Research Xor-Binary-Linear Operator.

We are going to understand it step by step

04 June 2017

Problem description

a) v = $\{0, 1\}^N$, for example if N =5 v can be equals to {0,1,1,0,0}
b) $v_1 + v_2$ means v_1 xor x_2
c) Let A be a generation set: $A = \{a_i \mid \ i \in 1..K, 0 < K \leq 2^N\}$. Generation means that using a_i we can get 2^Kvectors like $\sum^K{}_{i=1} \ \beta_i \, \alpha_i$, $\beta_i \in \{0, 1\}$.
d) Vectors' weight means the number of 1 inside it.
Example: W({0,1,1,0,0}) = 5

Task: for ∀ N, A plot a histogram of vectors number by their weight.

In[1743]:=
```
Xor[bd1_, bd2_] := If[bd1 ≠ bd2, 1, 0]
({#, Xor[#] /. List → Sequence}) & /@ Tuples[{0, 1}, 2] // MatrixForm
```

Out[1744]//MatrixForm=

$$\begin{pmatrix} \{0, 0\} & 0 \\ \{0, 1\} & 1 \\ \{1, 0\} & 1 \\ \{1, 1\} & 0 \end{pmatrix}$$

In[1745]:=
```
v1 = {0, 0, 1, 1, 0}
v2 = {1, 1, 1, 0, 1}
MapThread[Xor, {v1, v2}]
```

Out[1745]= {0, 0, 1, 1, 0}

Out[1746]= {1, 1, 1, 0, 1}

Out[1747]= {1, 1, 0, 1, 1}

Generation Set

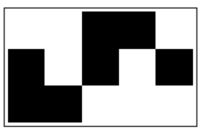
Let A be a generation set:

In[1748]:=
```
A = {
    {0, 0, 1, 1, 0},
    {1, 0, 1, 0, 1},
    {1, 1, 0, 0, 0}
  };
```

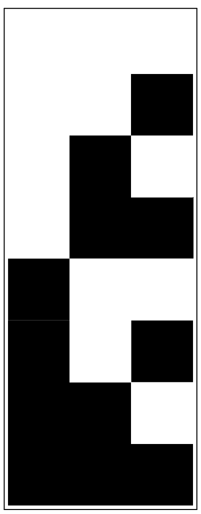
In[1749]:=
```
B = Tuples[{0, 1}, A // Length]
ArrayPlot@A
ArrayPlot@B
```

Out[1749]=
```
{{0, 0, 0}, {0, 0, 1}, {0, 1, 0}, {0, 1, 1}, {1, 0, 0}, {1, 0, 1}, {1, 1, 0}, {1, 1, 1}}
```

Out[1750]=



Out[1751]=



Call to generation set action.

Combines $a_i \in A$ in all possible ways B we get all possibles vectors for A (genA) :

In[1787]:=
```
B[[-2]]
genElem = MapThread[Times, {A, B[[-1]]}]
```

Out[1787]= {1, 1, 0}

Out[1788]= {{0, 0, 1, 1, 0}, {1, 0, 1, 0, 1}, {1, 1, 0, 0, 0}}

```
XorVec = MapThread[Xor, {#1, #2}] &;
XorVec[genElem[[1]], genElem[[2]]]
XorVec[XorVec[genElem[[1]], genElem[[2]] ], genElem[[3]] ]
(* it's Fold mechanics... *)
Fold[XorVec]@genElem
```

Out[1790]= {1, 0, 0, 1, 1}

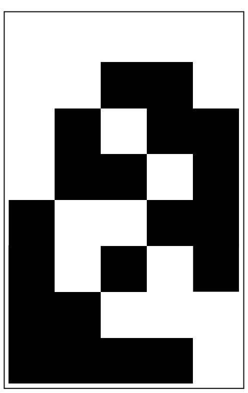
Out[1791]= {0, 1, 0, 1, 1}

Out[1792]= {0, 1, 0, 1, 1}

In[1793]:=
```
XorVec = MapThread[Xor, {#1, #2}] &;
fTimes = MapThread[Times, {#1, #2}] &;
genA = Fold[XorVec] /@ ((fTimes[A, #] &) /@ B) // Union
ArrayPlot@genA
```

Out[1795]= {{0, 0, 0, 0, 0}, {0, 0, 1, 1, 0}, {0, 1, 0, 1, 1}, {0, 1, 1, 0, 1}, {1, 0, 0, 1, 1}, {1, 0, 1, 0, 1}, {1, 1, 0, 0, 0}, {1, 1, 1, 1, 0}}

Out[1796]=



Weight calculation

Lets calculate weights of each vector in **genA**.

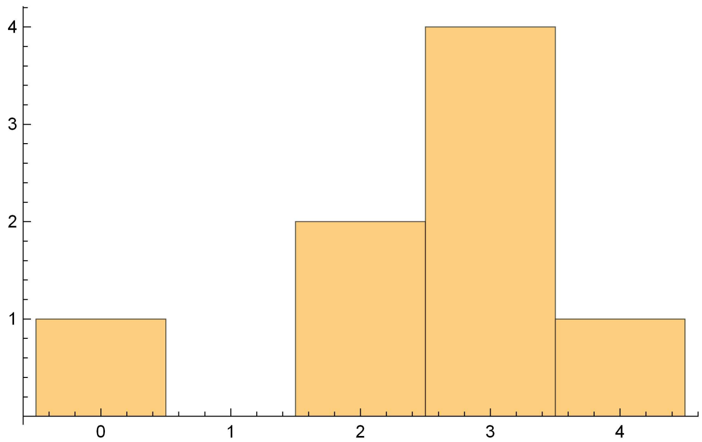
In[1797]:=
```
𝒲 = Total;
weights = 𝒲 /@ genA
```

Out[1798]= {0, 2, 3, 3, 3, 3, 2, 4}

Now we have to plot the weights histogram.

