**CountKnapsackApprox**

The algorithm for random uniform sampling of the solutions to the rounded version of the problem is presented below:

```
0   procedure drawRandomSample(a[], n, table[][])
1         solutions <- []
2         n² <- bound <- n*n
3         for i <- n downto 1 do
4               if bound >= a[i-1] do
5                     ratio <- table[i-1][bound - a[i-1]] / table[i][bound]
6                     random <- Math.random()
7                     if ratio <= random do
8                           solutions <- solutions.push(i)
9                           bound <- bound - a[i-1]
10        return solutions
```

The algorithm works as follows:

We loop over the rows of and at each row we decide whether to `i`, the row number, into the set of feasible solutions or not. This choice is made on random and depends on the ratio of `count(i,n²)` and `count(i,n²-a_n)`. The ratio compared to a randomly generated value decides if we take `i` as a feasible solution or not. If a decision to include the solution is made, our bound variable is reduced by the value of $a_i$.

**Termination:**

The algorithm always terminates as there is a fixed number of steps in terms of n.

**Correctness:**

For each row of the pre-computed table, we either decide to pick `count(i,n²)` or `count(i,n²-a_n)` with probability given by their ratio. Not taking `count(i,n²-a_n)` will cause the row to be skipped and no index taken - sampling is still uniform and over the complete range of numbers. However, if we do decide to take `count(i,n²-a_n)`, our range for uniform sampling shrinks by $a_i$. All subsequent samples, if taken, will be solutions to the original problem and thus valid solutions. Therefore, the algorithm samples uniformly from the set of solutions to the original problem.

**Runtime:**

Firstly, initialization and pre-assignment of values occurs in lines 1-2, all operations are performed in constant time `Θ(1)`. Secondly, we loop n times in line 3. For lines 4-9, we perform at most six constant time operations - `Θ(6)=Θ(1)`. Therefore, lines 3-9 will be executed n `Θ(1)` times which simplifies to `Θ(n)`. Finally, line 10 is executed once giving us final running time of `Θ(n)`.

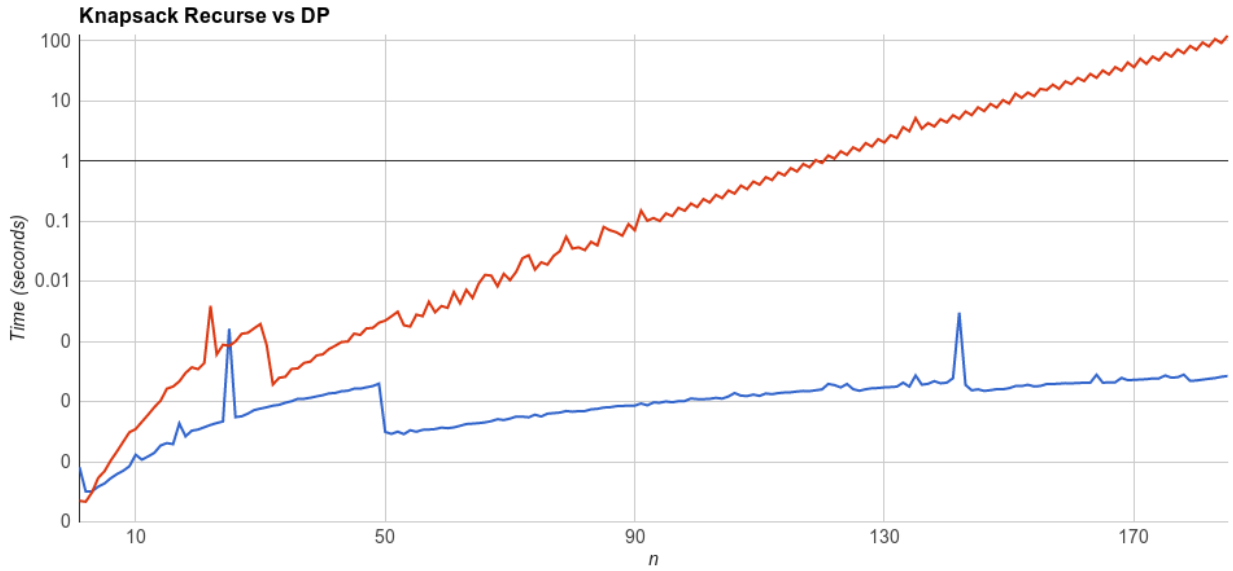
**Analysis of CountKnapsackApprox**

Firstly, line 1-3 are initializations and will take Θ(1) time. Line 4 is executed n times, line 5 takes Θ(1) to compute giving us `Θ(n)` for lines 4-5. Line 6 takes $\Theta(n^3)$ as in the assignment on

page 5. For lines 7-11, we make m iterations and do $\Theta(n)+\Theta(n)+\Theta(1)$ steps giving us runtime of $\Theta(mn)$ for the loop. However, $m=10n$ so we get $\Theta(n^2)$ for the running time of the loop on lines 7-11. Lines 12-13 take constant time to compute. Therefore, the running time is dominated by the computation time of $countKnapsackDP(a, n^2)$

**KnapsackRecurse and KnapsackDP**

Firstly, we plot time taken to compute the same solutions to the counting knapsack problem using KnapsackRecurse and KnapsackDP. Tests on both the problems with generated settings have been run. For $n$ between 0 and 185, values of $w=\{0,1,..,n\}$ and $B=n$ are considered and running times of both are plotted below.

