Master Index

Local Index

MAGIC SQUARES

This algorithm simply checks the row and column totals for a list of 16 consecutive prime numbers that we know can be arranged to form a 4 x 4 semi-magic square. The smallest member of these groups of 16 primes is listed in OEIS A270865:

A270865	Smallest primes of 4 X 4 semi-magic squares formed from consecutive primes.

The initial members of the sequence are:

5, 19, 29, 31, 37, 47, 53, 79, 397, 409, 599, 787, 1229, 1381, 1439, 1993, 2087, 2767, 4003, 4159, 4931, 5791, 5981, 8117, 9293, 9349, 9833, 10939, 10979, 11213, 12553, 12907, 14557, 16361, 18047, 21089, 21557, 21577, 25903, 26339, 28439, 33547, 56813, 57667

Here is a SageMath algorithm generated by z.ai that will generate the magic squares associated with each of the above numbers:

```
def generate_4x4_magic_square(start_prime, verbose=False):
    """
    Generates a 4x4 magic square from 16 consecutive primes starting with start_prime.

    Args:
        start_prime (int): The starting prime
        verbose (bool): Whether to print detailed information
```

return to Local Index

```
    Returns:
        list: A 4x4 magic square as a list of lists, or None if no solution found
    """
    # Step 1: Generate 16 consecutive primes
    primes = [start_prime]
    p = start_prime
    while len(primes) < 16:
        p = next_prime(p)
        primes.append(p)

    if verbose:
        print(f"Primes: {primes}")

    # Step 2: Calculate magic constant
    total = sum(primes)
    if total % 4 != 0:
        if verbose:
            print(f"Total sum {total} is not divisible by 4")
        return None

    M = total // 4
    if verbose:
        print(f"Magic constant: {M}")

    # Step 3: Find all possible 4-element subsets that sum to M
    from itertools import combinations
    row_candidates = []
    for combo in combinations(primes, 4):
        if sum(combo) == M:
            row_candidates.append(set(combo))

    if verbose:
        print(f"Found {len(row_candidates)} potential rows")

    # Step 4: Find 4 disjoint rows that cover all primes
    from itertools import combinations, permutations

    # Try all combinations of 4 row candidates
    for rows in combinations(row_candidates, 4):
        # Check if rows are disjoint and cover all primes
        union = set().union(*rows)
        if union != set(primes):
            continue

        # Try all permutations of elements within rows
        row_perms = [list(permutations(row)) for row in rows]
```

return to Local Index

```
    # Try all combinations of row permutations
    for p0 in row_perms[0]:
      for p1 in row_perms[1]:
        for p2 in row_perms[2]:
          for p3 in row_perms[3]:
            # Check column sums
            valid = True
            for col in range(4):
              col_sum = p0[col] + p1[col] + p2[col] + p3[col]
              if col_sum != M:
                valid = False
                break
            if valid:
              magic_square = [list(p0), list(p1), list(p2), list(p3)]
              return magic_square

  return None

# Example usage with the provided primes
primes_to_test = [5, 19, 29, 31, 37, 47, 53, 79, 397, 409, 599, 787, 1229, 1381, 1439, 1993, 2087,
2767, 4003, 4159, 4931, 5791, 5981, 8117, 9293, 9349, 9833, 10939, 10979, 11213, 12553, 12907,
14557, 16361, 18047, 21089, 21557, 21577, 25903, 26339, 28439, 33547, 56813, 57667]

for prime in primes_to_test:
  print(f"\nTesting prime: {prime}")
  magic_square = generate_4x4_magic_square(prime, verbose=True)
  if magic_square:
    print("\nMagic Square Found:")
    for row in magic_square:
      print(row)
    print(f"Magic constant: {sum(magic_square[0])}")
  else:
    print("\nNo magic square found.")
```

```
Testing prime: 5
Primes: [5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61]
Magic constant: 124
Found 46 potential rows

Magic Square Found:
[59, 53, 5, 7]
[11, 23, 29, 61]
[13, 31, 43, 37]
[41, 17, 47, 19]
Magic constant: 124

Testing prime: 19
Primes: [19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83]
Magic constant: 204
Found 50 potential rows
```

return to Local Index

```
Magic Square Found:
[83, 19, 79, 23]
[31, 71, 29, 73]
[47, 53, 37, 67]
[43, 61, 59, 41]
Magic constant: 204

Testing prime: 29
Primes: [29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
Magic constant: 240
Found 51 potential rows

Magic Square Found:
[73, 41, 29, 97]
[31, 61, 89, 59]
[53, 71, 79, 37]
[83, 67, 43, 47]
Magic constant: 240

Testing prime: 31
Primes: [31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101]
Magic constant: 258
Found 46 potential rows

Magic Square Found:
[89, 41, 97, 31]
[59, 61, 37, 101]
[43, 83, 53, 79]
[67, 73, 71, 47]
Magic constant: 258

Testing prime: 37
Primes: [37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103]
Magic constant: 276
Found 46 potential rows

Magic Square Found:
[41, 101, 37, 97]
[103, 43, 83, 47]
[73, 61, 89, 53]
[59, 71, 67, 79]
Magic constant: 276

Testing prime: 47
Primes: [47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113]
Magic constant: 328
Found 36 potential rows

Magic Square Found:
[113, 59, 109, 47]
[53, 89, 79, 107]
[61, 97, 67, 103]
```

return to Local Index

```
[101, 83, 73, 71]
Magic constant: 328

Testing prime: 53
Primes: [53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127]
Magic constant: 348
Found 46 potential rows

Magic Square Found:
[107, 61, 53, 127]
[67, 113, 109, 59]
[101, 71, 97, 79]
[73, 103, 89, 83]
Magic constant: 348

Testing prime: 79
Primes: [79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157]
Magic constant: 468
Found 41 potential rows

Magic Square Found:
[151, 137, 101, 79]
[97, 139, 83, 149]
[113, 89, 157, 109]
[107, 103, 127, 131]
Magic constant: 468

Testing prime: 397
Primes: [397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479,
487]
Magic constant: 1764
Found 38 potential rows

Magic Square Found:
[401, 487, 397, 479]
[457, 409, 467, 431]
[463, 419, 461, 421]
[443, 449, 439, 433]
Magic constant: 1764

Testing prime: 409
Primes: [409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491,
499]
Magic constant: 1812
Found 35 potential rows

Magic Square Found:
[409, 443, 461, 499]
[467, 487, 439, 419]
[479, 449, 421, 463]
[457, 433, 491, 431]
Magic constant: 1812
```

return to Local Index

```
Testing prime: 599
Primes: [599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677,
683]
Magic constant: 2556
Found 37 potential rows

Magic Square Found:
[601, 673, 683, 599]
[647, 607, 641, 661]
[631, 659, 613, 653]
[677, 617, 619, 643]
Magic constant: 2556

Testing prime: 787
Primes: [787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881,
883]
Magic constant: 3354
Found 33 potential rows

Magic Square Found:
[809, 787, 877, 881]
[863, 883, 797, 811]
[853, 857, 821, 823]
[829, 827, 859, 839]
Magic constant: 3354

Testing prime: 1229
Primes: [1229, 1231, 1237, 1249, 1259, 1277, 1279, 1283, 1289, 1291, 1297, 1301, 1303,
1307, 1319, 1321]
Magic constant: 5118
Found 37 potential rows

Magic Square Found:
[1249, 1321, 1229, 1319]
[1307, 1231, 1297, 1283]
[1303, 1277, 1301, 1237]
[1259, 1289, 1291, 1279]
Magic constant: 5118

Testing prime: 1381
Primes: [1381, 1399, 1409, 1423, 1427, 1429, 1433, 1439, 1447, 1451, 1453, 1459, 1471,
1481, 1483, 1487]
Magic constant: 5768
Found 27 potential rows

Magic Square Found:
[1481, 1483, 1381, 1423]
[1451, 1399, 1471, 1447]
[1409, 1433, 1487, 1439]
[1427, 1453, 1429, 1459]
Magic constant: 5768

Testing prime: 1439
```

