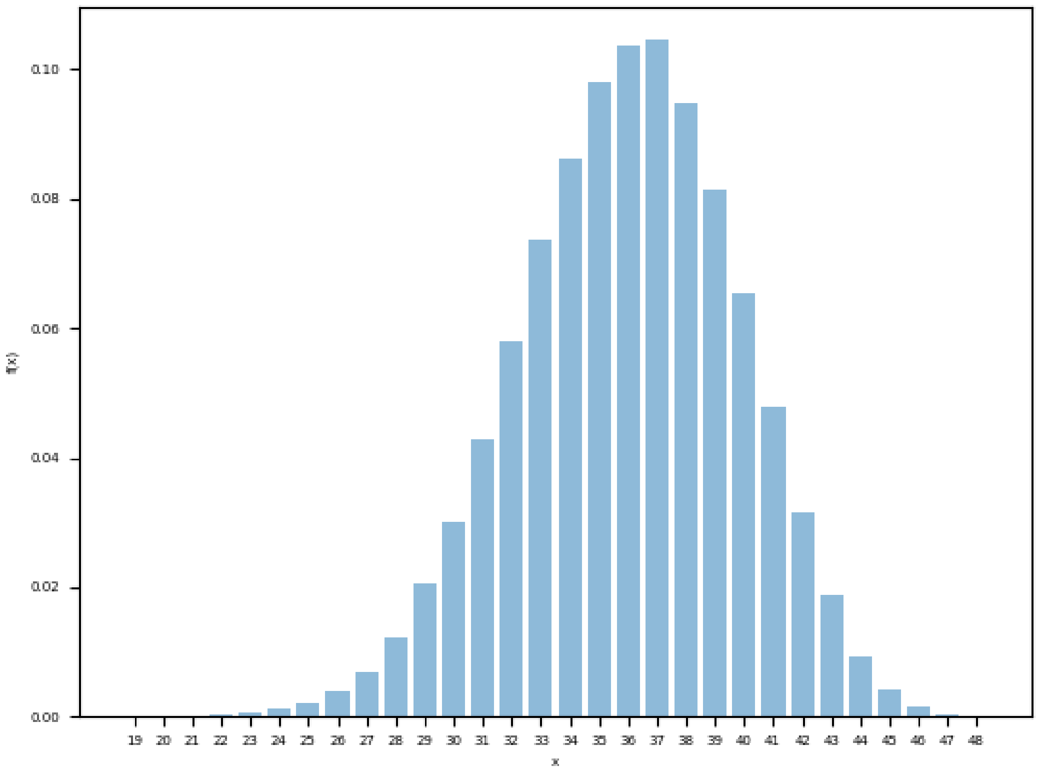
Each test question is graded on an A = 4, B = 3, C = 2, D = 1, F = 0 scale and a score of 36 or better is required to pass the test.
(a) If there are 90 Class I questions in the list, use Monte Carlo Simulation and 100,000 replications to generate a discrete-data histogram of scores.
(b) From this histogram, what is the probability that you will pass the test?

The probability of passing the test is approximately 56%.

Terminal Output:

