HW1

Brandon Walker

Introduction

This document outlines the test plan for the dice-rolling software component. The purpose of this component is to "simultaneously" roll up to five six-sided dice, with the number of dice to be rolled specified by the user. The primary focus of this test plan is to rigorously verify the randomness and independence of the generated results, as per the project requirements.

Assumptions:

- The user input for the number of dice to roll will be a valid integer between 1 and 5, inclusive. Error checking for invalid inputs is outside the scope of this project.
- The output of the dice-rolling routine will be returned as output parameters to the calling routine, not printed directly.
- The software under test is a separate component, and a test harness will be used to automate the testing process.
- Being able to instantiate a die, and use that die to get whatever number of rolls I call for, in a real-world scenario I would thus know that the number of dice called is irrelevant, for instance, I could call six million rolls, and separate it into whatever partitions I wanted for analysis, and it would have exactly the same randomness as calling for six dice a million times. I will be working on the assumption that I am supposed to pretend not to know this.

Instructions for Alice:

- To execute the test plan, run the test harness code which will call the dice-rolling component and perform the checks outlined in the following sections.
- The harness is designed to automate the data collection and analysis, including the Chi-squared tests.
- The results will be logged and can be reviewed in the "Actual Output" sections of each test case.

Tests to be used:

1. Distribution Tests

These check if each outcome happens about as often as it should.

- For a single die, each face (1–6) should appear equally often.
- For two or more dice, each possible sum (like 2–12 for two dice) should appear with the correct probability based on the number of ways it can happen.
- We use the **chi-squared goodness-of-fit test**, which compares the counts we observed with the counts we expected.
- If the differences are small (chi-squared value below the cutoff), the test passes. If it is too large, it fails.

2. Independence Tests

These check whether the dice are truly independent — meaning that one die or one roll does not influence another. I check independence in two ways:

- **Horizontal independence (within a single roll of multiple dice):** This checks that the dice in the same roll does not depend on each other. For example, on a two-dice roll, die #1 showing a 2 should not make it more or less likely that die #2 also shows a 2. If the dice are independent, the chance of rolling doubles (both dice equal) will automatically come out correct (1/6). Will also check run length, and frequency of

different lengths, for the horizontal runs it will check dice 1, to 2 etc, and then roll over to next roll if it's in the middle of a run.

- **Vertical independence (across rolls):** This checks that rolls do not depend on previous rolls. For example, if you roll a 3, the number that comes next should be no more or less likely to be a 4, a 5, or anything else. We test this by building a 6×6 "transition table" that records how often each number follows another. If the dice are independent across time, the table will match what we'd expect for random dice. Will also check run lengths across runs and check frequency of different lengths of runs across specific die, as well as independence of sums of rolls.

We use the **chi-squared test of independence** to check both horizontal and vertical independence. This compares the observed 6×6 table with the expected table under the assumption of independence. If the chi-squared value is below the cutoff, the dice pass the independence test.

Glossary of Formulas and Terms

Here are the main mathematical tools we are using, explained in plain language:

1. **Observed Count (O_i):**
   The number of times a particular outcome happened in our test. Example: if we rolled a single die 3,600,000 times and got "4" exactly 600,200 times, then $O_4 = 600{,}200$.

2. **Expected Count (E_i):**
   The number of times we *should* expect a particular outcome if the dice are fair. Example: for one die rolled 3,600,000 times, each face should appear 600,000 times, so $E_4 = 600{,}000$.

3. **Chi-squared statistic (χ^2):**
   A measure of how different the observed counts are from the expected counts. The formula is:
   - If O_i and E_i are close, χ^2 will be small.
   - If O_i and E_i are very different, χ^2 will be large.

4. **Degrees of Freedom (df):**
   A number used to determine the cutoff for the chi-squared test.
   - For a distribution test with k categories, df = k − 1.
   - For an independence test with an r*c table, df = (r − 1) x (c − 1).

5. **Critical Value:**
   The cutoff number we compare χ^2 against. It depends on the degrees of freedom and the significance level (1%). If χ^2 is larger than the critical value, the test fails.

6. **Significance Level ($\alpha = 0.01$):**
   The threshold we chose. It means we are willing to accept a 1% chance of mistakenly rejecting fair dice.

7. **Goodness-of-Fit Test (chi-squared version):**
   A test to see if observed counts match expected counts for a single categorical variable (like die faces or sums).

8. **Test of Independence (chi-squared version):**
   A test to see if two variables are independent (like die 1 and die 2 or roll t and roll t+1).

9. **Transition Table:**
   A 6×6 table showing how often one roll is followed by another. Used in horizontal independence tests.

10. **Lag-1 Autocorrelation:**
    A test that looks deeper than chi-squared, but only at linear relationships. Cutoff by $\pm\frac{2}{\sqrt{n}}$ with n being number of rolls.

11. **Permutation P-Value:**
    A test that looks deeper than chi-squared, for relationships. Returns a value from 0 to 1 that represents percent chance of chi-value being as high as it is, generally if chance is above 5, or 1 when being strict, it is considered unrelated. Computationally EXTREMELY expensive.

Test Plan

The following test cases are designed to verify the randomness and independence of the dice-rolling software. The sample sizes for each test are chosen to be large enough to provide statistically significant results for Chi-squared analysis.

Test Case 1: Single Die Distribution and Independence

- **Input:** Roll a single die 3.6 million times.
- **Expected Outputs:**
    - **Distribution:** Each face (1, 2, 3, 4, 5, 6) is expected to appear approximately 600,000 times (3,600,000/6). The Chi-squared test should show a p-value that indicates a uniform distribution.
    - **Independence:** The frequency of any specific roll being followed by another specific roll (e.g., a "3" followed by a "4" vs. a "3" followed by a "5") should be approximately equal. For example, a "3" followed by a "4" is expected to occur approximately 100,000 times (3,600,000/36). The Chi-squared test on the sequential pairs should not show a dependency between consecutive rolls.

Actual Output: Number of times each number was rolled:
1: 599689
2: 601080
3: 599593
4: 599883
5: 599712
6: 600043

Critical value (df=5, alpha=0.05): 11.07
Result: 2.54542 Pass

Transition Counts (for independence test):

	1	2	3	4	5	6
1	100102	99980	99782	100122	99813	99889
2	99829	100023	100817	99885	100406	100120
3	99481	100300	99774	100327	100185	99526
4	100029	100303	99832	99831	99867	100021
5	99651	100462	99624	100047	99641	100287
6	100597	100012	99764	99671	99799	100200

Critical value (df=25, alpha=0.05): 37.65
Result: 27.3578 Pass

Test Case 2: Two Dice Distribution, Sums, and Independence

- **Input:** Roll two dice 3.6 million times.
- **Expected Outputs:**
    - **Distribution:**
        - **Faces:** Each face (1, 2, 3, 4, 5, 6) is expected to appear approximately 1,200,000 times (3,600,000/6x2). The Chi-squared test will be used to test for a uniform distribution.
        - **Sums:** The distribution of the sum of the dice should follow a normal distribution. The Chi-squared test will be used to test for this distribution.
    - **Independence:**
        - **Horizontal:** The results of the first die across all rolls should be independent of the results of the second die for the same roll. A Chi-squared test will be used to test for significant dependency.
        - **Vertical:** The outcome of a die from one roll should not influence the outcome of a die from the next roll. Chi-squared test of independence will be used to test for significant dependency.
            - **Sums:** The outcome of the sums of one roll should not influence the outcome of another roll. A Chi-squared test will be used to test for significant dependency.
- **Actual Output:** Number of times each number was rolled:
  1: 1198987
  2: 1200547
  3: 1199698
  4: 1200599
  5: 1200121
  6: 1200048

Critical value (df=5, alpha=0.05): 11.07
Result: 1.49361 Pass

Sum distribution:
2: 100090
3: 199692
4: 299917
5: 399991
6: 499949
7: 599620
8: 500009
9: 400681
10: 299924
11: 200414
12: 99713

Critical value (df=10, alpha=0.05): 18.31
Result: 3.68384 Pass
Horizontal Transition Counts (for independence test):

	1	2	3	4	5	6
1	199193	200812	199359	199801	200018	199804
2	200152	199646	199894	200772	200383	199700

3	200468	200511	199670	199994	199427	199628
4	199673	199908	200163	200093	200598	200163
5	199958	199351	200326	199895	200063	200527
6	199542	200319	200286	200043	199632	200226

Critical value (df=25, alpha=0.05): 37.65
Result: 25.9478 Pass

Vertical Transition Counts (for independence test):

