APPENDIX 1

The partition function—Here, the mathematical convention of representing the total as *n* (instead of *q*) is followed. The number of partitions of an integer *n*, i.e. *p*(*n*), equals the number of partitions of *n* having *n* or less as the largest element (Andrews & Eriksson 2006, Bóna 2006). Likewise, the number of partitions of an integer *n* into *k* or less parts, i.e. $p_k(n)$ equals the number of partitions of an integer *n* into *k* or less parts and partitions of *n* having *k* or less as the largest part (Bóna 2006). Consequently, knowing $p_k(n)$ for $k = \{1, \ldots, n)$ leads to p(*n*), the generating function for which is given by:

$$\sum_{n=0}^{\infty} p(n)x^n = \prod_{k=1}^{\infty}\left(\frac{1}{1-x^k}\right)$$

The recurrence relation for the number of partitions of *n* into parts having *k* or less as the largest part (or having *k* or less parts or *k* or less as the first part):

$$p_k(n) = p_k(n-k) + p_{k-1}(n)$$

Interpretation: Partitions of *n* into at most *k* parts either have exactly *k* parts or they have fewer than *k* parts. By convention $p(n = 0) = 1$. Also $p_{k=0}(n) = 0$ if $n > 0$.

Simple recursive function showing how the recurrence relation is implemented. This function is too slow for ecologically realistic values of :

```
def parts(n, k):   # find the number of partitions for q with k as the largest part
   if k == 0: return 0
   if n == 0: return 1
   if n < 0: return 0
   return parts(n, k - 1) + parts(n - k, k)
```

Partition functions for Locey and McGlinn (2013) can be found here:
https://github.com/klocey/partitions/blob/master/partitions.py

Explanation of the ‘multiplicity’ approach, specifically the statement: The set of partitions of q having a number of k’s equal to m contains the set of partitions of $q – k*m$ having less than k as the first part:

Consider the 8 partitions of $q = 10$ having $k = 3$ as the first element and note that the possible multiples (m) of 3 are 3, 2, 1:

[3, 3, 3, 1]

[3, 3, 2, 2]

[3, 3, 2, 1, 1]

[3, 3, 1, 1, 1, 1]

[3, 2, 2, 2, 1]

[3, 2, 2, 1, 1, 1]

[3, 2, 1, 1, 1, 1, 1]

[3, 1, 1, 1, 1, 1, 1, 1]

According to the above statement, the partitions of $q = 10$ having three 3’s will contain the set of partitions of $q – k*m = 10 – 9 = 1$ having less than 3 as the first part. Indeed, there is only one possible partition, i.e. [1].

Likewise, the partitions of $q = 10$ having two 3’s will contain the set of partitions of $q – k*m = 10 – 6 = 4$ having less than 3 as the first part. Indeed, the possible partitions are [2, 2], [2, 1, 1] and [1, 1, 1, 1], i.e. the feasible set for $q = 4$ having less than three as the largest part.

Finally, the partitions of $q = 10$ having one 3 will contain the set of partitions of $q – k*m = 10 – 3 = 7$ having less than 3 as the first part. Indeed, the possible partitions are [2, 2, 2, 1], [2, 2, 1, 1, 1], [2, 1, 1, 1, 1, 1] and [1, 1, 1, 1, 1, 1, 1], i.e. the feasible set for $q = 7$ having less than three as the largest part.

Fig 1. Kernel density curves for **skewness** are derived from random samples of 500 partitions for different combinations of Q and N reveal that the integer partitioning algorithms, i.e. Multiplicity, Top-down, Divide and conquer, Bottom-up produce random samples of feasible sets (FS) both when excluding and including parts having zero values, left and right, respectively. We used Sage to generate the entire feasible set (thick red line) for Q = 50 and N = 10. We used the random partitioning function in Sage to generate 500 partitions for Q = 500 and N = 10, which is too large to enumerate in full.