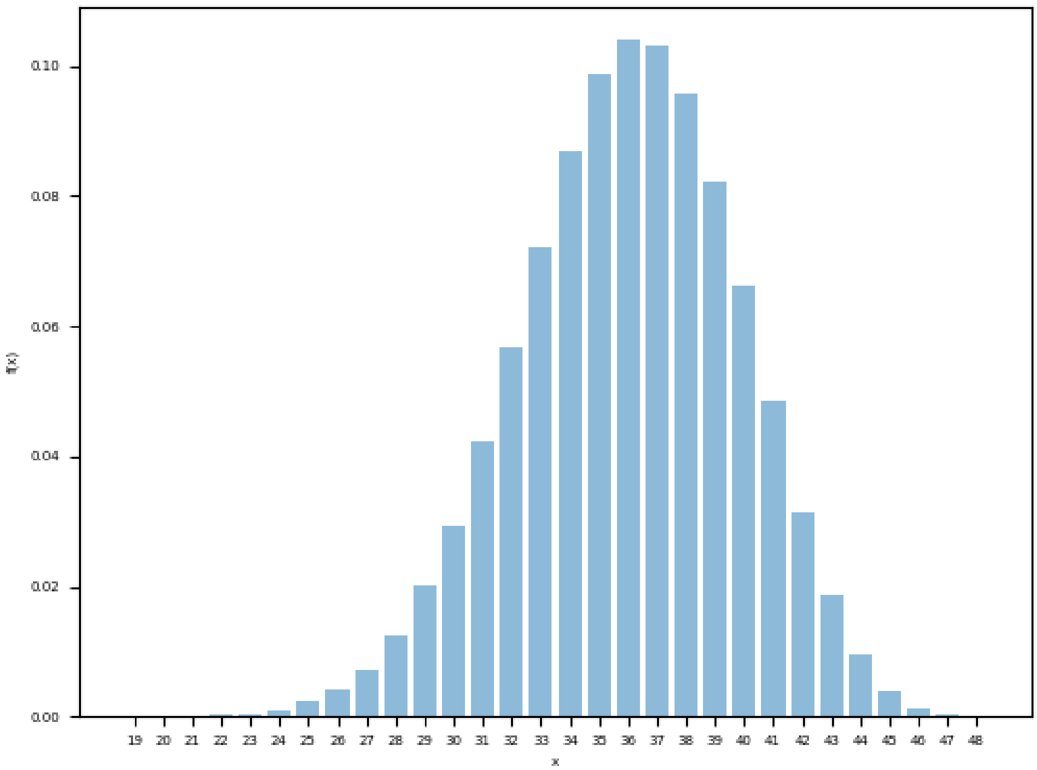
```
hlsun: Simulation-and-Perforr
hlsun:Simulation-and-Performance-Evaluation$ ./Homework4.3 n 100000 s 123456 f 4_3_123456.csv
Saving to file: 4_3_123456.csv
The probability of scoring at least 36 is 0.56627
hlsun:Simulation-and-Performance-Evaluation$ ./Homework4.3 n 100000 s 13579 f 4_3_13579.csv
Saving to file: 4_3_13579.csv
The probability of scoring at least 36 is 0.56511
hlsun:Simulation-and-Performance-Evaluation$ ./Homework4.3 n 100000 s 2468 f 4_3_2468.csv
Saving to file: 4_3_2468.csv
The probability of scoring at least 36 is 0.5634
hlsun:Simulation-and-Performance-Evaluation$ python3 Histogram.py 4_3_123456.csv 4_3_123456.png
hlsun:Simulation-and-Performance-Evaluation$ python3 Histogram.py 4_3_13579.csv 4_3_13579.png
hlsun:Simulation-and-Performance-Evaluation$ python3 Histogram.py 4_3_2468.csv 4_3_2468.png
hlsun:Simulation-and-Performance-Evaluation$
```