	1	**2**	**3**	**4**	**5**	**6**
1	200180	199692	199638	199727	200280	199470
2	199692	200558	200264	200311	199811	199911
3	199638	200264	198716	200339	199958	200783
4	199727	200311	200339	200280	199898	200044
5	200280	199811	199958	199898	199760	200414
6	199470	199911	200783	200044	200414	199426

Critical value (df=25, alpha=0.05): 37.65
Result: 27.1421 Pass

Sum Transition Counts (for independence test):

	2	3	4	5	6	7	8	9	10	11	12
2	2660	5597	8382	11112	13896	16736	13920	11122	8411	5540	2714
3	5498	11127	16745	22314	28086	32993	27780	22032	16548	11065	5504
4	8407	16707	25072	33262	41787	50273	41379	33417	24788	16551	8274
5	11238	22073	33382	44485	55494	66840	55510	44435	33277	22094	11163
6	13751	27583	41423	55579	69511	82780	69314	55685	42301	28048	13974
7	16669	33126	49949	66409	83017	100073	83765	66612	49972	33591	16437
8	14049	27852	41811	55410	69567	83176	69746	55687	41321	27755	13635
9	11087	22237	33174	44641	55529	66772	55322	45003	33539	22187	11189
10	8421	16726	25020	33279	41518	50052	41500	33247	24828	16925	8408
11	5536	11048	16598	22405	27832	33270	27791	22381	16714	11187	5652
12	2774	5616	8361	11094	13712	16655	13982	11060	8225	5471	2763

Critical value (df=100, alpha=0.01): 44.31
Result: 102.403 Fail
Lag-1 autocorrelation of sums(0.001054) : 0.000502873
Permutation empirical p-value for chi-square: 0.406593

Test Case 3: Three Dice Distribution and Independence

- **Inputs:** Roll three dice 4.32 million times.
- **Expected Outputs:**
  - **Distribution:**
    - **Faces:** Each face (1, 2, 3, 4, 5, 6) is expected to appear approximately 2,160,000 times (4,320,000/6x3). The Chi-squared test will be used to test for a uniform distribution.
    - **Sums:** The distribution of the sum of the dice should follow a normal distribution. The Chi-squared test will be used to test for this distribution.
  - **Independence:**
    - **Horizontal:** The results of any die should be independent of the results of any other die for the same roll. A Chi-squared test using an orthogonal array strength two will be used to test for significant dependency.
    - **Vertical:** The outcome of a die from one roll should not influence the outcome of a die from the next roll. Chi-squared test of independence on the sequential rolls will be used to test for significant dependency.
      - **Sums:** The outcome of the sums of one roll should not influence the outcome of another roll. A Chi-squared test will be used to test for significant dependency.

Actual Output: Number of times each number was rolled:

1: 2160094
2: 2158960
3: 2162155
4: 2160564
5: 2158216
6: 2160011

Critical value (df=5, alpha=0.05): 11.07
Result: 4.27562 Pass

Sum distribution:

3: 19779
4: 59795
5: 119819
6: 200843
7: 299954
8: 420043
9: 500180
10: 539206
11: 540617
12: 500762
13: 420186
14: 299290
15: 199674
16: 119690
17: 59966
18: 20196

Critical value (df=15, alpha=0.05): 26.12
Result: 15.1137 Pass

Horizontal Transition Counts (for independence test):

	1	2	3	4	5	6
1	360242	358794	361078	361355	359401	359224
2	360188	360219	359725	359963	359578	359286
3	360076	360459	361430	359014	360095	361081
4	359571	360085	360326	360710	359378	360492
5	360248	358921	359881	359469	359850	359847
6	359768	360482	359715	360052	359914	360080

Critical value (df=25, alpha=0.05): 37.65
Result: 30.6422 Pass

Vertical Transition Counts (for independence test):

	1	2	3	4	5	6
1	359568	359393	360542	360434	360094	360063
2	359704	359672	359566	360167	360249	359602
3	360422	360217	360881	360391	360185	360059
4	360600	359947	360852	359433	358971	360761
5	360136	359903	360223	360221	358798	358935
6	359664	359828	360091	359918	359919	360591

Critical value (df=25, alpha=0.05): 37.65
Result: 16.8437 Pass

Sum Transition Counts (for independence test):

	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	75	269	550	946	1434	1897	2396	2423	2511	2213	1858	1359	945	539	260	104
4	260	838	1717	2700	4232	5856	6910	7530	7537	6879	5774	4097	2743	1619	823	280
5	585	1654	3308	5476	8312	11787	13894	14884	14962	13865	11652	8289	5644	3333	1663	511
6	907	2721	5562	9408	13871	19712	23454	25094	25285	23294	19190	13968	9205	5528	2763	881
7	1360	4206	8444	13973	20884	29191	34720	37598	37401	34659	28994	20892	13854	8169	4203	1406
8	1946	5852	11629	19710	29009	40945	48267	52436	52127	49187	41079	28817	19432	11736	5854	2017
9	2309	6856	13736	23100	34868	48545	57926	62539	62255	58136	48977	34672	23178	13859	6895	2329
10	2432	7301	15020	25023	37453	52550	62342	67591	68008	62273	52187	37062	25052	15024	7366	2521
11	2412	7564	14998	25010	37519	52532	63215	67032	67177	62479	52924	37949	24941	14833	7602	2430
12	2305	7007	13776	23269	34874	48570	57857	62586	62770	58109	48640	34627	23188	13843	6962	2379
13	2019	5849	11699	19573	29122	40931	48589	52105	52885	48611	40594	28971	19563	11721	5908	2046
14	1313	4146	8268	13997	20816	28974	34539	37407	37128	34954	29572	20774	13576	8234	4197	1395
15	886	2749	5531	9382	13664	19382	23047	24917	25282	22954	19395	13876	9288	5639	2731	951
16	575	1661	3331	5527	8364	11537	13740	15118	15185	13826	11588	8247	5409	3369	1649	564
17	312	862	1682	2825	4138	5611	6966	7414	7563	6989	5831	4252	2738	1659	832	292
18	83	260	568	924	1394	2023	2318	2532	2540	2334	1931	1438	918	585	258	90

Critical value (df=225, alpha=0.01): 95.02
Result: 251.392 Fail
Lag-1 autocorrelation of sums (0.000962): 0.000807697
Permutation empirical p-value for chi-square: 0.106893

Test Case 4: Four Dice Distribution and Independence

- **Inputs:** Roll four dice 6.48 million times.
- **Expected Outputs:**
  - **Distribution:**
    - **Faces:** Each face (1, 2, 3, 4, 5, 6) is expected to appear approximately 4,320,000 times (4,320,000/6x4). The Chi-squared test will be used to test for a uniform distribution.
    - **Sums:** The distribution of the sum of the dice should follow a normal distribution. The Chi-squared test will be used to test for this distribution.
  - **Independence:**
    - **Horizontal:** The results of any die should be independent of the results of any other die for the same roll. A Chi-squared test using an orthogonal array strength two will be used to test for significant dependency.
    - **Vertical:** The outcome of a die from one roll should not influence the outcome of a die from the next roll. Chi-squared test of independence on the sequential rolls will be used to test for significant dependency.
      - **Sums:** The outcome of the sums of one roll should not influence the outcome of another roll. A Chi-squared test will be used to test for significant dependency.

Actual Output: Number of times each number was rolled:
1: 4317882
2: 4318930
3: 4318962
4: 4321903
5: 4319011
6: 4323312

Critical value (df=5, alpha=0.05): 11.07 Result: 5.15675 Pass

Sum distribution:
4: 4917
5: 19915
6: 50034
7: 100427
8: 173936
9: 279840
10: 399552
11: 519796
12: 625218
13: 699774
14: 729813
15: 700201
16: 625068
17: 520795
18: 400502
19: 279326
20: 175369
21: 100341
22: 50081
23: 20104
24: 4991

Critical value (df=20, alpha=0.05): 33.92 Result: 17.0868 Pass

Horizontal Transition Counts (for independence test):

	1	2	3	4	5	6
1	718867	720567	719826	719593	719924	719105
2	719555	718250	719618	720304	720572	720630
3	720053	719798	718908	720682	720056	719465
4	719291	721414	720128	720711	719005	721354
5	719868	718355	719994	720450	718886	721457
6	720247	720546	720487	720163	720567	721300

Critical value (df=25, alpha=0.05): 37.65 Result: 22.4572 Pass

Vertical Transition Counts (for independence test):