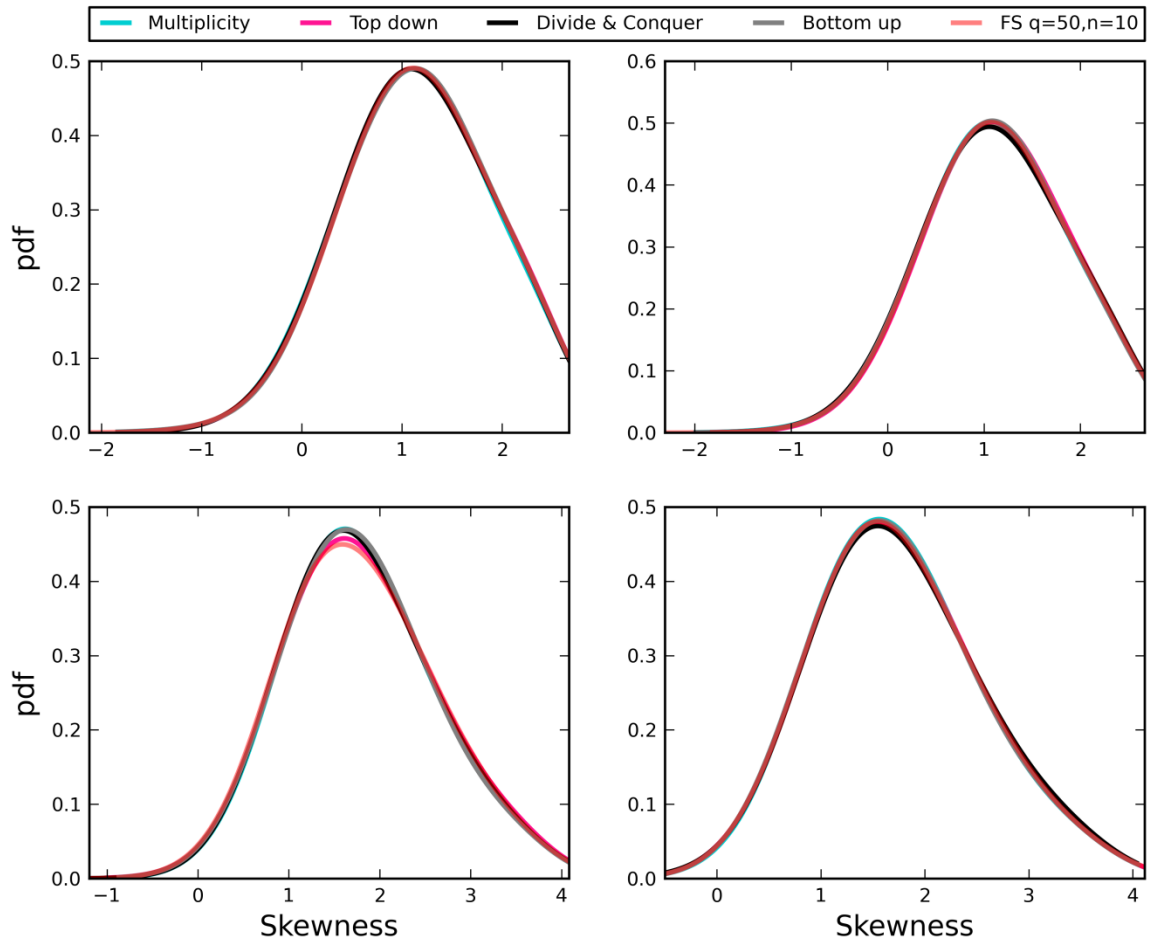

Fig 2. Kernel density curves for the **variance** are derived from random samples of 500 partitions for different combinations of Q and N reveal that the integer partitioning algorithms, i.e. Multiplicity (Multi), Top down (T-D), Divide and conquer (D-Q), Bottom up (B-U) produce random samples of feasible sets (F-S) both when excluding and including parts having zero values, left and right, respectively. We used Sage to generate the entire feasible set (thick grey line) for Q = 50 and N = 10. We used the random partitioning function in Sage to generate 500 partitions for Q = 500 and N = 10, which is too large to enumerate in full.