Seed: 2468



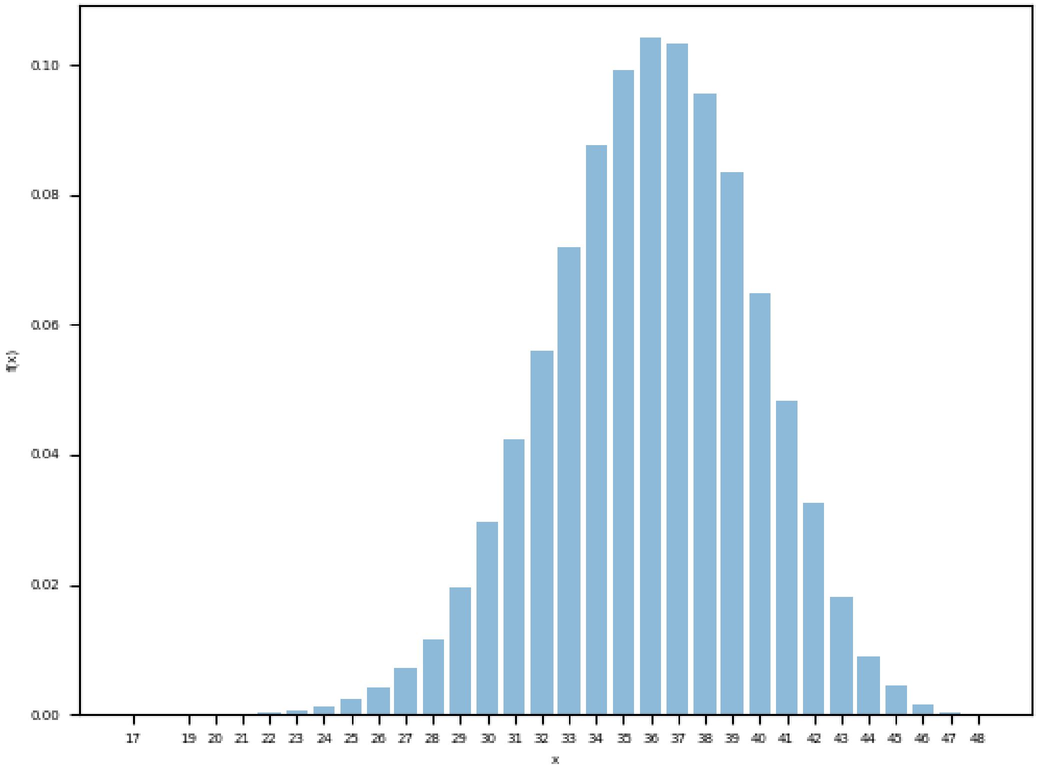
0.10
0.08
0.06
0.04
0.02
0.00
f(x)
19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
x

Seed: 13579



0.10
0.08
0.06
0.04
0.02
0.00
f(x)
19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
x