return to Local Index

```
Primes: [1439, 1447, 1451, 1453, 1459, 1471, 1481, 1483, 1487, 1489, 1493, 1499, 1511,
1523, 1531, 1543]
Magic constant: 5940
Found 33 potential rows

Magic Square Found:
[1447, 1511, 1543, 1439]
[1499, 1459, 1451, 1531]
[1523, 1481, 1453, 1483]
[1471, 1489, 1493, 1487]
Magic constant: 5940

Testing prime: 1993
Primes: [1993, 1997, 1999, 2003, 2011, 2017, 2027, 2029, 2039, 2053, 2063, 2069, 2081,
2083, 2087, 2089]
Magic constant: 8160
Found 30 potential rows

Magic Square Found:
[1993, 1997, 2089, 2081]
[2069, 2063, 2029, 1999]
[2087, 2017, 2003, 2053]
[2011, 2083, 2039, 2027]
Magic constant: 8160

Testing prime: 2087
Primes: [2087, 2089, 2099, 2111, 2113, 2129, 2131, 2137, 2141, 2143, 2153, 2161, 2179,
2203, 2207, 2213]
Magic constant: 8574
Found 25 potential rows

Magic Square Found:
[2153, 2131, 2203, 2087]
[2179, 2089, 2099, 2207]
[2113, 2213, 2111, 2137]
[2129, 2141, 2161, 2143]
Magic constant: 8574

Testing prime: 2767
Primes: [2767, 2777, 2789, 2791, 2797, 2801, 2803, 2819, 2833, 2837, 2843, 2851, 2857,
2861, 2879, 2887]
Magic constant: 11298
Found 29 potential rows

Magic Square Found:
[2791, 2879, 2861, 2767]
[2837, 2797, 2777, 2887]
[2851, 2789, 2857, 2801]
[2819, 2833, 2803, 2843]
Magic constant: 11298

Testing prime: 4003
```

return to Local Index

```
Primes: [4003, 4007, 4013, 4019, 4021, 4027, 4049, 4051, 4057, 4073, 4079, 4091, 4093,
4099, 4111, 4127]
Magic constant: 16230
Found 25 potential rows

Magic Square Found:
[4003, 4127, 4093, 4007]
[4099, 4027, 4013, 4091]
[4049, 4019, 4051, 4111]
[4079, 4057, 4073, 4021]
Magic constant: 16230

Testing prime: 4159
Primes: [4159, 4177, 4201, 4211, 4217, 4219, 4229, 4231, 4241, 4243, 4253, 4259, 4261,
4271, 4273, 4283]
Magic constant: 16932
Found 43 potential rows

Magic Square Found:
[4271, 4219, 4283, 4159]
[4243, 4253, 4177, 4259]
[4201, 4229, 4261, 4241]
[4217, 4231, 4211, 4273]
Magic constant: 16932

Testing prime: 4931
Primes: [4931, 4933, 4937, 4943, 4951, 4957, 4967, 4969, 4973, 4987, 4993, 4999, 5003,
5009, 5011, 5021]
Magic constant: 19896
Found 36 potential rows

Magic Square Found:
[4993, 4931, 5021, 4951]
[5009, 5011, 4933, 4943]
[4937, 4987, 4969, 5003]
[4957, 4967, 4973, 4999]
Magic constant: 19896

Testing prime: 5791
Primes: [5791, 5801, 5807, 5813, 5821, 5827, 5839, 5843, 5849, 5851, 5857, 5861, 5867,
5869, 5879, 5881]
Magic constant: 23364
Found 33 potential rows

Magic Square Found:
[5881, 5879, 5813, 5791]
[5801, 5827, 5869, 5867]
[5839, 5807, 5861, 5857]
[5843, 5851, 5821, 5849]
Magic constant: 23364

Testing prime: 5981
```

return to Local Index

```
Primes: [5981, 5987, 6007, 6011, 6029, 6037, 6043, 6047, 6053, 6067, 6073, 6079, 6089,
6091, 6101, 6113]
Magic constant: 24202
Found 20 potential rows

Magic Square Found:
[6113, 5981, 6029, 6079]
[5987, 6053, 6089, 6073]
[6091, 6067, 6037, 6007]
[6011, 6101, 6047, 6043]
Magic constant: 24202

Testing prime: 8117
Primes: [8117, 8123, 8147, 8161, 8167, 8171, 8179, 8191, 8209, 8219, 8221, 8231, 8233,
8237, 8243, 8263]
Magic constant: 32778
Found 25 potential rows

Magic Square Found:
[8219, 8179, 8117, 8263]
[8221, 8243, 8191, 8123]
[8167, 8147, 8233, 8231]
[8171, 8209, 8237, 8161]
Magic constant: 32778

Testing prime: 9293
Primes: [9293, 9311, 9319, 9323, 9337, 9341, 9343, 9349, 9371, 9377, 9391, 9397, 9403,
9413, 9419, 9421]
Magic constant: 37452
Found 30 potential rows

Magic Square Found:
[9337, 9403, 9293, 9419]
[9311, 9377, 9421, 9343]
[9413, 9323, 9397, 9319]
[9391, 9349, 9341, 9371]
Magic constant: 37452

Testing prime: 9349
Primes: [9349, 9371, 9377, 9391, 9397, 9403, 9413, 9419, 9421, 9431, 9433, 9437, 9439,
9461, 9463, 9467]
Magic constant: 37668
Found 35 potential rows

Magic Square Found:
[9461, 9467, 9349, 9391]
[9371, 9403, 9463, 9431]
[9439, 9377, 9419, 9433]
[9397, 9421, 9437, 9413]
Magic constant: 37668

Testing prime: 9833
```

return to Local Index

```
Primes: [9833, 9839, 9851, 9857, 9859, 9871, 9883, 9887, 9901, 9907, 9923, 9929, 9931,
9941, 9949, 9967]
Magic constant: 39582
Found 23 potential rows

Magic Square Found:
[9833, 9851, 9931, 9967]
[9949, 9923, 9839, 9871]
[9941, 9901, 9883, 9857]
[9859, 9907, 9929, 9887]
Magic constant: 39582

Testing prime: 10939
Primes: [10939, 10949, 10957, 10973, 10979, 10987, 10993, 11003, 11027, 11047, 11057,
11059, 11069, 11071, 11083, 11087]
Magic constant: 44070
Found 23 potential rows

Magic Square Found:
[10939, 10973, 11087, 11071]
[11059, 11083, 10979, 10949]
[11069, 10987, 10957, 11057]
[11003, 11027, 11047, 10993]
Magic constant: 44070

Testing prime: 10979
Primes: [10979, 10987, 10993, 11003, 11027, 11047, 11057, 11059, 11069, 11071, 11083,
11087, 11093, 11113, 11117, 11119]
Magic constant: 44226
Found 28 potential rows

Magic Square Found:
[11113, 10979, 11087, 11047]
[11003, 11119, 10987, 11117]
[11083, 11057, 11093, 10993]
[11027, 11071, 11059, 11069]
Magic constant: 44226

Testing prime: 11213
Primes: [11213, 11239, 11243, 11251, 11257, 11261, 11273, 11279, 11287, 11299, 11311,
11317, 11321, 11329, 11351, 11353]
Magic constant: 45146
Found 23 potential rows

Magic Square Found:
[11353, 11329, 11251, 11213]
[11239, 11287, 11299, 11321]
[11243, 11273, 11279, 11351]
[11311, 11257, 11317, 11261]
Magic constant: 45146

Testing prime: 12553
```