	1	2	3	4	5	6
1	720350	718457	719174	720132	718731	721038
2	719686	719054	720623	720031	719585	719951
3	719858	719540	719410	719933	720425	719796
4	718765	720682	719253	720532	721108	721563
5	719610	721144	719455	721255	718317	719230
6	719613	720053	721047	720020	720845	721734

Critical value (df=25, alpha=0.05): 37.65 Result: 26.4712 Pass

Sum Transition Counts (for independence test):

	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
4	7	14	44	76	120	217	329	382	486	505	535	538	500	362	325	200	144	77	32	16	8
5	20	55	189	318	524	820	1183	1617	1908	2113	2193	2203	1919	1575	1271	899	555	312	163	63	15
6	37	148	377	793	1395	2073	3112	3989	4913	5372	5592	5457	4723	4074	3066	2226	1302	787	393	169	36
7	88	316	807	1551	2823	4280	6178	7978	9619	10949	11311	10924	9631	8232	6101	4302	2645	1540	757	306	89
8	113	513	1383	2667	4729	7472	10685	14190	16841	18739	19521	18655	16773	13978	10731	7580	4652	2670	1362	534	148
9	227	849	2150	4429	7541	12242	17404	22445	26721	30168	31377	30286	26938	22600	17271	12111	7427	4385	2173	889	207
10	327	1234	3108	6221	10508	17225	24628	32210	38831	42855	45310	42779	38457	32039	24838	17216	10973	6199	3095	1175	324
11	386	1589	4067	8124	13859	22367	32265	41720	50099	55934	58494	56430	50114	41641	32099	22281	14129	8136	4074	1610	378
12	494	1982	4705	9686	16578	27390	38374	50302	60556	67601	70049	67672	60093	50452	38790	26732	16893	9665	4794	1930	480
13	522	2166	5406	10819	18618	30302	42997	56173	67412	76092	78612	75764	67749	56102	43070	30204	19011	10710	5341	2174	530
14	538	2208	5588	11203	19837	31457	44742	58149	70503	79245	82202	78656	70225	58924	45098	31509	19877	11374	5738	2175	565
15	542	2203	5492	10759	18801	30402	43507	56132	67685	75412	78968	75936	67075	56151	43180	30141	18855	10857	5311	2254	538
16	496	1834	4715	9727	16847	26690	38682	50111	60105	67485	70891	67433	60363	50201	38632	27026	16837	9728	4874	1908	483
17	356	1572	3969	8062	14162	22522	31818	41643	50297	56352	58445	56366	50401	41809	32039	22588	14178	8074	4047	1684	411
18	268	1241	3132	6245	10608	17321	24770	32178	38479	43045	45513	42959	38973	31993	24763	17142	10953	6232	3129	1252	306
19	217	858	2139	4404	7471	12086	17222	22518	26947	30108	31453	30002	27109	22488	17412	12050	7453	4184	2138	869	197
20	132	587	1376	2632	4737	7638	10847	14117	16829	18916	19687	18812	17058	14217	10733	7557	4716	2750	1330	546	152
21	85	317	786	1558	2731	4184	6129	7966	9799	10753	11277	11088	9815	7786	6339	4333	2732	1505	749	327	82
22	39	168	382	758	1355	2104	3144	3961	4700	5444	5610	5497	4763	4176	3150	2163	1353	776	379	132	27
23	17	49	159	307	551	822	1240	1639	1985	2158	2212	2204	1904	1616	1267	867	549	305	166	74	13
24	6	12	60	88	141	226	296	376	503	528	561	539	485	379	327	199	135	75	36	17	2

Critical value (df=400, alpha=0.01): 163.65 Result: 411.805 Fail

Lag-1 autocorrelation of sums (0.000786): 0.000347751

Permutation empirical p-value for chi-square: 0.311688

Test Case 5: Five Dice Distribution and Independence

- **Inputs:** Roll five dice 7.776 million times.
- **Expected Outputs:**
  - **Distribution:**
    - **Faces:** Each face (1, 2, 3, 4, 5, 6) is expected to appear approximately 6,480,000 times (7,776,000/6x5). The Chi-squared test will be used to test for a uniform distribution.
    - **Sums:** The distribution of the sum of the dice should follow a normal distribution. The Chi-squared test will be used to test for this distribution.
  - **Independence:**
    - **Horizontal:** The results of any die should be independent of the results of any other die for the same roll. A Chi-squared test using an orthogonal array strength two will be used to test for significant dependency.
    - **Vertical:** The outcome of a die from one roll should not influence the outcome of a die from the next roll. Chi-squared test of independence on the sequential rolls will be used to test for significant dependency.
      - **Sums:** The outcome of the sums of one roll should not influence the outcome of another roll. A Chi-squared test will be used to test for significant dependency.

Actual Output: Number of times each number was rolled:

1: 6479348
2: 6482753
3: 6478801
4: 6477983
5: 6481762
6: 6479353

Critical value (df=5, alpha=0.05): 11.07 Result: 2.62859 Pass

Sum distribution:

5: 1060
6: 4948
7: 14989
8: 34967
9: 70108
10: 126174
11: 204113
12: 305431
13: 420579
14: 538996
15: 651972
16: 734553
17: 779812
18: 779783
19: 735995
20: 651279
21: 540178
22: 419091
23: 305413
24: 204919
25: 125878
26: 69889
27: 34769
28: 15079

29: 5023
30: 1002

Critical value (df=25, alpha=0.05): 41.64 Result: 19.956 Pass

Horizontal Transition Counts (for independence test):

	0	1	2	3	4	5
0	1080240	1081803	1078448	1078579	1080734	1079544
1	1079612	1081379	1080669	1080604	1080700	1079788
2	1078908	1079909	1080000	1080184	1079856	1079943
3	1080141	1080514	1079200	1078225	1079339	1080564
4	1080819	1079588	1080817	1080947	1079663	1079927
5	1079626	1079558	1079667	1079443	1081470	1079587

Critical value (df=25, alpha=0.05): 37.65 Result: 16.58 Pass

Vertical Transition Counts (for independence test):

	0	1	2	3	4	5
0	1079536	1080939	1079502	1079700	1080223	1079448
1	1081272	1080165	1080010	1080631	1080569	1080106
2	1079029	1078921	1080147	1079881	1081074	1079749
3	1079594	1079334	1080370	1078699	1079849	1080137
4	1079310	1082512	1079394	1079616	1079635	1081295
5	1080607	1080882	1079378	1079456	1080412	1078618

Critical value (df=25, alpha=0.05): 37.65 Result: 16.0609 Pass

Sum Transition Counts (for independence test):