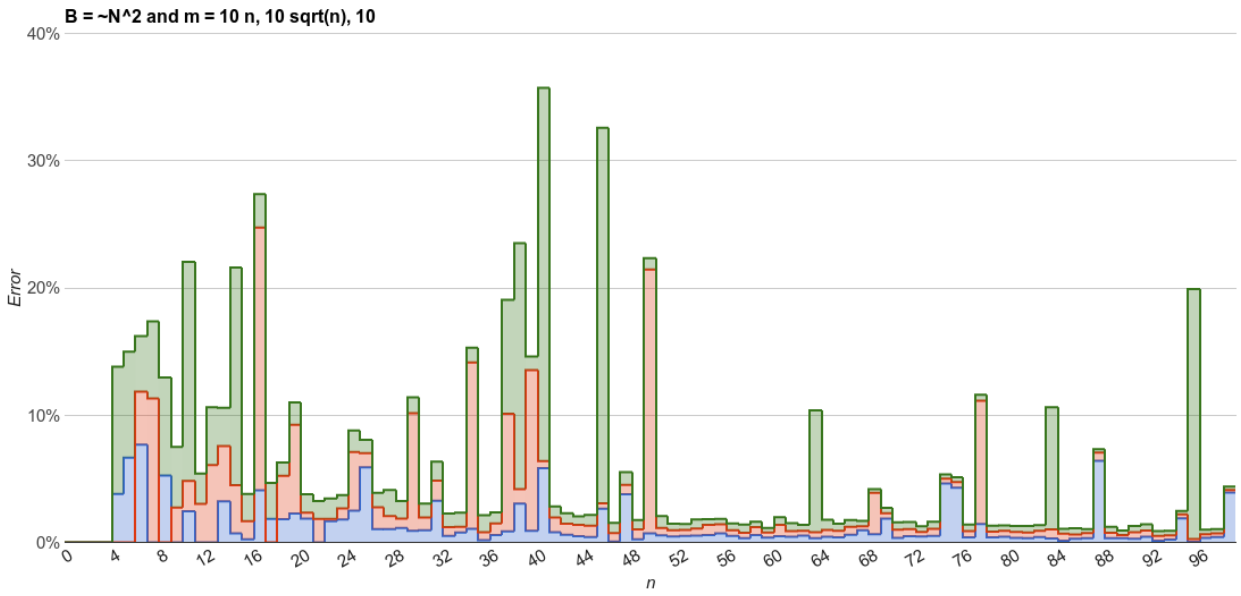



*Knapsack Recurse (red) vs DP (blue)*

Vertical axis is time taken to compute a solution as a logarithm base 10, horizontal axis is the size of n. We can observe that DP solutions performs far better than the recursive solution. To give an example, for n = 185 recursive solution takes ~104.47600 seconds while dynamic programming solution takes ~0.00032 seconds to compute. The spikes in the graph can be explained by Java's garbage collection which is run at random times in the execution of the program and cannot be directly controlled, despite that, the trend is fairly stable.

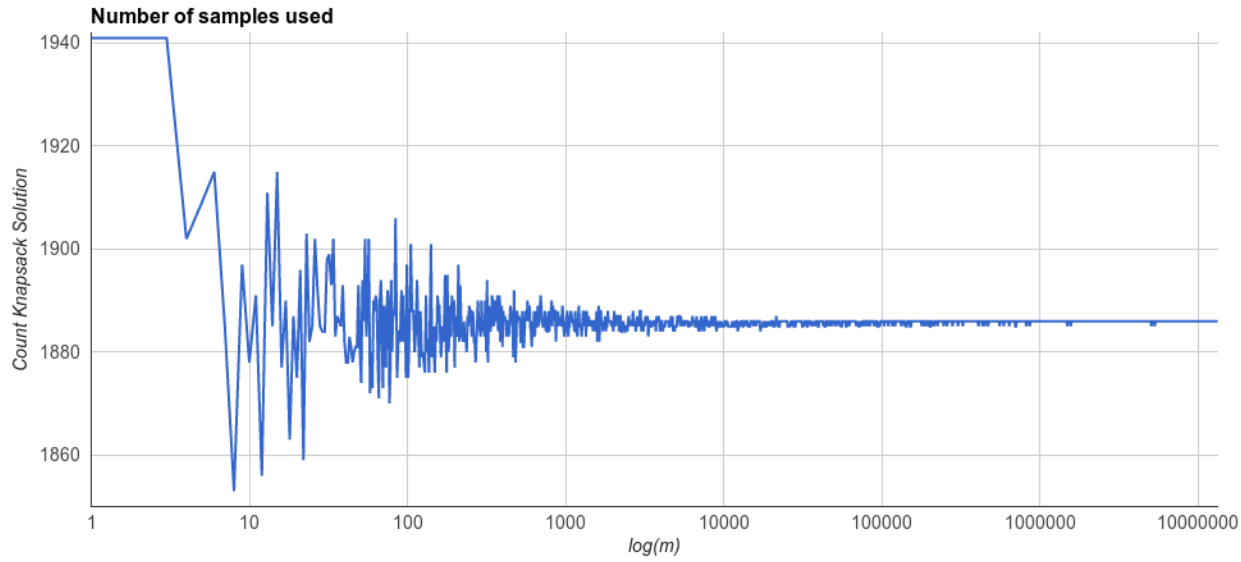
**Approximation algorithm**

Let us now consider the case of the approximation algorithm and the exact solution. The absolute error has been plotted below for various values of m. It is apparent that for the case when $m=10$ we get the highest error as the distribution is not sampled enough. Conversely, $m=10n$ performs relatively well.




*Error graph for Knapsack Approx and DP. $B=n^2*1.05$, $m=10n$ (blue), $m=10\sqrt{n}$ (red), $m=10$ (green)*

The graph below shows the results for CountKnapsackApprox as the sample size m is increased from 1 to 10 million. The values converge a single value of 1886. This shows that a sample size of 10n appears to be sufficiently large to obtain fairly accurate approximation results.




*Results of CountKnapsackApprox as m ranges from 1 to 10n = 10,000,000.*