Seed: 123456


0.10
0.08
0.06
0.04
0.02
0.00
f(x)
17
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
x

```
/**
 * Harrison Sun (sun.har@northeastern.edu)
 * EECE 5643 - Simulation and Performance Evaluation
 * Homework 4.3
 */

#define DEFAULT_SEED        12345
#define DEFAULT_N_RUNS        100000
#define DEFAULT_PASSING    36
#define DEFAULT_OFILE    "output.txt"

// Define the grade weights
#define WEIGHT_A        4
#define WEIGHT_B        3
#define WEIGHT_C        2
#define WEIGHT_D        1
#define WEIGHT_F        0

#include <cstdlib>
#include <cstring>
#include <fstream>
#include <stdio.h>
#include <exception>
#include <iostream>
#include <list>
#include <math.h>
#include <string>
#include <vector>
#include "c_lib/rvgs.h"
#include "c_lib/rngs.h"
#include "checkarg/checkarg.h"

/**
 * int weight
 *
 * @param int n - The grade decision
 * @param int A - The weight of A
 * @param int B - The weight of B
 * @param int C - The weight of C
 * @param int D - The weight of D
 * @param int F - The weight of F
 *
 * @return int grade - The weighted grade
 */

int weight(int n, int A, int B, int C, int D, int F)
{
  int grade = 0;
  switch (n) {
  case 0:
    grade = A;
    break;
  case 1:
    grade = B;
    break;
  case 2:
    grade = C;
    break;
  case 3:
    grade = D;
    break;
  case 4:
    grade = F;
    break;
  }
```

```cpp
  return grade;
}

 /**
  * int main() - The main function
  *
  * @param int argc - the number of arguments
  * @param char* argv[] - the arguments
  *
  * @return 0 if the program runs successfully
  */

int main(int argc, char* argv[])
{
  // There are 120 possible questions in the exam
  const int num_q{ 120 };
  // 90 Questions are classified as Class 1
  const int num_c1{ 90 };
  // 30 Questions are classified as Class 2
  const int num_c2{ 30 };
  // Doubly linked list to store the number of times each score is achieved <int score, long number of times scored>
  std::list<std::pair<int, long>> scoreCount;

    // Set the seed
    for (int i = 0; i < argc; ++i)
    {
        if (*argv[i] == 's' && checkArg(argv[i + 1]))
        {
            PutSeed(std::stol(argv[i + 1]));
            break;
        }
        else
        {
            PutSeed(DEFAULT_SEED);
        }
    }

  // Set the number of runs
  int nRuns{};
  for (int i = 0; i < argc; ++i)
  {
    if (*argv[i] == 'n' && checkArg(argv[i + 1]))
    {
      nRuns = std::stoi(argv[i + 1]);
      break;
    }
    else
    {
      nRuns = DEFAULT_N_RUNS;
    }
  }

  // Set the passing score
  int passingScore{};
  for (int i = 0; i < argc; ++i)
  {
    if (*argv[i] == 'p' && checkArg(argv[i + 1]))
    {
      passingScore = std::stoi(argv[i + 1]);
      break;
    }
    else
    {
      passingScore = DEFAULT_PASSING;
```

```
    }
  }

  // Set the grade weights
  int weightA{};
  int weightB{};
  int weightC{};
  int weightD{};
  int weightF{};

  for (int i = 0; i < argc; ++i)
  {
    if (*argv[i] == 'A' && checkArg(argv[i + 1]))
    {
      weightA = std::stoi(argv[i + 1]);
      break;
    }
    else
    {
      weightA = WEIGHT_A;
    }
  }

  for (int i = 0; i < argc; ++i)
  {
    if (*argv[i] == 'B' && checkArg(argv[i + 1]))
    {
      weightB = std::stoi(argv[i + 1]);
      break;
    }
    else
    {
      weightB = WEIGHT_B;
    }
  }

  for (int i = 0; i < argc; ++i)
  {
    if (*argv[i] == 'C' && checkArg(argv[i + 1]))
    {
      weightC = std::stoi(argv[i + 1]);
      break;
    }
    else
    {
      weightC = WEIGHT_C;
    }
  }

  for (int i = 0; i < argc; ++i)
  {
    if (*argv[i] == 'D' && checkArg(argv[i + 1]))
    {
      weightD = std::stoi(argv[i + 1]);
      break;
    }
    else
    {
      weightD = WEIGHT_D;
    }
  }

  for (int i = 0; i < argc; ++i)
  {
    if (*argv[i] == 'F' && checkArg(argv[i + 1]))
```

```
    {
      weightF = std::stoi(argv[i + 1]);
      break;
    }
    else
    {
      weightF = WEIGHT_F;
    }
  }

  // Set the filestream
  std::string outputFileName{};
  for (int i = 0; i < argc; ++i)
  {
    if (*argv[i] == 'f')
    {
      outputFileName = argv[i + 1];
      break;
    }
    else
    {
      outputFileName = DEFAULT_OFILE;