	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
5	0	1	3	3	7	18	40	41	59	86	107	101	92	103	102	75	75	48	35	35	12	10	3	3	1	0
6	0	5	9	24	39	73	134	224	277	324	376	473	458	526	448	440	362	250	197	127	103	43	22	10	0	4
7	4	16	21	61	143	237	408	584	848	1005	1260	1423	1570	1542	1397	1220	1014	805	603	371	232	138	58	20	8	1
8	4	25	64	167	330	571	938	1393	1903	2406	2889	3341	3555	3482	3242	2939	2425	1884	1382	899	556	307	167	74	18	6
9	11	45	126	299	626	1091	1816	2737	3857	4723	6017	6577	7028	7071	6716	5853	4801	3817	2723	1880	1148	642	319	133	42	10
10	17	79	228	578	1167	2017	3349	5018	6845	8745	10707	11910	12631	12644	11934	10589	8779	6724	5051	3218	2019	1091	515	227	75	17
11	28	122	362	929	1846	3224	5379	8017	11099	14120	17117	19202	20571	20663	19382	17228	14070	10987	8038	5314	3157	1842	881	358	148	29
12	41	201	586	1395	2728	5040	8007	12096	16684	21389	25402	28633	30532	30601	29122	25438	21379	16441	11998	7983	4828	2677	1352	631	220	27
13	51	276	846	1944	3766	6849	10896	16389	22923	29015	35305	39882	42024	42300	39577	35257	29387	22613	16719	11047	6687	3765	1890	854	251	66
14	80	336	1034	2409	4971	8715	14090	21039	28966	37251	44796	50933	54466	54144	51218	45281	37241	29165	21112	14242	8749	4839	2425	1052	374	68
15	90	413	1248	2887	5880	10459	17214	25737	35277	44954	55066	61633	65462	65026	61969	54471	45062	35397	25495	17078	10644	5883	2910	1224	410	83
16	105	470	1445	3199	6551	11887	19120	28888	39662	51094	61609	69360	73997	73851	69200	61384	51185	39788	28848	19204	11978	6510	3275	1400	458	85
17	103	502	1484	3582	7043	12898	20573	30664	42140	54290	65270	73491	77665	77771	73918	65142	54371	42142	30601	20671	12660	7161	3514	1543	508	105
18	105	480	1549	3569	7042	12722	20277	30597	41871	54109	65628	74225	78151	77617	73794	65507	54429	41897	30674	20501	12582	6957	3458	1469	456	117
19	97	445	1410	3171	6612	11907	19276	28663	39991	51147	61522	69549	73576	74046	69487	61989	51020	39748	28816	19731	11849	6609	3338	1410	488	98
20	88	384	1236	2925	5903	10530	17337	25785	35151	45123	54487	61145	65473	65472	61881	54379	45018	35112	25451	17180	10612	5858	2942	1290	442	75
21	65	378	1011	2470	4823	8826	14287	21199	29251	37354	45106	51125	53997	54140	51100	45404	37648	28904	21191	14151	8911	4889	2463	1058	356	71
22	56	244	790	1914	3710	6838	11065	16511	22562	29023	35290	39552	42076	42198	39741	35054	29219	22466	16302	10925	6804	3797	1810	836	251	56
23	42	218	595	1365	2807	4972	8123	11970	16633	21155	25636	28855	30669	30388	28726	25530	21191	16439	12088	8173	4851	2750	1380	605	214	38
24	29	133	389	907	1859	3341	5226	7992	10996	14256	17214	19424	20520	20711	19347	17054	14269	11033	8175	5471	3320	1796	917	391	129	20
25	22	87	275	618	1123	1988	3336	4931	6704	8839	10643	11893	12637	12766	11905	10439	8551	6691	4972	3354	2050	1165	550	249	79	11
26	16	45	151	293	621	1106	1766	2774	3859	4782	5796	6505	7108	7092	6481	5995	4770	3772	2795	1838	1162	651	327	124	54	6
27	5	32	77	148	327	551	900	1318	1886	2353	2934	3354	3524	3472	3282	2810	2422	1866	1358	959	599	326	172	66	26	2
28	1	8	37	73	133	222	392	615	790	1034	1290	1415	1453	1570	1450	1295	1069	773	577	390	254	130	54	36	12	6
29	0	2	12	33	36	73	134	210	284	346	422	473	485	491	473	422	363	269	172	151	89	43	22	15	2	1
30	0	1	1	4	15	18	30	39	61	73	83	79	92	96	103	84	58	60	40	26	22	10	5	1	1	0

Critical value (df=625, alpha=0.01): 486.36 Result: 589.15 Fail

Lag-1 autocorrelation of sums (0.000717): 0.000374124

Permutation empirical p-value for chi-square: 1

Top 20 cell contributions 2-dice (contrib, row(prevSum), col(currSum), obs, exp, resid):
0: 10.1163 (6,10) obs=42301 exp=41651.9 resid=3.18062
1: 5.41729 (2,2) obs=2660 exp=2782.78 resid=-2.32751
2: 4.51423 (3,6) obs=28086 exp=27732.2 resid=2.12467
3: 3.71796 (9,9) obs=45003 exp=44595.8 resid=1.9282
4: 3.31543 (8,12) obs=13635 exp=13849.3 resid=-1.82083
5: 3.11501 (10,11) obs=16925 exp=16696.9 resid=1.76494
6: 2.9079 (6,7) obs=82780 exp=83272.1 resid=-1.70526
7: 2.80029 (7,8) obs=83765 exp=83282.1 resid=1.67341
8: 2.70808 (8,10) obs=41321 exp=41656.9 resid=-1.64563
9: 2.15829 (3,7) obs=32993 exp=33260.9 resid=-1.46911
10: 2.0304 (4,7) obs=50273 exp=49954.5 resid=1.42492
11: 1.9452 (9,8) obs=55322 exp=55651 resid=-1.3947
12: 1.84065 (4,8) obs=41379 exp=41655.9 resid=-1.3567
13: 1.83476 (11,12) obs=5652 exp=5551.08 resid=1.35453
14: 1.76707 (7,12) obs=16437 exp=16608.3 resid=-1.32931
15: 1.68953 (3,9) obs=22032 exp=22225.8 resid=-1.29982
16: 1.6694 (6,11) obs=28048 exp=27832.4 resid=1.29205
17: 1.59666 (6,2) obs=13751 exp=13900 resid=-1.26359
18: 1.58104 (4,10) obs=24788 exp=24986.8 resid=-1.25739
19: 1.56198 (8,2) obs=14049 exp=13901.6 resid=1.24979

Top 20 cell contributions 3-dice (contrib, row(prevSum), col(currSum), obs, exp, resid):
0: 8.2728 (17,8) obs=5611 exp=5830.63 resid=-2.87625
1: 7.31532 (14,13) obs=29572 exp=29110.5 resid=2.70469
2: 6.54191 (11,14) obs=37949 exp=37454 resid=2.55771
3: 6.16194 (11,9) obs=63215 exp=62594 resid=2.48233
4: 6.09483 (6,13) obs=19190 exp=19535.1 resid=-2.46877
5: 5.10757 (17,3) obs=312 exp=274.553 resid=2.25999
6: 5.06919 (8,12) obs=49187 exp=48690.2 resid=2.25149
7: 4.9008 (3,9) obs=2396 exp=2290.06 resid=2.21377
8: 4.79082 (14,15) obs=13576 exp=13833.4 resid=-2.18879
9: 4.70996 (13,3) obs=2019 exp=1923.81 resid=2.17024
10: 4.3133 (5,18) obs=511 exp=560.154 resid=-2.07685
11: 4.1703 (10,11) obs=68008 exp=67477.5 resid=2.04213
12: 3.75244 (11,18) obs=2430 exp=2527.39 resid=-1.93712
13: 3.65434 (8,11) obs=52127 exp=52565.3 resid=-1.91163
14: 3.5755 (6,18) obs=881 exp=938.941 resid=-1.8909
15: 3.53257 (10,4) obs=7301 exp=7463.37 resid=-1.87951
16: 3.46555 (15,11) obs=25282 exp=24987.7 resid=1.8616
17: 3.39216 (13,18) obs=2046 exp=1964.37 resid=1.84178
18: 3.36634 (11,11) obs=67177 exp=67654.2 resid=-1.83476
19: 2.94493 (11,10) obs=67032 exp=67477.8 resid=-1.71608

Top 20 cell contributions 4-dice (contrib, row(prevSum), col(currSum), obs, exp, resid):
0: 11.9537 (24,6) obs=60 exp=38.537 resid=3.45742
1: 9.60872 (21,17) obs=7786 exp=8064.37 resid=-3.09979
2: 8.07167 (5,6) obs=189 exp=153.77 resid=2.84107
3: 6.01545 (7,8) obs=2823 exp=2695.66 resid=2.45264
4: 5.66557 (22,17) obs=4176 exp=4024.99 resid=2.38025
5: 5.6287 (12,9) obs=27390 exp=27000.2 resid=2.37249
6: 5.56309 (21,15) obs=11088 exp=10842.4 resid=2.35862
7: 5.1403 (21,9) obs=4184 exp=4333.25 resid=-2.26722
8: 4.68639 (4,24) obs=8 exp=3.78715 resid=2.16481
9: 4.61414 (19,21) obs=4184 exp=4325.27 resid=-2.14806
10: 4.38124 (10,8) obs=10508 exp=10724.8 resid=-2.09314
11: 4.28164 (20,5) obs=587 exp=538.962 resid=2.06921
12: 4.24079 (18,4) obs=268 exp=303.899 resid=-2.05932
13: 4.04776 (8,11) obs=14190 exp=13952.4 resid=2.01191
14: 3.9422 (16,5) obs=1834 exp=1921.02 resid=-1.9855
15: 3.88399 (17,4) obs=356 exp=395.177 resid=-1.97079
16: 3.65921 (18,14) obs=45513 exp=45106.7 resid=1.91291
17: 3.62687 (13,13) obs=76092 exp=75568.5 resid=1.90443
18: 3.61059 (10,15) obs=42779 exp=43173.8 resid=-1.90015
19: 3.60752 (22,12) obs=4700 exp=4832.03 resid=-1.89935

