Small input:  
{1,2,3,4,5,6,7, 8, 9, 10}

{{4,5,6,7}, {2, 4, 5}, {6,7}, {1,3,7}}

Medium input:

{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20}

{{4,6,7}, {3,4,5}, {6,7,8,9,10}, {2}, {1,3,7,11}, {2,3,4,6,8,20}, {1, 4, 5, 6, 8, 11, 14}}

Large input:

{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27}

{{1}, {1,2}, {3}, {4, 9, 12, 13, 15, 19, 25}, {1, 6, 7, 10, 11, 17, 21, 23, 24, 25}, {4, 5, 7, 8, 9, 10, 11, 16, 19, 20}, {7, 19, 27}, {7, 8, 14, 16}, {15, 16, 17, 20}, {3, 6, 23, 26}, {5, 13, 17, 18}, {3, 10, 11, 13, 21, 27}, {3, 6, 18, 21, 26, 27}, {2, 3, 5, 6, 8, 15}}

Thought process on fitness function and stop criteria.

Hitting set does not have a static solution size so I decided to make half the fitness function on how accurate the solution size was. If it was the exact size, the fitness function would automatically be 1. I made the elements of the solution worth a total of 1 fitness score also. Giving a combined max fitness score of 2. Each element was worth the same amount so the calculation for each element was, 1/array size. Each element correct added 1/array\_size to the fitness score (in hitting set there can be more than one solution so I checked each correct solution, to my current chromosome and took the fitness score that solution would give, then went to the next correct solution produced from my brute force method, if this fitness score was higher for my current chromosome, the higher score was used, so lets say the solution size was 2, and one SOLUTION was [5,7] , and another was [3, 9], the chromosome , [3,7] would only produce a fitness score of 1.5, not 2). If the element was not 100% correct but when taking the floor produced an element that can be in the solution, half the score was given, so (1/array\_size)/2 was added to the fitness score. So let say my chromosome was 1.7 and solution was 1, that would give half points to the score.

My stop criteria was just the generation count. Once the max generations where reached, the program stopped.

Each time the program is ran, the chromosome with the highest fitness score changes, due to the rand function being used to crossover rate, mutation rate, and initializing the initial population (the first generation). So the fitness scores for the chromosomes with the best fitness score and the chromosome themselves can be view the output file. These are the results from the last run I did.

The small input set produced the chromosome: [0.7044040050704694, 0.41447828121054053] with a fitness score of 1.5.

The medium input set produced the chromosome: [0.73581828383972, 0.21386242837068603, 0.5632745369200645] with the fitness score of 1.5.

The large input set produced the chromosome: [0.3113532117688339, 0.8927390544762153, 0.19878370489950492, 0.7664425500642832, 0.11640729492880764] with the fitness score of 1.4.

In terms of random verse GA, GA produced much faster results without a lose of precision or accuracy. The GA part of my code ran much faster than the random and brute force. With the more input sizes, the more obvious the run time different between random and brute force, random being much faster. And the much more obvious run time different between random and GA.

Brute force produced the 100% correct solution, but both GA and Random produced close solutions to the correct solution. Most situations, the solution was only a few decimals values off.

The larger the input size, the less accurate the solutions became but still maintained a high degree of accuracy. The average fitness value for the ga solution using the large input size was approximately 1 (out of 2) for each run and the max fitness value for the ga solution using the large input size was always the same or greater then the random, which was usually at 1.4. The random average fitness value for the large input set was usually slightly less then 1.