    }
  }

  // Class conditional probability
  const std::vector<double> class1Prob{ 0.6, 0.3, 0.1, 0.0, 0.0 };
  const std::vector<double> class2Prob{ 0.0, 0.1, 0.4, 0.4, 0.1 };


  // Run the simulation nRuns times
  for (int n = 0; n < nRuns; ++n)
  {
    // Reset the number of questions to the default
    int nq { num_q  };
    int nc1{ num_c1 };
    int nc2{ num_c2 };

    // Select 12 questions from the 120 questions without replacement
    // Determine the class and value of each question
    for (int i = 0; i < 12; ++i)
    {
      // Select a question
      int question = Equilikely(1, nq--); /* Note to self: nq-- is used here to decrement nq AFTER taking the Equilikely. --nq would not work the same way */
      // Determine the class of the question
      int questionClass = (question <= nc1) ? 1 : 2;
      // Determine the value of the question based on the class conditional probability and decrement the question class
      double random01 = Uniform(0, 1);
      // Store the cumulative value of the questions
      static int cumulativeValue{ 0 };

      if (questionClass == 1)
      {
        for (int j = 0; j < (int) class1Prob.size(); ++j)
        {
          if (random01 < class1Prob[j])
          {
            cumulativeValue += weight(j, weightA, weightB, weightC, weightD, weightF);
            break;
          }
          else
          {
            // This adjusts the randomly generated Uniform(0, 1) to account for rejecting
```

```cpp
    the first comparison. The sum of the class conditional pdf is 1.0.
          random01 -= class1Prob[j];
        }
      }
      --nc1;
    }

    else  // (questionClass == 2)
    {
      for (int j = 0; j < (int) class2Prob.size(); ++j)
      {
      if (random01 < class2Prob[j])
        {
          cumulativeValue += weight(j, weightA, weightB, weightC, weightD, weightF);
          break;
        }
      else
        {
          // This adjusts the randomly generated Uniform(0, 1) to account for rejecting the first comparison. The sum of the class conditional pdf is 1.0.
          random01 -= class2Prob[j];
        }
      }
      --nc2;
    }
    // For the final question, store the score in scoreCount
    if (i == 11)
    {
      // Iterate through the linked list to find the correct scoreCount
      // If the score is not in the list, add it to the list and set the count to 1.
      // The score is sorted in ascending order.
      bool found = false;
      for (auto it = scoreCount.begin(); it != scoreCount.end(); ++it)
      {
        // The cumulativeValue is in the list
        if (it->first == cumulativeValue)
        {
          it->second++;
          found = true;
          break;
        }

        // The box is not in the list and a greater cumulativeValue value is found
        else if (it->first > cumulativeValue)
        {
          scoreCount.insert(it, std::make_pair(cumulativeValue, 1));
          found = true;
          break;
        }
      }
      // The cumulativeValue is not in the list and is greater than all cumulativeValues currently in the list
      if (!found)
      {
        scoreCount.push_back(std::make_pair(cumulativeValue, 1));
      }
      // Reset the cumulativeValue to 0 for the next run
      cumulativeValue = 0;
    } // End if (i == 11)
  } // End Question Selection
}
std::cout << "Saving to file: " << outputFileName << std::endl; /*************************************/
// Store the contents of the linked list in csv format for plotting in Python
std::ofstream outputFile;
```

```
  outputFile.open(outputFileName);
  for (auto it = scoreCount.begin(); it != scoreCount.end(); ++it)
  {
    outputFile << it->first << "," << (double) it->second / (double) nRuns << std::endl;
  }
  outputFile.close();

  // Determine whether pass or fail
  long passCount{ 0 };
  for (auto it = scoreCount.begin(); it != scoreCount.end(); ++it)
  {
    if (it->first >= passingScore)
    {
      passCount += it->second;
    }
  }

  std::cout << "The probability of scoring at least " << passingScore << " is " << (double)
    passCount / static_cast<double>(nRuns) << std::endl;
  return 0;
}
```