Top 20 cell contributions 5-dice (contrib, row(prevSum), col(currSum), obs, exp, resid):
0: 17.7321 (6,30) obs=4 exp=0.63759 resid=4.21095
1: 8.47062 (28,30) obs=6 exp=1.94305 resid=2.91043
2: 6.55633 (11,25) obs=3157 exp=3304.18 resid=-2.56053
3: 6.54805 (6,25) obs=103 exp=80.0983 resid=2.55892
4: 5.80527 (19,8) obs=3171 exp=3309.61 resid=-2.40941
5: 5.80355 (19,24) obs=19731 exp=19395.5 resid=2.40906
6: 5.32826 (5,11) obs=40 exp=27.824 resid=2.3083
7: 4.80017 (29,8) obs=33 exp=22.5874 resid=2.19093
8: 4.76848 (25,8) obs=618 exp=566.046 resid=2.18368
9: 4.73637 (17,10) obs=12898 exp=12653.2 resid=2.17632
10: 4.5231 (6,12) obs=224 exp=194.351 resid=2.12676
11: 4.51899 (18,29) obs=456 exp=503.71 resid=-2.12579
12: 4.40846 (27,6) obs=32 exp=22.1241 resid=2.09963
13: 4.39791 (26,5) obs=16 exp=9.52705 resid=2.09712
14: 4.37845 (7,6) obs=16 exp=9.53775 resid=2.09248
15: 4.34856 (24,11) obs=5226 exp=5378.94 resid=-2.08532
16: 4.31511 (25,7) obs=275 exp=242.642 resid=2.07729
17: 4.31032 (18,16) obs=74225 exp=73661.5 resid=2.07613
18: 4.30525 (18,18) obs=77617 exp=78197.2 resid=-2.07491
19: 4.2845 (10,27) obs=515 exp=564.165 resid=-2.0699

```
// main.cpp
// This file simulates rolling dice, analyzes the results for fairness and independence, and prints detailed statistics.
#include <iostream>
#include <thread>
#include <chrono>
#include <vector>
#include <iomanip>
#include <random>
#include <algorithm>
#include <tuple>
#include <future>
#include <cmath>

#include "DiceRoller.h"

using namespace std;

double chiSquareDist(int observed[], int totalRolls) {
// Checks if each die face (1-6) appears about as often as expected (fairness test).
    double expected = totalRolls / 6.0;
    double chiSquared = 0.0;
    for(int i = 0; i < 6; ++i) {
        double diff = observed[i] - expected;
        chiSquared += (diff * diff) / expected;
    }
    return chiSquared;
}

// Helper to count ways to get each sum for n dice
std::vector<int> diceSumWays(int numDice) {
// Calculates how many ways you can get each possible sum when rolling numDice dice.
    int minSum = numDice;
    int maxSum = numDice * 6;
    std::vector<int> ways(maxSum + 1, 0);
    // Dynamic programming: ways[s][d] = ways to get sum s with d dice
    std::vector<std::vector<int>> dp(numDice + 1, std::vector<int>(maxSum + 1, 0));
    dp[0][0] = 1;
    for (int d = 1; d <= numDice; ++d) {
        for (int s = 0; s <= maxSum; ++s) {
            for (int face = 1; face <= 6; ++face) {
                if (s - face >= 0)
                    dp[d][s] += dp[d-1][s-face];
            }
        }
    }
    for (int s = minSum; s <= maxSum; ++s) {
        ways[s] = dp[numDice][s];
    }
    return ways;
}

// Chi-squared test for dice sums (non-uniform expected)
double chiSquareSum(const std::vector<int>& observed, int numDice, int numRolls) {
// Checks if the distribution of dice sums matches what you'd expect from random dice.
```

```
    int minSum = numDice;
    int maxSum = numDice * 6;
    std::vector<double> expectedCounts(maxSum + 1, 0.0);
    std::vector<int> ways = diceSumWays(numDice);
    int totalWays = 0;
    for (int s = minSum; s <= maxSum; ++s) totalWays += ways[s];
    for (int s = minSum; s <= maxSum; ++s) {
        expectedCounts[s] = static_cast<double>(ways[s]) / totalWays * numRolls;
    }
    double chiSquared = 0.0;
    for (int s = minSum; s <= maxSum; ++s) {
        double expected = expectedCounts[s];
        double obs = observed[s];
        if (expected > 0) {
            double diff = obs - expected;
            chiSquared += (diff * diff) / expected;
        }
    }
    return chiSquared;
}

double chiSquareInd(const vector<vector<int>>& observed, int totalTransitions) {
// Tests if the transitions between die faces (from one roll to the next) are independent.
    vector<int> rowTotals(6, 0);
    vector<int> colTotals(6, 0);
    int grandTotal = 0;
    for (int i = 0; i < 6; ++i) {
        for (int j = 0; j < 6; ++j) {
            rowTotals[i] += observed[i][j];
            colTotals[j] += observed[i][j];
            grandTotal += observed[i][j];
        }
    }

    double chiSquared = 0.0;
    for (int i = 0; i < 6; ++i) {
        for (int j = 0; j < 6; ++j) {
            double expected_ij = (double)rowTotals[i] * colTotals[j] / grandTotal;
            if (expected_ij > 0) {
                double diff = observed[i][j] - expected_ij;
                chiSquared += (diff * diff) / expected_ij;
            }
        }
    }
    return chiSquared;
}

// double chiSquareIndSum(const std::vector<std::vector<int>>& transMatrix) {
// // Tests if the transitions between sums (from one roll to the next) are independent.
//     int n = transMatrix.size();
//     std::vector<int> rowTotals(n, 0);
//     std::vector<int> colTotals(n, 0);
//     int grandTotal = 0;
```

```
//     // Calculate row, column, and grand totals
//     for (int i = 0; i < n; ++i) {
//         for (int j = 0; j < n; ++j) {
//             rowTotals[i] += transMatrix[i][j];
//             colTotals[j] += transMatrix[i][j];
//             grandTotal += transMatrix[i][j];
//         }
//     }

//     double chiSquared = 0.0;
//     for (int i = 0; i < n; ++i) {
//         for (int j = 0; j < n; ++j) {
//             double expected = (double)rowTotals[i] * colTotals[j] / grandTotal;
//             if (expected > 0) {
//                 double diff = transMatrix[i][j] - expected;
//                 chiSquared += (diff * diff) / expected;
//             }
//         }
//     }
//     return chiSquared;
// }

// return contributors list: tuple(contribution, i, j, observed, expected, residual)
std::vector<std::tuple<double,int,int,int,double,double>>
chiSquareIndSumWithDiagnostics(const std::vector<std::vector<int>>& transMatrix,
                               double &chiOut) {
// Like chiSquareIndSum, but also records which transitions contribute most to the test statistic.
    int n = (int)transMatrix.size();
    std::vector<int> rowTotals(n, 0);
    std::vector<int> colTotals(n, 0);
    int grandTotal = 0;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) {
            rowTotals[i] += transMatrix[i][j];
            colTotals[j] += transMatrix[i][j];
            grandTotal += transMatrix[i][j];
        }

    chiOut = 0.0;
    std::vector<std::tuple<double,int,int,int,double,double>> contribs;
    contribs.reserve(static_cast<size_t>(n) * static_cast<size_t>(n));
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            double expected = 0.0;
            if (grandTotal > 0)
                expected = (double)rowTotals[i] * (double)colTotals[j] / (double)grandTotal;
            if (expected > 0.0) {
                double diff = transMatrix[i][j] - expected;
                double c = (diff*diff) / expected;
                double resid = diff / sqrt(expected);
                chiOut += c;
                contribs.emplace_back(c, i, j, transMatrix[i][j], expected, resid);
            }
        }
```

```
    }
    // sort contributions descending
    std::sort(contribs.begin(), contribs.end(),
            [](auto &a, auto &b){ return std::get<0>(a) > std::get<0>(b); });
    return contribs;
}

// compute lag-1 autocorrelation of sums vector
double lag1_autocorr(const std::vector<int>& sums) {
// Measures how much each sum is correlated with the previous sum (lag-1 autocorrelation).
    if (sums.size() < 2) return 0.0;
    double n = (double)sums.size();
    double mean = 0.0;
    for (int v : sums) mean += v;
    mean /= n;
    double num = 0.0, den = 0.0;
    for (size_t i = 0; i+1 < sums.size(); ++i)
        num += (sums[i] - mean) * (sums[i+1] - mean);
    for (size_t i = 0; i < sums.size(); ++i)
        den += (sums[i] - mean) * (sums[i] - mean);
    if (den == 0.0) return 0.0;
    return num / den;
}

// This version was a test with minor changes to reduce memory usage and improve speed
// permutation test: permute prior sums and recompute chi-square many times
