(Please see attached .ods/.xls file for specific data)

The algorithm used for hashing is an implementation of the classic Prime-plus two prime multiplier hash, with two prime numbers, A and B that were chosen to be sufficiently large without being unreasonably huge. For 64 bit systems and the current implementation, best coverage and performance was found at A= 263, B = 1000000007. When A becomes larger, say, 270, the coverage of the function’s hash matching decreases noticably. When it is smaller than 260, clash-avoidance suffers.

Prime-plus (also known as PolyPrime) is compared with the standard example and first-order factoring function proposed in the course notes.

Several values of B were tried; 1000000021 and 1000000033 are the highest values attempted. Nothing smaller than 1000000007 was used – for the sheer reason that for the multiplicative hash function B has to be sufficiently large to allow randomisation: B is used as *large prime modulo* factor in the function, where the hash value

hash[f] := (k\*A mod B) mod s where k is the key value we’re interested in hashing, and s is the key/bucket size that is being used. Key/bucket size *must* match the size of the bucket dicts for this function to work properly.

Through repeated runs; it found that this function; the poly\_prime hash function; has *far better* coverage than the default function proposed; *and* it has more regular distributions throughout the entire space of the key; packing values more “evenly” and finding more key value matches in large key sets.

Performance differences between Prime and PolyPrime become even more apparent for larger and larger sizes of key\_size and array\_size; demonstrating the more “Even” nature of PolyPrime in creating appropriately regular arrays.

One further optimisation is focused on the *type of data* we’re hashing. Jnumbers are *always* one character, and eight digits. We take the eight digits at their integer value and discard the J as it is a constant and thus should not form part of the hash table at all – then the resulting number is processed with PolyPrime to arrive at the appropriate hash value we’re interested in.

NumPy was used as a source of “truth” for random numerical values in the testing, approaching the best possible entropy source given the project and distance from true RNG hardware and time/space constraints.