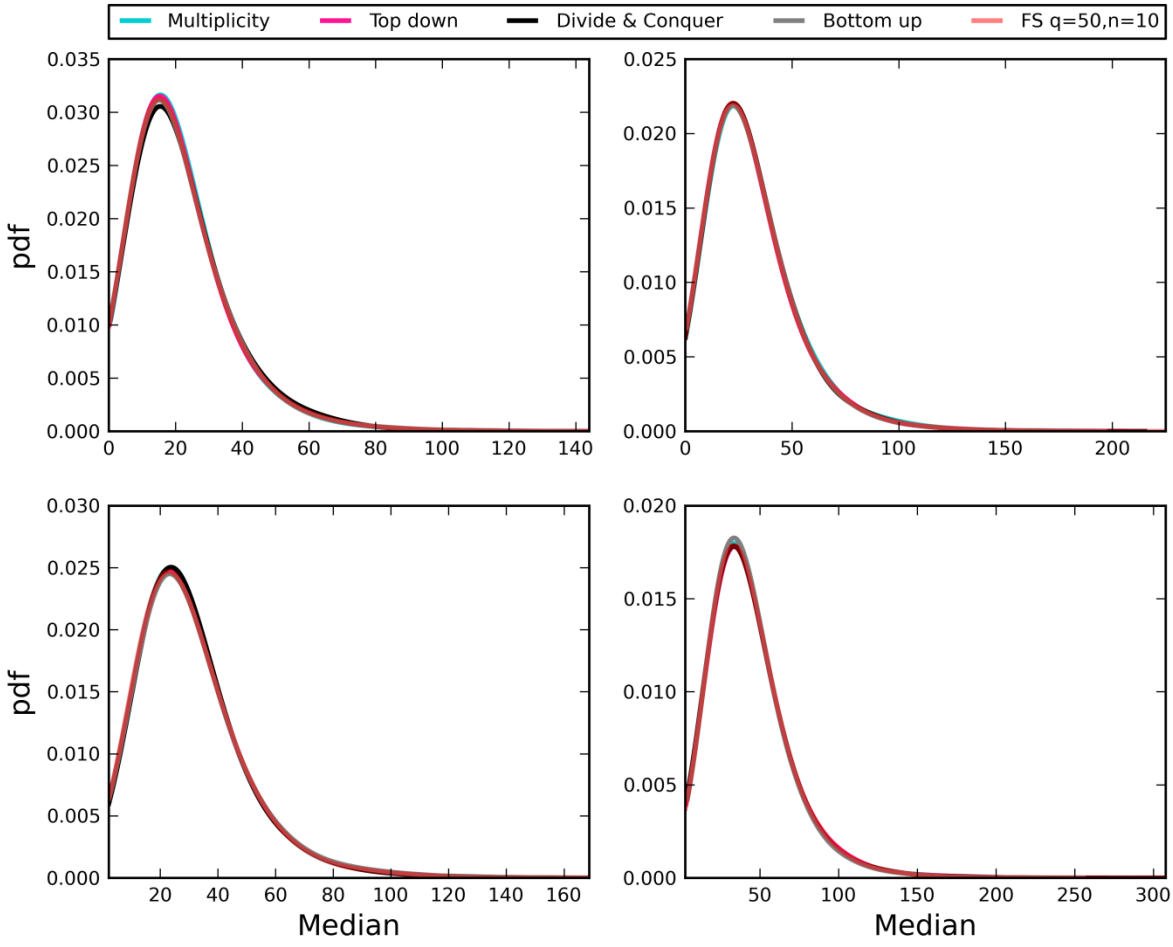

Fig 3. Time required to generate a single random partition (no zero values) for ecologically large combinations of Q and N using the random partitioning algorithms derived here, implemented in Python. The multiplicity algorithm greatly outperforms the bottom-up and divide-and-conquer methods for these types of Q-N combinations.