// double permutation_pvalue_chi_parallel2(const std::vector<int>& prevSums, const std::vector<int>& currSums, int
maxSum, int nperms, double observedChi, mt19937_64& rng, int nthreads=0) {
//     // Shuffles the previous sums many times to see how often you get a chi-squared value as extreme as observed
(permutation test for independence).
//     // build original contingency as counts

//     const size_t N = prevSums.size();
//     if (N == 0 || nperms <= 0) return 1.0;
//     // Determine actual sum range [base..maxVal]; base should equal numDice in your setup
//     int base = std::numeric_limits<int>::max();
//     int maxVal = std::numeric_limits<int>::min();
//     for (int v : prevSums) { base = std::min(base, v); maxVal = std::max(maxVal, v); }
//     for (int v : currSums) { base = std::min(base, v); maxVal = std::max(maxVal, v); }
//     // Fall back to provided maxSum if needed
//     maxVal = std::max(maxVal, maxSum);
//     int S = maxVal - base + 1;

//     // Precompute row/col totals once (constant across permutations)
//     std::vector<int> rowCounts(S, 0), colCounts(S, 0);
//     for (int v : prevSums) rowCounts[v - base]++;
//     for (int v : currSums) colCounts[v - base]++;

//     // Decide thread count and partition work
//     unsigned hw = std::thread::hardware_concurrency();
//     int T = nthreads > 0 ? nthreads : (hw ? (int)hw : 2);
//     T = std::max(1, std::min(T, nperms));
//     std::vector<int> chunk(T, nperms / T);
//     for (int r = 0; r < nperms % T; ++r) ++chunk[r];
```

```
//     // Derive per-thread seeds from the provided rng (single-threaded draw)
//     std::vector<uint64_t> seeds(T);
//     for (int t = 0; t < T; ++t) seeds[t] = rng();

//     std::vector<int> ge_counts(T, 0);
//     std::vector<std::thread> threads;
//     threads.reserve(T);

//     for (int ti = 0; ti < T; ++ti) {
//         threads.emplace_back([&, ti]{
//             std::mt19937_64 trng(seeds[ti]);
//             // Local working buffers
//             std::vector<int> prior = prevSums;
//             std::vector<std::vector<int>> Tp(S, std::vector<int>(S, 0));

//             int ge = 0;
//             for (int p = 0; p < chunk[ti]; ++p) {
//                 // Reset
//                 for (int i = 0; i < S; ++i) {
//                     std::fill(Tp[i].begin(), Tp[i].end(), 0);
//                 }
//                 // Permute and build contingency in compact index space
//                 std::shuffle(prior.begin(), prior.end(), trng);
//                 for (size_t k = 0; k < N; ++k) {
//                     int r = prior[k] - base;
//                     int c = currSums[k] - base;
//                     Tp[r][c]++;
//                 }
//                 // Compute chi using precomputed marginals
//                 double chi = 0.0;
//                 for (int i = 0; i < S; ++i) {
//                     for (int j = 0; j < S; ++j) {
//                         double exp = (N > 0) ? (double)rowCounts[i] * colCounts[j] / (double)N : 0.0;
//                         if (exp > 0.0) {
//                             double d = Tp[i][j] - exp;
//                             chi += (d * d) / exp;
//                         }
//                     }
//                 }
//                 if (chi >= observedChi) ++ge;
//             }
//             ge_counts[ti] = ge;
//         });
//     }
//     for (auto& th : threads) th.join();

//     long long ge_total = 0;
//     for (int v : ge_counts) ge_total += v;
//     return (double)(ge_total + 1) / (double)(nperms + 1); // add-one correction
// }

// permutation test: permute prior sums and recompute chi-square many times
```

```
double permutation_pvalue_chi_parallel(const std::vector<int>& prevSums, const std::vector<int>& currSums, int maxSum, int nperms, double observedChi, mt19937_64& rng, int nthreads=0) {
   // Shuffles the previous sums many times to see how often you get a chi-squared value as extreme as observed (permutation test for independence).
   // build original contingency as counts

   const size_t N = prevSums.size();
   if (N == 0 || nperms <= 0) return 1.0;
   // Decide thread count
   unsigned hw = std::thread::hardware_concurrency();
   int T = nthreads > 0 ? nthreads : (hw ? (int)hw : 2);
   T = std::max(1, std::min(T, nperms));

   // Derive per-thread seeds from the provided rng (single-threaded) to avoid sharing rng
   std::vector<uint64_t> seeds(T);
   for (int t = 0; t < T; ++t) {
      seeds[t] = rng();
   }

   // Partition work across threads
   std::vector<int> chunk(T, nperms / T);
   for (int r = 0; r < nperms % T; ++r) ++chunk[r];

   std::vector<int> ge_counts(T, 0);
   std::vector<std::thread> threads;
   threads.reserve(T);

   for (int ti = 0; ti < T; ++ti) {
      threads.emplace_back([&, ti]{
         std::mt19937_64 trng(seeds[ti]);

         // Local working buffers to avoid contention
         std::vector<int> prior = prevSums;
         std::vector<std::vector<int>> Tp(maxSum + 1, std::vector<int>(maxSum + 1, 0));
         std::vector<int> rtot(maxSum + 1, 0), ctot(maxSum + 1, 0);

         int ge = 0;
         for (int p = 0; p < chunk[ti]; ++p) {
            // Reset
            for (int i = 0; i <= maxSum; ++i) {
               std::fill(Tp[i].begin(), Tp[i].end(), 0);
            }
            std::fill(rtot.begin(), rtot.end(), 0);
            std::fill(ctot.begin(), ctot.end(), 0);

            // Permute and build contingency
            std::shuffle(prior.begin(), prior.end(), trng);
            for (size_t k = 0; k < N; ++k) Tp[ prior[k] ][ currSums[k] ]++;

            // Compute chi-square
            double chi = 0.0;
            int grand = 0;
            for (int i = 0; i <= maxSum; ++i) for (int j = 0; j <= maxSum; ++j) {
               rtot[i] += Tp[i][j];
```

```
                ctot[j] += Tp[i][j];
                grand += Tp[i][j];
            }
            for (int i = 0; i <= maxSum; ++i) for (int j = 0; j <= maxSum; ++j) {
                double exp = (grand > 0) ? (double)rtot[i] * ctot[j] / (double)grand : 0.0;
                if (exp > 0.0) {
                    double d = Tp[i][j] - exp;
                    chi += (d * d) / exp;
                }
            }
            if (chi >= observedChi) ++ge;
        }
        ge_counts[ti] = ge;
    });
  }

  for (auto& th : threads) th.join();
  long long ge_total = 0;
  for (int v : ge_counts) ge_total += v;

  return (double)(ge_total + 1) / (double)(nperms + 1); // add-one correction
}

// Old slow single-threaded version
// double permutation_pvalue_chi(const std::vector<int>& prevSums, const std::vector<int>& currSums, int maxSum,
int nperms, double observedChi, mt19937_64& rng) {
//    std::vector<std::vector<int>> T(maxSum+1, std::vector<int>(maxSum+1,0));
//    size_t N = prevSums.size();
//    for (size_t k = 0; k < N; ++k) {
//        T[prevSums[k]][currSums[k]]++;
//    }

//    int ge = 0;
//    // make a vector of prior-sum values to shuffle
//    std::vector<int> prior = prevSums;
//    std::vector<std::vector<int>> Tp(maxSum+1, std::vector<int>(maxSum+1,0));
//    std::vector<int> rtot(maxSum+1,0), ctot(maxSum+1,0);
//    for (int p = 0; p < nperms; ++p) {
//        for (int i=0;i<=maxSum;++i) { std::fill(Tp[i].begin(), Tp[i].end(), 0); }
//        std::fill(rtot.begin(), rtot.end(), 0);
//        std::fill(ctot.begin(), ctot.end(), 0);
//        std::shuffle(prior.begin(), prior.end(), rng);
//        for (size_t k = 0; k < N; ++k) Tp[ prior[k] ][ currSums[k] ]++;
//        double chi = 0.0;
//        int grand=0;
//        for (int i=0;i<=maxSum;++i) for (int j=0;j<=maxSum;++j) {
//            rtot[i]+=Tp[i][j];
//            ctot[j]+=Tp[i][j];
//            grand+=Tp[i][j];
//        }
//        for (int i=0;i<=maxSum;++i) for (int j=0;j<=maxSum;++j) {
//            double exp = (grand>0) ? (double)rtot[i]*ctot[j]/(double)grand : 0.0;
//            if (exp>0.0) {