return to Local Index

```
Primes: [12553, 12569, 12577, 12583, 12589, 12601, 12611, 12613, 12619, 12637, 12641,
12647, 12653, 12659, 12671, 12689]
Magic constant: 50478
Found 31 potential rows

Magic Square Found:
[12553, 12689, 12653, 12583]
[12613, 12569, 12659, 12637]
[12671, 12619, 12577, 12611]
[12641, 12601, 12589, 12647]
Magic constant: 50478

Testing prime: 12907
Primes: [12907, 12911, 12917, 12919, 12923, 12941, 12953, 12959, 12967, 12973, 12979,
12983, 13001, 13003, 13007, 13009]
Magic constant: 51838
Found 24 potential rows

Magic Square Found:
[13009, 12907, 13003, 12919]
[12953, 13007, 12911, 12967]
[12917, 13001, 12941, 12979]
[12959, 12923, 12983, 12973]
Magic constant: 51838

Testing prime: 14557
Primes: [14557, 14561, 14563, 14591, 14593, 14621, 14627, 14629, 14633, 14639, 14653,
14657, 14669, 14683, 14699, 14713]
Magic constant: 58522
Found 22 potential rows

Magic Square Found:
[14683, 14629, 14557, 14653]
[14591, 14561, 14657, 14713]
[14621, 14699, 14639, 14563]
[14627, 14633, 14669, 14593]
Magic constant: 58522

Testing prime: 16361
Primes: [16361, 16363, 16369, 16381, 16411, 16417, 16421, 16427, 16433, 16447, 16451,
16453, 16477, 16481, 16487, 16493]
Magic constant: 65718
Found 26 potential rows

Magic Square Found:
[16361, 16493, 16411, 16453]
[16477, 16363, 16451, 16427]
[16447, 16481, 16369, 16421]
[16433, 16381, 16487, 16417]
Magic constant: 65718

Testing prime: 18047
```

return to Local Index

```
Primes: [18047, 18049, 18059, 18061, 18077, 18089, 18097, 18119, 18121, 18127, 18131,
18133, 18143, 18149, 18169, 18181]
Magic constant: 72438
Found 30 potential rows

Magic Square Found:
[18089, 18121, 18181, 18047]
[18119, 18143, 18049, 18127]
[18169, 18077, 18059, 18133]
[18061, 18097, 18149, 18131]
Magic constant: 72438

Testing prime: 21089
Primes: [21089, 21101, 21107, 21121, 21139, 21143, 21149, 21157, 21163, 21169, 21179,
21187, 21191, 21193, 21211, 21221]
Magic constant: 84630
Found 28 potential rows

Magic Square Found:
[21089, 21221, 21163, 21157]
[21179, 21101, 21211, 21139]
[21193, 21187, 21107, 21143]
[21169, 21121, 21149, 21191]
Magic constant: 84630

Testing prime: 21557
Primes: [21557, 21559, 21563, 21569, 21577, 21587, 21589, 21599, 21601, 21611, 21613,
21617, 21647, 21649, 21661, 21673]
Magic constant: 86418
Found 32 potential rows

Magic Square Found:
[21577, 21673, 21611, 21557]
[21563, 21559, 21649, 21647]
[21661, 21587, 21569, 21601]
[21617, 21599, 21589, 21613]
Magic constant: 86418

Testing prime: 21577
Primes: [21577, 21587, 21589, 21599, 21601, 21611, 21613, 21617, 21647, 21649, 21661,
21673, 21683, 21701, 21713, 21727]
Magic constant: 86562
Found 28 potential rows

Magic Square Found:
[21577, 21683, 21713, 21589]
[21727, 21649, 21587, 21599]
[21647, 21613, 21601, 21701]
[21611, 21617, 21661, 21673]
Magic constant: 86562

Testing prime: 25903
```

return to Local Index

```
Primes: [25903, 25913, 25919, 25931, 25933, 25939, 25943, 25951, 25969, 25981, 25997,
25999, 26003, 26017, 26021, 26029]
Magic constant: 103862
Found 24 potential rows

Magic Square Found:
[25999, 25931, 26029, 25903]
[25951, 25913, 25981, 26017]
[25943, 25997, 25919, 26003]
[25969, 26021, 25933, 25939]
Magic constant: 103862

Testing prime: 26339
Primes: [26339, 26347, 26357, 26371, 26387, 26393, 26399, 26407, 26417, 26423, 26431,
26437, 26449, 26459, 26479, 26489]
Magic constant: 105646
Found 24 potential rows

Magic Square Found:
[26387, 26489, 26339, 26431]
[26479, 26347, 26449, 26371]
[26357, 26393, 26459, 26437]
[26423, 26417, 26399, 26407]
Magic constant: 105646

Testing prime: 28439
Primes: [28439, 28447, 28463, 28477, 28493, 28499, 28513, 28517, 28537, 28541, 28547,
28549, 28559, 28571, 28573, 28579]
Magic constant: 114076
Found 24 potential rows

Magic Square Found:
[28559, 28499, 28579, 28439]
[28517, 28571, 28447, 28541]
[28463, 28493, 28573, 28547]
[28537, 28513, 28477, 28549]
Magic constant: 114076

Testing prime: 33547
Primes: [33547, 33563, 33569, 33577, 33581, 33587, 33589, 33599, 33601, 33613, 33617,
33619, 33623, 33629, 33637, 33641]
Magic constant: 134398
Found 30 potential rows

Magic Square Found:
[33587, 33547, 33641, 33623]
[33563, 33637, 33581, 33617]
[33629, 33601, 33599, 33569]
[33619, 33613, 33577, 33589]
Magic constant: 134398

Testing prime: 56813
```

return to Local Index

```
Primes: [56813, 56821, 56827, 56843, 56857, 56873, 56891, 56893, 56897, 56909, 56911,
56921, 56923, 56929, 56941, 56951]
Magic constant: 227550
Found 24 potential rows

Magic Square Found:
[56921, 56923, 56893, 56813]
[56821, 56891, 56941, 56897]
[56951, 56827, 56843, 56929]
[56857, 56909, 56873, 56911]
Magic constant: 227550

Testing prime: 57667
Primes: [57667, 57679, 57689, 57697, 57709, 57713, 57719, 57727, 57731, 57737, 57751,
57773, 57781, 57787, 57791, 57793]
Magic constant: 230936
Found 19 potential rows

Magic Square Found:
[57787, 57667, 57709, 57773]
[57679, 57751, 57793, 57713]
[57689, 57791, 57737, 57719]
[57781, 57727, 57697, 57731]
Magic constant: 230936
```

return to Local Index

MAGNANIMOUS NUMBERS

A magnanimous number is a number (which we assume to be of at least 2 digits) such that the sum obtained inserting a "+" among its digits in any position gives a prime.