4 Ex. 4.3.5

(a) Construct a continuous-data histogram of the service times (in ac.dat - Exercise 1.2.6).
(b) Compare the histogram mean and standard deviation with the corresponding sample mean and standard deviation, and justify your choice of histogram parameters a, b, and either k or δ.

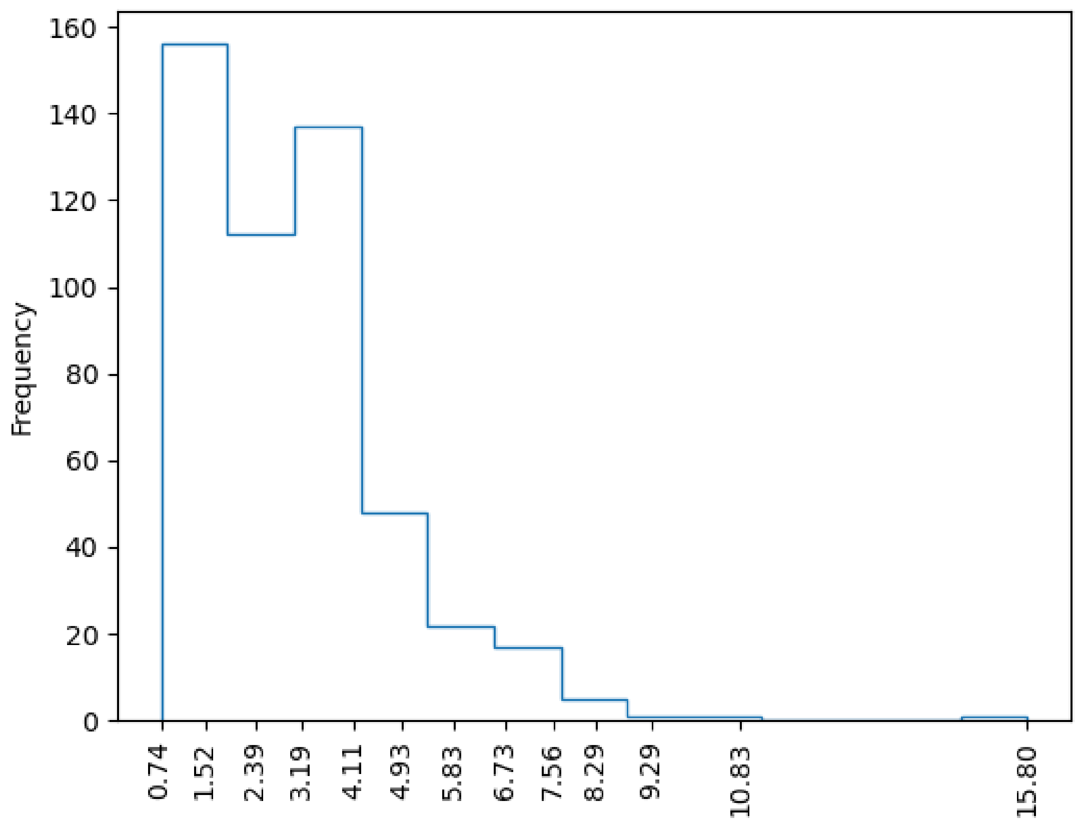
Terminal Output

```
hlsun: Simulation-and-Perforr
(base) hlsun:Simulation-and-Performance-Evaluation$ make
make: Nothing to be done for 'all'.
(base) hlsun:Simulation-and-Performance-Evaluation$ ./Homework4.4 i ac.dat o 4_4.csv
The sample mean: 3.03189 and sample standard deviation: 1.82446
The histogram mean: 3.03189 and histogram standard deviation: 1.8081
(base) hlsun:Simulation-and-Performance-Evaluation$ python3 Continuous_Hist.py 4_4.csv 4_4.png
(base) hlsun:Simulation-and-Performance-Evaluation$
```

The sample mean: 3.03
The sample standard deviation: 1.82
The histogram mean: 3.03
The histogram standard deviation: 1.81

We can see that the sample and histogram mean and standard deviation are similar. This means that the number of bins used was sufficient to ensure an accurate representation of the data. Additionally, the histogram is mostly smooth, meaning the number of bins used was not excessively high in order to achieve this sample statistic performance.

a and b were chosen to be the minimum and maximum values, as this allows the histogram to fit all of the data while also not allocating bins for data outside the range. The number of bins was chosen such that $k \in [\lfloor ln(n) \rfloor, \lfloor \sqrt{n} \rfloor]$. Given that $n = 500$, $k \in [6, 22]$. 18 was chosen because more bins than the minimum were needed knowing that outlier data is significantly larger than the mean. This can been seen in the histogram below, where there are several empty bins before reaching the maximum bin. However, setting k too high, such as $k = 22$ would result in a noisy histogram. Therefore, it was decided that $k = 18$.