//                double d = Tp[i][j]-exp;
```

```
//                 chi += (d*d)/exp;
//             }
//         }
//         if (chi >= observedChi) ++ge;
//     }
//     return (double)(ge+1) / (double)(nperms+1); // add-one correction
// }

// Old slow single-threaded version with changes to reduce memory usage and improve speed
// double permutation_pvalue_chi(const std::vector<int>& prevSums, const std::vector<int>& currSums, int maxSum,
int nperms, double observedChi, mt19937_64& rng) {
//     // Determine actual sum range [base..maxVal]; base should equal numDice in your setup
//     int base = std::numeric_limits<int>::max();
//     int maxVal = std::numeric_limits<int>::min();
//     for (int v : prevSums) { base = std::min(base, v); maxVal = std::max(maxVal, v); }
//     for (int v : currSums) { base = std::min(base, v); maxVal = std::max(maxVal, v); }
//     // Fall back to provided maxSum if needed
//     maxVal = std::max(maxVal, maxSum);
//     int S = maxVal - base + 1;

//     // Precompute row/col totals once (constant across permutations)
//     std::vector<int> rowCounts(S, 0), colCounts(S, 0);
//     for (int v : prevSums) rowCounts[v - base]++;
//     for (int v : currSums) colCounts[v - base]++;

//     // Local buffers
//     std::vector<int> prior = prevSums;
//     std::vector<std::vector<int>> Tp(S, std::vector<int>(S, 0));

//     int ge = 0;
//     for (int p = 0; p < nperms; ++p) {
//         // Reset
//         for (int i = 0; i < S; ++i) {
//             std::fill(Tp[i].begin(), Tp[i].end(), 0);
//         }
//         std::shuffle(prior.begin(), prior.end(), rng);

//         // Build contingency in compact index space
//         for (size_t k = 0; k < N; ++k) {
//             int r = prior[k] - base;
//             int c = currSums[k] - base;
//             Tp[r][c]++;
//         }

//         // Compute chi using precomputed marginals (expected is constant per cell)
//         double chi = 0.0;
//         for (int i = 0; i < S; ++i) {
//             for (int j = 0; j < S; ++j) {
//                 double exp = (N > 0) ? (double)rowCounts[i] * colCounts[j] / (double)N : 0.0;
//                 if (exp > 0.0) {
//                     double d = Tp[i][j] - exp;
//                     chi += (d * d) / exp;
//                 }
//             }
```

```
//         }
//         if (chi >= observedChi) ++ge;
//     }

//     return (double)(ge + 1) / (double)(nperms + 1); // add-one correction
// }

struct RollTestResults {
// Holds all the results and statistics from a dice rolling experiment.
    std::vector<int> rollCount;
    double chiDist;
    std::vector<int> rollSum;
    double chiSums;
    std::vector<std::vector<int>> horizTrans;
    double chiHorInd;
    std::vector<std::vector<int>> vertTrans;
    double chiVertInd;
    std::vector<std::vector<int>> sumTrans;
    std::vector<int> prevSumsList;
    std::vector<int> currSumsList;
    double chiObserved;
    std::vector<std::tuple<double,int,int,int,double,double>> topContribs;
    double lag1Autocorr;
    double permPValue;
};

RollTestResults rollTest(int numRolls, int numDice){
// Simulates rolling numDice dice numRolls times, collects statistics, and runs all tests in parallel threads.
    using clock = std::chrono::high_resolution_clock;
    DiceRoller roller(numDice);
    std::vector<int> rollCount(6, 0);
    int minSum = numDice;
    int maxSum = numDice * 6;
    std::vector<int> rollSum(maxSum + 1, 0);
    std::vector<std::vector<int>> sumTrans(maxSum + 1, std::vector<int>(maxSum + 1, 0));
    std::vector<std::vector<int>> vertTrans(6, std::vector<int>(6, 0));
    std::vector<std::vector<int>> horizTrans(6, std::vector<int>(6, 0));
    std::vector<int> prevRoll(numDice, 0);
    //std::vector<std::vector<int>> results;
    std::vector<int> prevSumsList;
    std::vector<int> currSumsList;
    prevSumsList.reserve(std::max(0, numRolls-1));
    currSumsList.reserve(std::max(0, numRolls-1));

    for(int i = 0; i < numRolls; ++i) {
        vector<int> roll = roller.rollDice();
        //results.push_back(roll);
        int sum = 0;
        int prevSum = 0;

        // Sequential transitions within a roll
        for(int j = 0; j < numDice; ++j) {
            rollCount[roll[j] - 1]++;
            sum += roll[j];
```

```
            // Compare die j to die (j+1)%numDice
            int nextIdx = (j + 1) % numDice;
            int curr = roll[j];
            int next = roll[nextIdx];
            vertTrans[curr - 1][next - 1]++;
            if(i > 0) {
                int prev = prevRoll[j];
                int curr = roll[j];
                horizTrans[prev - 1][curr - 1]++;
                prevSum += prevRoll[j];
            }
        }
        if (sum >= minSum && sum <= maxSum) {
            rollSum[sum]++;
        }
        // Track transitions between sums
        if (i > 0 && prevSum >= minSum && prevSum <= maxSum && sum >= minSum && sum <= maxSum) {
            sumTrans[prevSum][sum]++;
            prevSumsList.push_back(prevSum);
            currSumsList.push_back(sum);
        }
        prevRoll = std::move(roll);
    }

    double chiDist = 0.0;
    double chiSums = 0.0;
    double chiHorInd = 0.0;
    double chiVertInd = 0.0;
    double chiObserved = 0.0;
    double lag1Autocorr = 0.0;
    double permPValue = 0.0;
    std::vector<std::tuple<double,int,int,int,double,double>> topContribs;
    // chiObserved = chiSquareIndSum(sumTrans);  Old version that didnt have diagnostics for my failing sum
transitions
    topContribs = chiSquareIndSumWithDiagnostics(sumTrans, chiObserved);
    topContribs.resize(std::min(20, (int)topContribs.size()));
    std::thread t1([&]{ permPValue = permutation_pvalue_chi_parallel(prevSumsList, currSumsList, maxSum, 1000,
chiObserved, roller.getEngine()); });
    lag1Autocorr = lag1_autocorr(currSumsList);
    chiDist = chiSquareDist(rollCount.data(), numRolls * numDice);
    chiHorInd = chiSquareInd(horizTrans, numRolls - 1);
    chiVertInd = chiSquareInd(vertTrans, numRolls - 1);
    chiSums = chiSquareSum(rollSum, numDice, numRolls);
    t1.join();

    return RollTestResults{
        rollCount,
        chiDist,
        rollSum,
        chiSums,
        horizTrans,
        chiHorInd,
        vertTrans,
        chiVertInd,
```

```
        sumTrans,
        prevSumsList,
        currSumsList,
        chiObserved,
        topContribs,
        lag1Autocorr,
        permPValue
    };
}

void printMatrix(const std::vector<std::vector<int>>& matrix, int minIdx, int maxIdx) {
// Nicely prints a matrix (like transition counts) for inspection.
    cout << "       ";
    for (int j = minIdx; j <= maxIdx; ++j) cout << setw(7) << j;
    cout << endl;
    for (int i = minIdx; i <= maxIdx; ++i) {
        cout << setw(7) << i << " |";
        for (int j = minIdx; j <= maxIdx; ++j) {
            cout << setw(7) << matrix[i][j];
        }
        cout << endl;
    }
}

void printRollTestResults(const RollTestResults& results, int numDice) {
// Prints all the results and statistics for a dice rolling experiment, including pass/fail for each test.
    const double chiCritSumsArr[4] = {18.31, 26.12, 33.92, 41.64}; // 2-5 dice
    const double chiCritSumIndArr[4] = {44.31, 95.02, 163.65, 486.36}; // 2-5 dice, alpha=0.01

    cout << "\nResults for " << numDice << " dice:" << endl;
    // Uniformity distribution
    cout << "Number of times each number was rolled:\n";
    for (size_t i = 0; i < results.rollCount.size(); ++i) {
        cout << (i + 1) << ": " << results.rollCount[i] << endl;
    }
    cout << "\nCritical value (df=5, alpha=0.05): 11.07\n";
    if (results.chiDist > 11.07)
        cout << "\nResult: " << results.chiDist << " Fail\n";
    else
        cout << "\nResult: " << results.chiDist << " Pass\n";

    // Sums distribution
    cout << "\nSum distribution:\n";
    int minSum = numDice;
    int maxSum = numDice * 6;
    for (int s = minSum; s <= maxSum; ++s) {