Here is the code to generate these numbers up to 26201:

```
M=[]
for n in [11..40000]:
   OK=1
   L=list(str(n))
   for i in [1..len(L)-1]:
      start,end="",""
      for l in L[0:i]:
         start+=l
      for l in L[i:len(L)]:
         end+=l
      sum=int(start)+int(end)
      if is_prime(sum)==0:
         OK=0
   if OK==1:
      M.append(n)
print(M)
```

```
[11, 12, 14, 16, 20, 21, 23, 25, 29, 30, 32, 34, 38, 41, 43, 47, 49, 50, 52, 56, 58, 61, 65, 67,
70, 74, 76, 83, 85, 89, 92, 94, 98, 101, 110, 112, 116, 118, 130, 136, 152, 158, 170, 172,
203, 209, 221, 227, 229, 245, 265, 281, 310, 316, 334, 338, 356, 358, 370, 376, 394, 398,
401, 403, 407, 425, 443, 449, 467, 485, 512, 518, 536, 538, 554, 556, 574, 592, 598, 601,
607, 625, 647, 661, 665, 667, 683, 710, 712, 730, 736, 754, 772, 776, 790, 794, 803, 809,
821, 845, 863, 881, 889, 934, 938, 952, 958, 970, 974, 992, 994, 998, 1001, 1112, 1130,
1198, 1310, 1316, 1598, 1756, 1772, 1910, 1918, 1952, 1970, 1990, 2209, 2221, 2225,
2249, 2261, 2267, 2281, 2429, 2447, 2465, 2489, 2645, 2681, 2885, 3110, 3170, 3310,
3334, 3370, 3398, 3518, 3554, 3730, 3736, 3794, 3934, 3974, 4001, 4027, 4063, 4229,
4247, 4265, 4267, 4427, 4445, 4463, 4643, 4825, 4883, 5158, 5176, 5374, 5516, 5552,
5558, 5594, 5752, 5972, 5992, 6001, 6007, 6067, 6265, 6403, 6425, 6443, 6485, 6601,
6685, 6803, 6821, 7330, 7376, 7390, 7394, 7534, 7556, 7592, 7712, 7934, 7970, 8009,
8029, 8221, 8225, 8801, 8821, 9118, 9172, 9190, 9338, 9370, 9374, 9512, 9598, 9710,
9734, 9752, 9910, 11116, 11152, 11170, 11558, 11930, 13118, 13136, 13556, 15572,
15736, 15938, 15952, 17716, 17752, 17992, 19972, 20209, 20261, 20861, 22061, 22201,
22801, 22885, 24407, 26201, 26285, 26881, 28285, 28429, 31370, 31756, 33118, 33538,
33554, 35116, 35776, 37190, 37556, 37790, 37930, 39158, 39394]
```


return to Local Index

MAP

The function **map**(**func**, seq1, seq2, ...) takes a function **func** and one or more sequences, and applies **func** to elements of those sequences. In particular, you end up with a list like so:

[**func**(seq1[0], seq2[0], ...), **func**(seq1[1], seq2[1], ...), ...]

In many cases, using map allows you to express the logic of your program in a concise manner without using list comprehension. For example, say you have two lists of integers and you want to add them element-wise. A list comprehension to accomplish this would be as follows:

```
A = [1, 2, 3, 4]
B = [2, 3, 5, 7]
[A[i] + B[i] for i in range(len(A))]
```

```
[3, 5, 8, 11]
```

Alternatively, you could use the Python built-in addition function **operator.add** together with **map** to achieve the same result:

```
from operator import add
A = [1, 2, 3, 4]
B = [2, 3, 5, 7]
list(map(add, A, B))
```

```
[3, 5, 8, 11]
```

An advantage of **map** is that you do not need to explicitly define a for loop as was done in the above list comprehension. Source

Here's an example of my own creation:

```
def square(x,y):
   return x^2+y^2
A = [1, 2, 3, 4, 6]
B = [2, 3, 5, 7, 8]
list(map(square, A, B))
```

```
[5, 13, 34, 65, 100]
```

Here's another example, this one taken from page 59 of "Mathematical Computation with SageMath" by Paul Zimmerman:

```
map(cos, [0, pi/6, pi/4, pi/3, pi/2])
```

```
[1, 1/2*sqrt(3), 1/2*sqrt(2), 1/2, 0]
```

return to Local Index

This example involves the use of **lambda** (page 60 of “Mathematical Computation with SageMath” by Paul Zimmerman:

```
map(lambda n : 2*n+1, [0..15])
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31]
```

return to Local Index

MATRIX

If we wish to populate a **matrix** with a given set of numbers, then the following approach can be used. The example given involves populating a sequence of square matrices with increasing prime numbers and finding the **trace** of each successive matrix (2x2, 3x3, …, 20x20). Remember the trace of a square matrix A, denoted tr(A), is defined to be the sum of elements on the main diagonal (from the upper left to the lower right) of A. The trace is only defined for a square matrix (n × n).

```
L=[]
for n in [1..20]:
   P=[]
   for i in [0..n*n-1]:
      P.append(Primes().unrank(i))
   M = matrix(n,P)
   L.append(M.trace())
print(L)
```

```
[2, 9, 36, 99, 224, 407, 724, 1129, 1700, 2451, 3382, 4543, 5986, 7661, 9724, 12041, 14762,
17891, 21482, 25499]
```

Adjusting the above algorithm to “for n in [0..9]” and adding “print M”, the 9x9 matrix produced looks like this (with some slight reformatting to align columns):

```
[  2   3   5   7  11  13  17  19  23]
[ 29  31  37  41  43  47  53  59  61]
[ 67  71  73  79  83  89  97 101 103]
[107 109 113 127 131 137 139 149 151]
[157 163 167 173 179 181 191 193 197]
[199 211 223 227 229 233 239 241 251]
[257 263 269 271 277 281 283 293 307]
[311 313 317 331 337 347 349 353 359]
[367 373 379 383 389 397 401 409 419]
```

To find the **determinant** of a square matrix is quite straightforward (**det(M)** can be used as well):

```
M=matrix([[1,0,-3],[-4,2,3],[3,-1,-2]])
print(M)
print("Determinant is",M.determinant())
```

```
[ 1  0 -3]
[-4  2  3]
[ 3 -1 -2]

Determinant is 5
```

To determine the inverse of a matrix, use **M.inverse()**.

return to Local Index

```
M=Matrix([[1,0,3],[3,-8,5],[3,7,-2]])
print(M)
print()
print(det(M)*M.inverse())
```

```
[ 1  0  3]
[ 3 -8  5]
[ 3  7 -2]

[-19  21  24]
[ 21 -11   4]
[ 45  -7  -8]
```

To determine the **permanent** of a square or rectangular matrix is equally simple. Here is an example for the same square matrix as earlier:

```
M=matrix([[1,0,-3],[-4,2,3],[3,-1,-2]])
print(M)
print()
print("Permanent is",M.permanent())
```

```
[ 1  0 -3]
[-4  2  3]
[ 3 -1 -2]

Permanent is -37
```

Here is an example for a rectangular matrix:

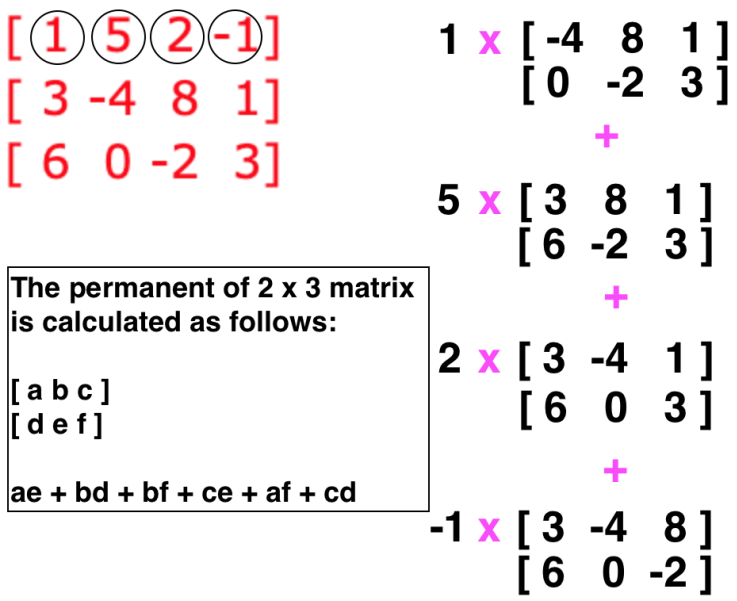
```
M=matrix([[1,5,2,-1],[3,-4,8,1],[6,0,-2,3]])
print(M)
print()
print("Permanent is", M.permanent())
```

```
[ 1  5  2 -1]
[ 3 -4  8  1]
[ 6  0 -2  3]

Permanent is 345
```

return to Local Index

How is the permanent calculated?



The permanent of 2 x 3 matrix is calculated as follows:

[a b c]
[d e f]

ae + bd + bf + ce + af + cd

Here's an example of matrix creation related to:

A209981	Number of singular 2 X 2 matrices having all elements in {-n,...,n}.

```
L=[]
for n in [1..10]:
  a,b,c,d=var('a,b,c,d')
  M=[]
  for a in [-n..n]:
    for b in [-n..n]:
      for c in [-n..n]:
        for d in [-n..n]:
          m=matrix([[a,b],[c,d]])
          M.append(m)
  count=0
  for m in M:
    if m.determinant()==0:
      count+=1
  L.append(count)
print(L)
```

```
[33, 129, 289, 545, 833, 1313, 1729, 2369, 3041, 3905]
```

return to Local Index

A **circulant matrix** is a square matrix in which each row vector is rotated one element to the right relative to the preceding row vector.

```
v=[2,6,0,2,7]
M=matrix.circulant(v)
print(M)
```

```
[2 6 0 2 7]
[7 2 6 0 2]
[2 7 2 6 0]
[0 2 7 2 6]
[6 0 2 7 2]
```

Here is an example of its use:

A219324	Positive integers n that are equal to the determinant of the circulant matrix formed by the decimal digits of n.

1, 2, 3, 4, 5, 6, 7, 8, 9, 247, 370, 378, 407, 481, 518, 592, 629, 1360, 3075, 26027

```
L=[]
for n in [1..26027]:
   v=list(reversed(n.digits()))
   M=matrix.circulant(v)
   if n==det(M):
      L.append(n)
print(L)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 247, 370, 378, 407, 481, 518, 592, 629, 1360, 3075, 26027]
```

Getting **v** right is a little tricky, read Reverse a List for the explanation.

It can be noted that the **trace** of a circulant matrix of a number will always be the product of its first digit by its number of digits. Thus for a number like 26027, the trace of its circulant matrix will be 2 x 5 = 10.

return to Local Index

MEDIANT

The mediant is simple enough to calculate. Given two fractions, a/b and c/d, the mediant is given by (a+b)/(c+d) and a/b < (a+b)/(c+d) < c/d. Of more interest is its use in approximating irrational and transcendental numbers. Here is an example of its use in approximating e.

```
target=e
number=target-int(target)
numL,denL,numR,denR=0,1,1,1
error=0.000000000001
L=[]
while abs(number-(numL+numR)/(denL+denR))>error:
   L.append(int(target)+(numL+numR)/(denL+denR))
   if (numL+numR)/(denL+denR)> number:
      numR=numL+numR
      denR=denL+denR
   else:
      numL=numL+numR
      denL=denL+denR
L.append(int(target)+(numL+numR)/(denL+denR))
print("Final approximating fraction for",target,"is",int(target)+(numL+numR)/(denL+denR))
print("Mediant decimal approximation is
for",target,"is",int(target)+numerical_approx((numL+numR)/(denL+denR)))
print("Actual decimal approximation for",target,"is",numerical_approx(target))
print("The progressive list of approximating fractions is given by:")
print(L)
```

```
Final approximating fraction for e is 1084483/398959
Mediant decimal approximation is for e is 2.71828182845856
Actual decimal approximation for e is 2.71828182845905
The progressive list of approximating fractions is given by:
[5/2, 8/3, 11/4, 19/7, 30/11, 49/18, 68/25, 87/32, 106/39, 193/71, 299/110, 492/181, 685/252,
878/323, 1071/394, 1264/465, 1457/536, 2721/1001, 4178/1537, 6899/2538, 9620/3539,
12341/4540, 15062/5541, 17783/6542, 20504/7543, 23225/8544, 25946/9545, 49171/18089,
75117/27634, 124288/45723, 173459/63812, 222630/81901, 271801/99990, 320972/118079,
370143/136168, 419314/154257, 468485/172346, 517656/190435, 566827/208524,
1084483/398959]
```

return to Local Index

MODEST NUMBER

A number is modest if there exists at least one partitioning of its decimal expansion wherein the number divided by the second part leaves a remainder of the first part.

For example:

- 2036 is modest because 2036 mod 36 = 20.
- 2037 is modest because 2037 mod 037 = 2.

Consecutive modest numbers like 2036 and 2037 are rare however. Here is a list up to 40,000 of numbers n such that n and $n+1$ are both modest: n such that n and $n+1$ are both modest:

411, 811, 1421, 2036, 2044, 2054, 3054, 4036, 4044, 4054, 8036, 12036, 16036, 20036, 24036, 28036, 32036, ...

By contrast, standalone modest numbers are far more common. In the range up to 40000, they occupy 2.25% of the range. Here is an algorithm to determine all the modest numbers in the range up to given limit, here 40000 (permalink):

```
L=[]
i,j=var('i,j')
upper=40000
for n in [11..upper]:
   l=len(str(n))
   for i in range(0,l-1):
      a=str(n) [0:i+1]
      b=str(n) [i+1:l]
      if int(b)!=0:
         if n%int(b)==int(a):
            L.append(n)
            break
print(L)
print(len(L)/upper*100.n(digits=3),"%")
```