Frequency
160
140
120
100
80
60
40
20
0
0.74
1.52
2.39
3.19
4.11
4.93
5.83
6.73
7.56
8.29
9.29
10.83
15.80

```cpp
/**
 * Harrison Sun (sun.har@northeastern.edu)
 * EECE 5643 - Simulation and Performance Evaluation
 * Homework 4.4
 */

#define DEFAULT_IFILE   "ac.dat"
#define DEFAULT_OFILE   "output.txt"

#include <algorithm>
#include <cstdlib>
#include <cstring>
#include <fstream>
#include <stdio.h>
#include <exception>
#include <iostream>
#include <list>
#include <math.h>
#include <string>
#include <vector>
#include "c_lib/rvgs.h"
#include "c_lib/rngs.h"
#include "checkarg/checkarg.h"

void sortDataIntoBins(const std::vector<double>& data, int numBins, std::vector<std::pair<double, long>>& meanOfBinAndCount) {
  // Initialize the meanOfBinAndCount vector with empty bins and counts of 0
  meanOfBinAndCount.clear();
  for (int i = 0; i < numBins; i++) {
    meanOfBinAndCount.push_back(std::make_pair(0.0, 0));
  }

  // Find the range of the data
  double minData = *std::min_element(data.begin(), data.end());
  double maxData = *std::max_element(data.begin(), data.end());
  double range = maxData - minData;

  // Calculate the width of each bin
  double binWidth = range / numBins;

  // Sort the data into bins and update the mean and count of each bin
  for (int i = 0; i < data.size(); i++) {
    // Determine which bin the data point belongs to
    int binIndex = std::min(static_cast<int>((data[i] - minData) / binWidth), numBins - 1);

    // Update the mean and count of the bin
    double binMean = meanOfBinAndCount[binIndex].first;
    long binCount = meanOfBinAndCount[binIndex].second;
    binMean = (binMean * binCount + data[i]) / (binCount + 1);
    binCount++;
    meanOfBinAndCount[binIndex] = std::make_pair(binMean, binCount);
  }
}



/**
 * int main() - The main function
 *
 * @param int argc - the number of arguments
 * @param char* argv[] - the arguments
 *
 * @return 0 if the program runs successfully
 *
 * This is the main function. It reads in a data file in tab separated format. The first
```

```
   column contains the arrival times and the second column
 * contains departure times.
 *
 */

int main(int argc, char* argv[])
{
  // Create two vectors to store the data
  std::vector<double> arrival_times{};
  std::vector<double> departure_times{};
  std::vector<double> service_times{};

  // Histogram Parameters
  int nHistogram { 500 }; // The number of jobs in the data file
  // The number of bins in the histogram. Ideally, k ~ [ln(n), sqrt(n)] => Choosing from k ~ [6, 22]

  /*
    ***************************************************************************************************

    The number of bins is chosen to be 18 This is significantly larger than the minimum k = 6. The reason for this choice is that the service times are empirical
    data and may have significant outliers that would not fit well with other data when given too few bins. However, 22 bins would be too many, as the data has
    high variance and would thus have a lot of empty bins. 18 bins is a good compromise between the two.

    ***************************************************************************************************
    */

  int nBins{18};

  // File names
  std::string inputFileName{};
  std::string outputFileName{};

  // Set the input file stream
  for (int i = 0; i < argc; ++i)
  {
    if (*argv[i] == 'i')
    {
      inputFileName = argv[i + 1];
      break;
    }
    else
    {
      inputFileName = DEFAULT_IFILE;
    }
  }

  // Set the output file stream
  for (int i = 0; i < argc; ++i)
  {
    if (*argv[i] == 'o')
    {
      outputFileName = argv[i + 1];
      break;
    }
    else
    {
      outputFileName = DEFAULT_OFILE;
    }
  }
```

```

double arrival_time{};
double departure_time{};

std::ifstream infile;
infile.open(inputFileName.c_str());

// Read in the data from the tsv to the vectors
while (infile >> arrival_time >> departure_time)
{
  arrival_times.push_back(arrival_time);
  departure_times.push_back(departure_time);
}
infile.close();

// Calculate the service times and store them in the service_times vector
for (int i = 0; i < arrival_times.size(); ++i)
{
  // Check if the service node is free at arrival
  if (arrival_times[i] > departure_times[i - 1])
  {
    service_times.push_back(departure_times[i] - arrival_times[i]);
  }
  // If the job has to wait in a queue, the service starts after the previous job is finished
  else
  {
    service_times.push_back(departure_times[i] - departure_times[i - 1]);
  }
}

// Calculate the total service time
double total_service_time{};
for (std::vector<double>::iterator i = service_times.begin(); i != service_times.end(); ++i)
{
  total_service_time += *i;
}

// Calculate the sample mean and standard deviation
double sample_mean{ total_service_time / service_times.size() };
double sample_std{};

// Standard Deviation
for (std::vector<double>::iterator i = service_times.begin(); i != service_times.end(); ++i)
{
  sample_std += pow(*i - sample_mean, 2);
}
sample_std = sqrt(sample_std / (service_times.size() - 1));

// Vector to store the bin information as a pair
std::vector<std::pair<double, long>> BinVector;

// Call the function to sort the data into bins
sortDataIntoBins(service_times, nBins, BinVector);

// Store the bin means and counts in output file
std::ofstream outfile;
outfile.open(outputFileName.c_str());

for (int i = 0; i < nBins; ++i)
{
  outfile << BinVector[i].first << ", " << BinVector[i].second << std::endl;
}
```

```

  outfile.close();

  // Calculate the histogram mean and standard deviation
  double histogram_mean{};
  double histogram_std{};

  for (int i = 0; i < nBins; ++i)
  {
    histogram_mean += BinVector[i].first * BinVector[i].second;
  }

  histogram_mean /= nHistogram;

  for (int i = 0; i < nBins; ++i)
  {
    histogram_std += BinVector[i].second * pow(BinVector[i].first - histogram_mean, 2);
  }

  histogram_std = sqrt(histogram_std / (nHistogram - 1));

  // Output the means and standard deviations
  std::cout << "The sample mean: " << sample_mean << " and sample standard deviation: " <<
    sample_std << std::endl;
  std::cout << "The histogram mean: " << histogram_mean << " and histogram standard
    deviation: " << histogram_std << std::endl;
  return 0;
}
```