        cout << s << ": " << results.rollSum[s] << endl;
    }

    cout << "\nCritical value (df=" << (maxSum-minSum) << ", alpha=0.05): ";
    if (numDice >= 2 && numDice <= 5) {
        cout << chiCritSumsArr[numDice-2] << "\n";
        if (results.chiSums > chiCritSumsArr[numDice-2])
            cout << "\nResult: " << results.chiSums << " Fail\n";
```

```
        else
            cout << "\nResult: " << results.chiSums << " Pass\n";
    } else {
        cout << "[see chi-squared table]\n";
        cout << "\nResult: " << results.chiSums << " [no pass/fail]\n";
    }

    // Horizontal independence
    cout << "\nHorizontal Transition Counts (for independence test):\n";
    printMatrix(results.horizTrans, 0, 5);

    cout << "\nCritical value (df=25, alpha=0.05): 37.65" << endl;
    if (results.chiHorInd > 37.65)
        cout << "\nResult: " << results.chiHorInd << " Fail\n";
    else
        cout << "\nResult: " << results.chiHorInd << " Pass\n";

    // Vertical independence
    cout << "\nVertical Transition Counts (for independence test):\n";
    printMatrix(results.vertTrans, 0, 5);

    cout << "\nCritical value (df=25, alpha=0.05): 37.65" << endl;
    if (results.chiVertInd > 37.65)
        cout << "\nResult: " << results.chiVertInd << " Fail\n";
    else
        cout << "\nResult: " << results.chiVertInd << " Pass\n";

    // Sum independence
    cout << "\nSum Transition Counts (for independence test):\n";
    printMatrix(results.sumTrans, minSum, maxSum);

    // This function is modified to use the dianostics version
    cout << "\nCritical value (df=" << (maxSum-minSum)*(maxSum-minSum) << ", alpha=0.01): ";
    if (numDice >= 2 && numDice <= 5) {
        cout << chiCritSumIndArr[numDice-2] << "\n";
        if (results.chiObserved > chiCritSumIndArr[numDice-2])
            cout << "\nResult: " << results.chiObserved << " Fail\n";
        else
            cout << "\nResult: " << results.chiObserved << " Pass\n";
    } else {
        cout << "[see chi-squared table]\n";
        cout << "\nResult: " << results.chiObserved << " [no pass/fail]\n";
    }
    cout << "Lag-1 autocorrelation of sums: " << results.lag1Autocorr << endl;
    cout << "Permutation empirical p-value for chi-square: " << results.permPValue << endl;

    cout << "\nTop " << results.topContribs.size() << " cell contributions (contrib, row(prevSum), col(currSum), obs,
exp, resid):\n";
    for (int i = 0; i < results.topContribs.size(); ++i) {
        auto [c, r, col, obs, exp, resid] = results.topContribs[i];
        cout << i << ": " << c << "  (" << r << "," << col << ")  obs=" << obs << " exp=" << exp << " resid=" << resid <<
"\n";
    }
}
```

```
int main() {
    // Thread pool using std::async for parallel rollTest
    auto fut5 = std::async(std::launch::async, rollTest, 7776000, 5);
    auto fut4 = std::async(std::launch::async, rollTest, 6480000, 4);
    auto fut3 = std::async(std::launch::async, rollTest, 4320000, 3);
    auto fut2 = std::async(std::launch::async, rollTest, 3600000, 2);

    constexpr int numRolls = 3600000;
    int rollCount[6] = {0};
    vector<vector<int>> transCount(6, vector<int>(6, 0));
    DiceRoller roller(1);
    int prev = 0;

    // Batch random generation for single die
    for(int i = 0; i < numRolls; ++i) {
        int val = roller.rollOne();
        ++rollCount[val - 1];
        if(i > 0) {
            int curr = val;
            transCount[prev - 1][curr - 1]++;
        }
        prev = val;
    }

    double chiDist = 0.0;
    double chiInd = 0.0;
    auto futDist = std::async(std::launch::async, chiSquareDist, rollCount, numRolls);
    auto futInd = std::async(std::launch::async, chiSquareInd, transCount, numRolls - 1);

    cout << "Number of times each number was rolled:\n";
    for (int i = 0; i < 6; ++i) {
        cout << (i + 1) << ": " << rollCount[i] << '\n';
    }

    chiDist = futDist.get();
    cout << "\nCritical value (df=5, alpha=0.05): 11.07\n";
    if (__builtin_expect(chiDist > 11.07, 0)) // [[unlikely]]
        cout << "\nResult: " << chiDist << " Fail\n" << endl;
    else
        cout << "\nResult: " << chiDist << " Pass\n" << endl;

    printMatrix(transCount, 0, 5);

    chiInd = futInd.get();
    cout << "\nCritical value (df=25, alpha=0.05): 37.65" << endl;
    if (__builtin_expect(chiInd > 37.65, 0)) // [[unlikely]]
        cout << "\nResult: " << chiInd << " Fail\n";
    else
        cout << "\nResult: " << chiInd << " Pass\n";

    // Wait for all dice results
    printRollTestResults(fut2.get(), 2);
    printRollTestResults(fut3.get(), 3);
```

```
    printRollTestResults(fut4.get(), 4);
    printRollTestResults(fut5.get(), 5);

    return 0;
}

#pragma once
#include <cstdint>
#include <random>
#include <chrono>
#include <array>

// xoshiro256pp code retained for future use:
class xoshiro256pp {
public:
    using result_type = uint64_t;
    std::array<uint64_t, 4> s;
    explicit xoshiro256pp(const uint64_t seed1, const uint64_t seed2, const uint64_t seed3, const uint64_t seed4)
noexcept
        : s{seed1, seed2, seed3, seed4} {}
    static constexpr uint64_t rotl(const uint64_t x, const int k) noexcept {
        return (x << k) | (x >> (64 - k));
    }
    inline uint64_t operator()() noexcept {
        const uint64_t result = rotl(s[0] + s[3], 23) + s[0];
        const uint64_t t = s[1] << 17;
        s[2] ^= s[0];
        s[3] ^= s[1];
        s[1] ^= s[2];
        s[0] ^= s[3];
        s[2] ^= t;
        s[3] = rotl(s[3], 45);
        return result;
    }
    static constexpr uint64_t min() noexcept { return 0; }
    static constexpr uint64_t max() noexcept { return UINT64_MAX; }
};

class DiceRoller {
public:
    DiceRoller(int numDice) noexcept;
    inline std::vector<int> rollDice() noexcept {
        std::vector<int> results;
        results.reserve(engines.size());
        for (auto& eng : engines) {
            results.push_back(dist(eng));
        }
        return results;
    }
    inline int rollOne() noexcept {
        return dist(engines[0]);
    }

    // inline std::vector<int> rollDice(const int numDice) noexcept {
```

```
    //     static thread_local std::mt19937_64 rng(std::random_device{}());
    //     std::vector<int> results;
    //     results.reserve(numDice);
    //     for (int i = 0; i < numDice; ++i) {
    //         results.push_back(dist(rng));
    //     }
    //     return results;
    // }
    std::mt19937_64& getEngine(int idx = 0) noexcept { return engines[idx]; }
    static thread_local std::uniform_int_distribution<int> dist;
private:
    std::vector<std::mt19937_64> engines;
};

// DiceRoller.cpp
// This file provides random dice rolling using multiple sources of randomness for extra unpredictability.
#include <thread>
#include "DiceRoller.h"

using namespace std;

DiceRoller::DiceRoller(int numDice) noexcept {
    engines.reserve(numDice);
    // auto hash64 = [](uint64_t x) {
    //     x ^= (x >> 33);
    //     x *= 0xff51afd7ed558ccdULL;
    //     x ^= (x >> 33);
    //     x *= 0xc4ceb9fe1a85ec53ULL;
    //     x ^= (x >> 33);
    //     return x;
    // };
    for (int i = 0; i < numDice; ++i) {
        std::random_device rd;
        uint64_t seed = ((uint64_t)rd()) << 32 | rd();
        engines.emplace_back(std::mt19937_64(seed));
        // uint64_t rd_seed = ((uint64_t)rd()) << 32 | rd();
        // uint64_t time = std::chrono::duration_cast<std::chrono::milliseconds>(
        //     std::chrono::system_clock::now().time_since_epoch()).count();
        // uint64_t mixed = time ^ rd_seed ^ (i * 0x9e3779b97f4a7c15ULL);
        // uint64_t h1 = hash64(mixed);
        // uint64_t h2 = hash64(mixed + i);
        // uint64_t h3 = hash64(mixed ^ (i << 16));
        // uint64_t h4 = hash64(mixed * (i+1));
        // engines.emplace_back(h1, h2, h3, h4);
        // std::this_thread::sleep_for(std::chrono::microseconds(1100));
    }
}

thread_local std::uniform_int_distribution<int> DiceRoller::dist(1, 6);
```