[13, 19, 23, 26, 29, 39, 46, 49, 59, 69, 79, 89, 103, 109, 111, 133, 199, 203, 206, 209, 211, 218, 222, 233, 266, 299, 309, 311, 327, 333, 399, 406, 409, 411, 412, 418, 422, 433, 436, 444, 466, 499, 509, 511, 515, 533, 545, 555, 599, 609, 611, 618, 622, 627, 633, 654, 666, 699, 709, 711, 721, 733, 763, 777, 799, 809, 811, 812, 818, 822, 824, 833, 836, 844, 866, 872, 888, 899, 911, 927, 933, 981, 999, 1003, 1009, 1011, 1015, 1018, 1022, 1027, 1030, 1033, 1037, 1045, 1055, 1066, 1090, 1099, 1111, 1133, 1199, 1218, 1222, 1227, 1233, 1236, 1244, 1254, 1266, 1299, 1333, 1339, 1399, 1418, 1421, 1422, 1433, 1442, 1463, 1466, 1477, 1499, 1527, 1533, 1545, 1555, 1599, 1618, 1622, 1624, 1633, 1636, 1644, 1648, 1666, 1672, 1688, 1699, 1733, 1751, 1799, 1822, 1827, 1833, 1854, 1866, 1881, 1899, 1933, 1957, 1999, 2003, 2006, 2009, 2018, 2022, 2027, 2030, 2033, 2036, 2037, 2044, 2045, 2054, 2055, 2060, 2066, 2074, 2090, 2099, 2111, 2127, 2133, 2163, 2177, 2199, 2222, 2233, 2266, 2299, 2333, 2369, 2399, 2427, 2433, 2436, 2444, 2454, 2466, 2472, 2488, 2499, 2533, 2545, 2555, 2575, 2599, 2633, 2639, 2666, 2678, 2699, 2733, 2781, 2799, 2833, 2836, 2842, 2844, 2863, 2866, 2877, 2884, 2899, 2933, 2987, 2999, 3009, 3027, 3033, 3037, 3045, 3054, 3055, 3066, 3081, 3090,

return to Local Index

3099, 3111, 3133, 3193, 3199, 3233, 3236, 3244, 3248, 3266, 3272, 3288, 3296, 3299, 3333, 3399,
3451, 3466, 3499, 3545, 3555, 3563, 3577, 3599, 3644, 3654, 3666, 3681, 3699, 3799, 3857, 3866,
3899, 3999, 4006, 4009, 4012, 4018, 4027, 4036, 4037, 4044, 4045, 4054, 4055, 4060, 4066, 4072,
4074, 4088, 4090, 4099, 4108, 4111, 4148, 4199, 4222, 4254, 4263, 4266, 4277, 4299, 4333, 4399,
4444, 4466, 4499, 4555, 4581, 4599, 4666, 4669, 4699, 4799, 4854, 4866, 4872, 4888, 4899, 4963,
4977, 4999, 5009, 5015, 5027, 5037, 5045, 5055, 5066, 5075, 5090, 5099, 5111, 5135, 5185, 5199,
5266, 5278, 5299, 5333, 5399, 5466, 5481, 5499, 5555, 5599, 5663, 5666, 5672, 5677, 5684, 5688,
5699, 5799, 5866, 5887, 5899, 5999, 6009, 6018, 6027, 6037, 6054, 6066, 6074, 6081, 6090, 6099,
6111, 6162, 6199, 6222, 6266, 6293, 6299, 6333, 6377, 6381, 6399, 6466, 6472, 6488, 6496, 6499,
6599, 6666, 6699, 6799, 6899, 6999, 7009, 7021, 7027, 7037, 7063, 7077, 7090, 7099, 7111, 7189,
7199, 7259, 7281, 7288, 7299, 7333, 7399, 7499, 7599, 7699, 7777, 7799, 7899, 7999, 8009, 8012,
8018, 8024, 8027, 8036, 8037, 8054, 8072, 8074, 8088, 8090, 8099, 8108, 8111, 8148, 8199, 8216,
8222, 8296, 8299, 8333, 8399, 8444, 8499, 8599, 8666, 8699, 8799, 8888, 8899, 8999, 9027, 9037,
9081, 9099, 9111, 9199, 9243, 9299, 9333, 9399, 9499, 9599, 9699, 9799, 9899, 9999, 10003,
10009, 10011, 10015, 10018, 10027, 10030, 10033, 10037, 10045, 10054, 10074, 10090, 10099,
10101, 10111, 10135, 10185, 10222, 10270, 10303, 10333, 10370, 10555, 10666, 10909, 10999,
11027, 11033, 11037, 11099, 11111, 11297, 11333, 11407, 11999, 12018, 12027, 12036, 12037,
12054, 12074, 12081, 12108, 12111, 12148, 12162, 12222, 12324, 12333, 12444, 12666, 12999,
13027, 13037, 13039, 13111, 13117, 13333, 13351, 13481, 13999, 14018, 14021, 14027, 14037,
14042, 14054, 14063, 14074, 14111, 14126, 14189, 14222, 14259, 14333, 14378, 14518, 14666,
14777, 14999, 15027, 15037, 15045, 15081, 15111, 15135, 15185, 15333, 15405, 15555, 15999,
16018, 16024, 16027, 16036, 16037, 16048, 16054, 16072, 16074, 16108, 16111, 16144, 16148,
16216, 16222, 16296, 16333, 16432, 16444, 16592, 16666, 16888, 16999, 17027, 17037, 17051,
17111, 17153, 17333, 17459, 17629, 17999, 18027, 18037, 18054, 18074, 18081, 18111, 18162,
18222, 18243, 18333, 18486, 18666, 18999, 19027, 19037, 19057, 19111, 19171, 19333, 19513,
19703, 19999, 20003, 20006, 20009, 20011, 20018, 20022, 20027, 20030, 20033, 20036, 20037,
20045, 20054, 20060, 20066, 20074, 20090, 20099, 20101, 20108, 20111, 20135, 20148, 20180,
20185, 20198, 20202, 20222, 20270, 20303, 20333, 20370, 20444, 20540, 20555, 20606, 20666,
20740, 20909, 20999, 21027, 21037, 21063, 21081, 21111, 21189, 21259, 21333, 21567, 21777,
21818, 21999, 22027, 22033, 22037, 22054, 22066, 22074, 22099, 22111, 22198, 22222, 22297,
22333, 22407, 22594, 22666, 22814, 22999, 23027, 23037, 23069, 23111, 23207, 23333, 23621,
23851, 23999, 24027, 24036, 24037, 24054, 24072, 24074, 24081, 24108, 24111, 24148, 24162,
24216, 24222, 24296, 24324, 24333, 24444, 24648, 24666, 24888, 24999, 25027, 25037, 25045,
25075, 25111, 25135, 25185, 25225, 25333, 25555, 25675, 25925, 25999, 26027, 26037, 26039,
26054, 26074, 26078, 26111, 26117, 26222, 26234, 26333, 26351, 26481, 26666, 26702, 26962,
26999, 27037, 27081, 27111, 27243, 27333, 27729, 27999, 28036, 28037, 28042, 28054, 28063,
28074, 28084, 28108, 28111, 28126, 28148, 28189, 28222, 28252, 28259, 28333, 28378, 28444,
28518, 28666, 28756, 28777, 28999, 29037, 29087, 29111, 29261, 29333, 29783, 29999, 30009,
30011, 30027, 30033, 30037, 30045, 30054, 30074, 30081, 30090, 30099, 30101, 30111, 30135,
30162, 30185, 30222, 30270, 30297, 30303, 30333, 30370, 30405, 30555, 30666, 30810, 30909,
30999, 31037, 31093, 31111, 31279, 31333, 31837, 31999, 32036, 32037, 32048, 32054, 32072,
32074, 32096, 32108, 32111, 32144, 32148, 32216, 32222, 32288, 32296, 32333, 32432, 32444,
32592, 32666, 32727, 32864, 32888, 32999, 33037, 33081, 33099, 33111, 33297, 33333, 33407,
33891, 33999, 34037, 34051, 34054, 34074, 34102, 34111, 34153, 34222, 34306, 34333, 34459,
34629, 34666, 34918, 34999, 35037, 35045, 35063, 35105, 35111, 35135, 35185, 35189, 35259,
35315, 35333, 35555, 35777, 35945, 35999, 36037, 36054, 36074, 36081, 36108, 36111, 36148,
36162, 36222, 36243, 36324, 36333, 36444, 36486, 36666, 36972, 36999, 37111, 37333, 37999,