5 Appendices: Histogram Code

Histogram:

```
# Harrison Sun (sun.har@northeastern.edu)
# EECE 5643 - Simulation and Performance Evaluation

# This python script generates a bar chart taking in csv inputs. The first column is the x
    axis value and the second column is the y axis value.

import numpy as np
import matplotlib.pyplot as plt
import sys

# Read in the csv file
data = np.genfromtxt(sys.argv[1], delimiter=',')

# Set font size
# plt.rcParams.update({'font.size': 5})

# Plot the bar chart
plt.bar(data[:,0], data[:,1], align='center', alpha=0.5)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.xticks(data[:,0])

# Save the bar chart
plt.savefig(sys.argv[2])
```

Continuous Histogram:

```
import sys
import pandas as pd
import matplotlib.pyplot as plt

# Get the input and output file names from the command-line arguments
input_file = sys.argv[1]
output_file = sys.argv[2]

# Read the data from the CSV file
data = pd.read_csv(input_file, header=None)

# Filter out the rows with 0 values
data = data[data[0] != 0]

# Create the step histogram
plt.hist(data[0], bins=data[0].nunique(), weights=data[1], histtype='step')

# Set the x-axis label
plt.xlabel('Values')

# Set the y-axis label
plt.ylabel('Frequency')

# Set the x-axis tick labels
plt.xticks(data[0])

# Rotate x labels
plt.xticks(rotation=90)

# Save the plot to a PNG file
plt.savefig(output_file)
```