return to Local Index

38054, 38057, 38074, 38111, 38114, 38171, 38222, 38333, 38342, 38513, 38666, 38703, 38999, 39081, 39111, 39117, 39333, 39351, 39481, 39999]
2.25 %

return to Local Index

MODULAR ARITHMETIC

To set up modular arithmetic, you choose a modulus, let's say 26, by simply declaring R = **IntegerModRing**(26) and then to refer to numbers as living in the world mod 26, you wrap them in the name of your number ring, R, i.e. you write R(3) instead of 3. Then you can do arithmetic with them e.g.

```
R = IntegerModRing(26)
print R(25) + R(2)
print R(2) -R(9)
print R(3) * R(11)
print R(21)/R(15)

1
19
7
17
```

Of course, unless the number of elements is a prime number, division will not always be possible. For example,

```
R = IntegerModRing(26)
print R(21)/R(16)

ZeroDivisionError: inverse of Mod(16, 26) does not exist
```

return to Local Index

MOEBIUS FUNCTION

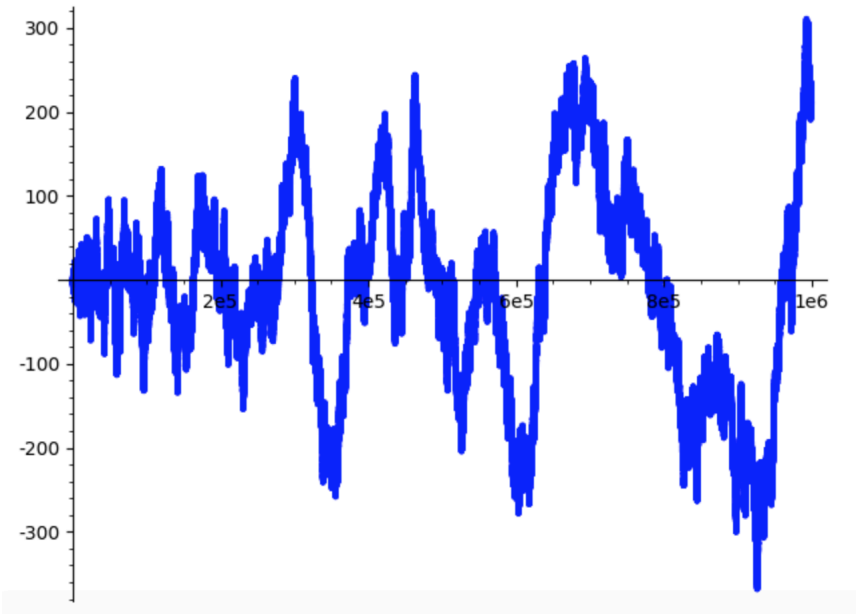
This function might be useful at some point. Input of multiples of square numbers returns 0, numbers with an even number of distinct prime factors returns 1 and numbers with an odd number of distinct prime factors returns -1.

```
for n in [1..10]:
   print(n, "-->",factor(n), "-->",moebius(n))
```

```
1 --> 1 --> 1
2 --> 2 --> -1
3 --> 3 --> -1
4 --> 2^2 --> 0
5 --> 5 --> -1
6 --> 2 * 3 --> 1
7 --> 7 --> -1
8 --> 2^3 --> 0
9 --> 3^2 --> 0
10 --> 2 * 5 --> 1
```

The cumulative Moebius function, the result of progressively adding the Moebius values of the integers, is associated with **Mertens Conjecture** (or least the visualisation of it):

```
L=[]
sum=0
for n in [1..1000000]:
   sum+=moebius(n)
   L.append(sum)
list_plot(L,plotjoined=False)
```



return to Local Index

MORAN NUMBERS

A number is a Moran number if, when divided by the sum of its digits, the result is a prime number. In this respect, a Moran number is a particular type of HARSHAD NUMBER.

The following SageMath code can be used to identify Moran numbers in a given range:

```
for n in [25600..25700]:
  if n % sum(n.digits())==0 and is_prime(int(n /sum(n.digits()))):
    print n,
```

```
25614 25656 25691 25699
```

The first pair of consecutive Moran numbers is (152, 153, while the first runs of length 3, 4, 5 and 6 start at 3031, 21481224, 25502420, and 4007565001480, respectively. For this last value we have

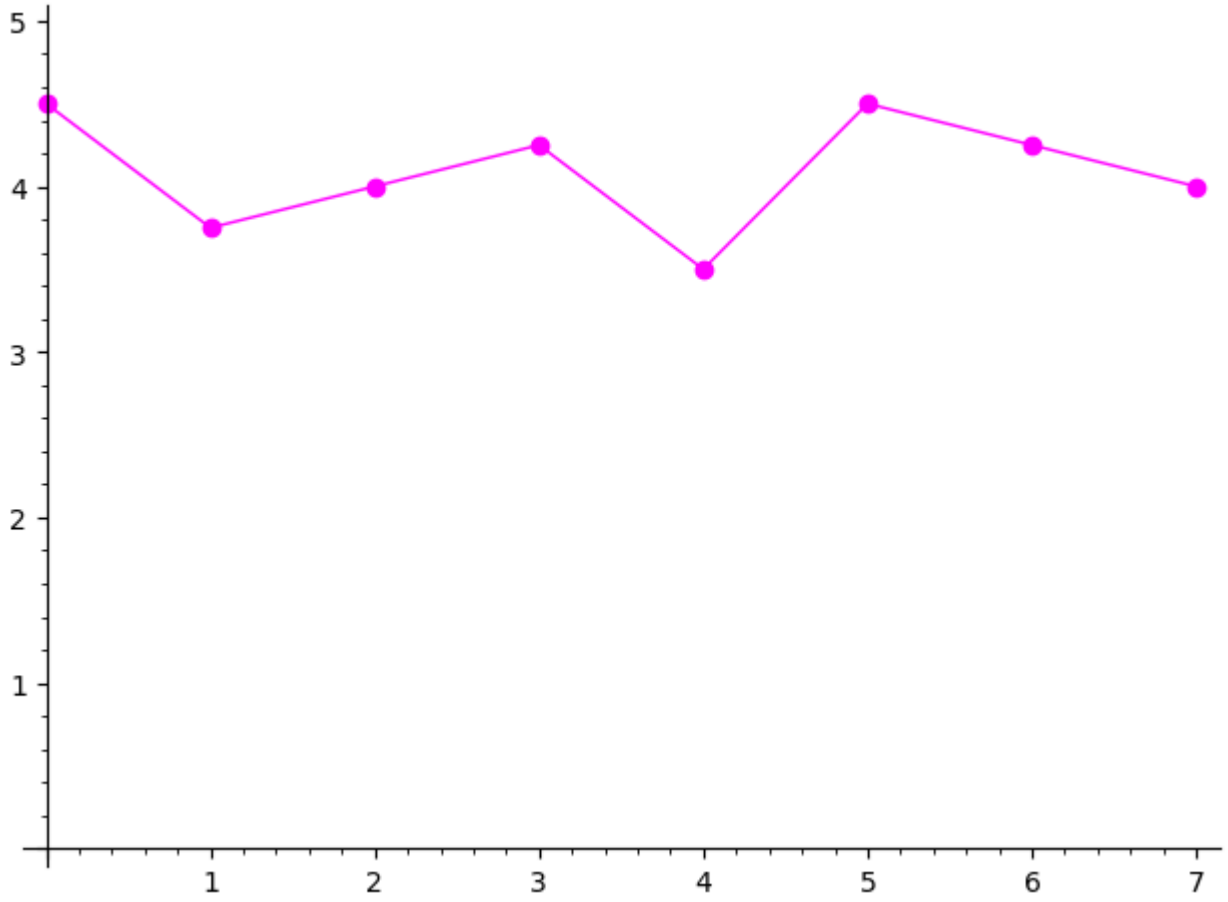
$$\begin{aligned}
4007565001480 &= 40 \cdot 100189125037 \\
4007565001481 &= 41 \cdot 97745487841 \\
4007565001482 &= 42 \cdot 95418214321 \\
4007565001483 &= 43 \cdot 93199186081 \\
4007565001484 &= 44 \cdot 91081022761 \\
4007565001485 &= 45 \cdot 89057000033
\end{aligned}$$

return to Local Index

MOVING AVERAGE

Here is an example of the use of the moving average function.

```
L=[6,3,4,5,3,4,5,2,7,3,4]
M=moving_average(L,4)
G=list_plot(M,plotjoined=True, marker="o", color="magenta")
G.show(ymin=0,ymax=5)
```



return to Local Index

MULTIFACTORIAL

Compute the k-th factorial $n!^{(k)}$ of self.

The multifactorial number $n!^{(k)}$ is defined for non-negative integers n as follows. For $k = 1$ this is the standard factorial, and for k greater than 1 it is the product of every k-th terms down from n to 1. The recursive definition is used to extend this function to the negative integers n.

Using the example of the 6-th multifactorial, it can also be referred to as 6-factorial, 6!!!!!! and n!6. The command is **n.multifactorial(k)** and by way of illustration, it can be applied to elements of OEIS A289697 whose initial members are 9, 11, 13, 17, 23, 25, 29, 31, 37, 43, 53, 65, 71, 77, 79, ...

A289697	Numbers k such that k!6 - 24 is prime, where k!6 is the sextuple factorial number (A085158).

```
n=37
print(n.multifactorial(6)-24, is_prime(n.multifactorial(6)-24))

49579051 True
```

The algorithm will time out in SageMathCell if the numbers are too large. Another approach is to use **prod** and **step** as shown but, even though this gives the correct answer, it introduces a problem with the recognition of a number as a prime. I don't know the reason for this. Here's what I mean:

```
n=37
print(n.multifactorial(6)-24, is_prime(n.multifactorial(6)-24))
print(prod([n..1,step=-6])-24, is_prime(prod([n..1,step=6])-24))

49579051 True
49579051 False
```

return to Local Index

MULTIPLICATIVE PERSISTENCE

The multiplicative persistence of a number counts the number of steps required to reach zero when the digits are multiplied together. Here is an example of its use:

A046515	Numbers with multiplicative persistence value 6.

Here is an algorithm to identify these numbers (up to 26748).

```
persistence=6
L=[]
for n in [11..40000]:
  number=n
  T=[]
  while len(n.digits())>1:
    T.append(n)
    n=prod(n.digits())
  L.append((number,n,len(T)))
N=[]
for n in L:
  if n[2]==persistence:
    N.append(n[0])
print(N)
```

[6788, 6878, 6887, 7688, 7868, 7886, 8678, 8687, 8768, 8786, 8867, 8876, 16788, 16878, 16887, 17688, 17868, 17886, 18678, 18687, 18768, 18786, 18867, 18876, 23788, 23878, 23887, 24678, 24687, 24768, 24786, 24867, 24876, 26478, 26487, 26748, 26784, 26847, 26874, 27388, 27468, 27486, 27648, 27684, 27838, 27846, 27864, 27883, 28378, 28387, 28467, 28476, 28647, 28674, 28738, 28746, 28764, 28783, 28837, 28873, 32788, 32878, 32887, 34478, 34487, 34748, 34784, 34847, 34874, 37288, 37448, 37484, 37828, 37844, 37882, 38278, 38287, 38447, 38474, 38728, 38744, 38782, 38827, 38872]

Not all numbers will reach zero. Some will reach a fixed number e.g. 26662

```
n=26662
L=[n]
while len(n.digits())>1:
  n=prod(n.digits())
  L.append(n)
print(L)
```

[26662, 864, 192, 18, 8]

The final 8 of [26662, 864, 192, 18, 8] is known as the multiplicative digital root of 26662 and is the multiplicative equivalent of the (additive) digital root (here it is 4). The digital root *d* of a

return to Local Index

number is always 0 <= *d* < *b* where *b* is the number base. So for base 10, we have 0 <= *d* <10. It's this property that I used in the above algorithms.

Here is an algorithm courtesy of z.ai that will find the first 100 primes that have a multiplicative persistence of 6 (permalink and OEIS A046506 link).

```
def filter(n):
    """
    Filter function that checks if a number meets the multiplicative persistence condition.
    Returns True if after 6 multiplications of digits the result is < 10,
    and all intermediate results (first 5 multiplications) are >= 10.
    """
    # Convert number to list of digits and compute their product
    L = prod(Integer(n).digits())

    # If first product is < 10, return False
    if L < 10:
        return False

    # Repeat the process 4 more times (total 5 times)
    for i in range(4):
        L = prod(Integer(L).digits())
        if L < 10:
            return False

    # Do one final multiplication
    L = prod(Integer(L).digits())

    # Return True if final result is < 10
    return L < 10

# Initialize variables
count = 0
Res = []
p = 11  # Starting prime

# Find 100 primes that satisfy the filter condition
while count < 100:
    p = next_prime(p)  # Get next prime
    if filter(p):
        count += 1
        Res.append(p)

# Print the result
print(Res)
```

[8867, 23887, 27883, 28387, 28837, 32887, 34487, 34847, 38287, 38447, 43487, 44647, 46447, 47843, 48437, 48473, 49999, 72883, 74843, 78283, 78823, 82387, 82837, 84347, 84437, 87443,

return to Local Index

88237, 88327, 94999, 118687, 123887, 126487, 128467, 128837, 128873, 132887, 142867, 148627, 162847, 164447, 167887, 168247, 168781, 172883, 176887, 178681, 182387, 182467, 184627, 186187, 186247, 186871, 186877, 187687, 187823, 187861, 188273, 188677, 188767, 196699, 199499, 199669, 213887, 214867, 218783, 218873, 224467, 232487, 232847, 234287, 238247, 238781, 238877, 239699, 241687, 241867, 242467, 242647, 242873, 246187, 246247, 246781, 246787, 246817, 248167, 248723, 261847, 264787, 264871, 267481, 268747, 269939, 273881, 274283, 274861, 274867, 276487, 276847, 277883, 278387]

return to Local Index

MULTIPURPOSE ALGORITHM

This is too large to post here and I have it saved elsewhere.

return to Local Index