In[1799]:=
```
Histogram@weights
```

Out[1799]=



Ok, we solve the task for generation set A and N, where N=5 and A is:

In[1800]:=
```
A // MatrixForm
```

Out[1800]//MatrixForm=

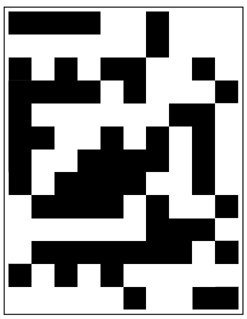
$$\begin{pmatrix} 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Generation set in wide

Lets generate more complex generations sets and see what happens.

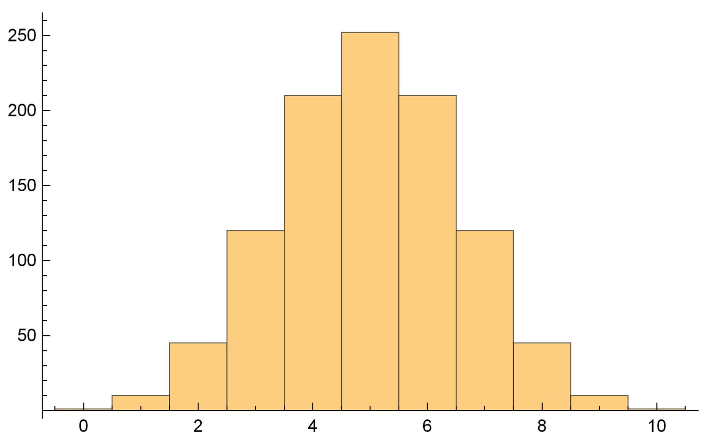
In[1801]:=
```
A = RandomChoice[{0, 1}, {13, 10}]
ArrayPlot@A
```

Out[1801]=
```
{{1, 1, 1, 1, 0, 0, 1, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
 {1, 0, 1, 0, 1, 1, 0, 0, 1, 0}, {1, 1, 1, 1, 0, 1, 0, 0, 0, 1},
 {1, 0, 0, 0, 0, 0, 0, 1, 1, 0}, {1, 1, 0, 0, 1, 0, 1, 0, 1, 0}, {1, 0, 0, 1, 1, 1, 1, 0, 1, 0},
 {1, 0, 1, 1, 1, 1, 0, 0, 1, 0}, {0, 1, 1, 1, 1, 0, 1, 0, 0, 1}, {0, 0, 0, 0, 0, 0, 1, 1, 1, 0},
 {0, 1, 1, 1, 1, 1, 1, 1, 0, 1}, {1, 0, 1, 0, 1, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 1, 0, 0, 1, 1}}
```

Out[1802]=


In[1803]:=
```
B = Tuples[{0, 1}, A // Length];
AbsoluteTiming[
 genA = Fold[𝒳orVec] /@ ((fTimes[A, #] &) /@ B) // Union;
 weights = 𝒲 /@ genA;
]
Histogram@weights
```

Out[1804]= {1.4343, Null}

Out[1805]=


Hm, Its took about 1.4 seconds to calculate the weights. Optimise it!

Optimization with Digits' Xor operation

In[1806]:=
```
AD = FromDigits[#, 2] & /@ A
```

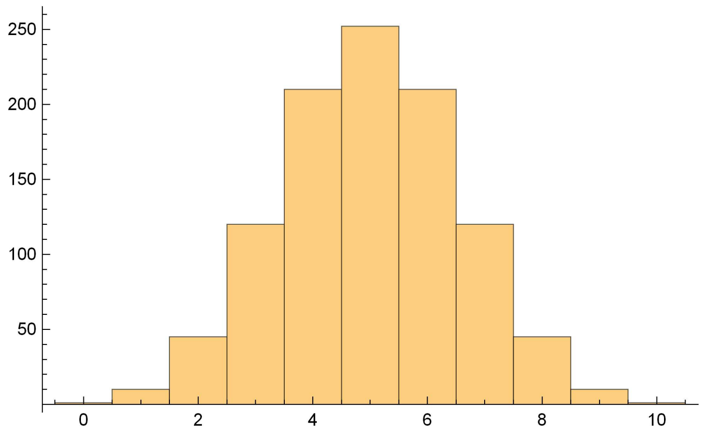
Out[1806]=
```
{968, 8, 690, 977, 518, 810, 634, 754, 489, 14, 509, 672, 19}
```

In[1807]:=
```
𝒲D = DigitCount[#, 2, 1] &;
AbsoluteTiming[
 genAD = Fold[BitXor] /@ ((Times[AD, #] &) /@ B) // Union;
 weightsD = 𝒲D /@ genAD;
]
```

Out[1808]=
```
{0.0935052, Null}
```

In[1809]:=
```
Histogram@weightsD
```

Out[1809]=



In[1810]:=
```
weights == weightsD
```

Out[1810]=
```
True
```

Fine! We make increase computation speed up to 10x-20x using digit XOR operator.

The only One function

In[1811]:=
```
A
```

Out[1811]=
```
{{1, 1, 1, 1, 0, 0, 1, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
 {1, 0, 1, 0, 1, 1, 0, 0, 1, 0}, {1, 1, 1, 1, 0, 1, 0, 0, 0, 1},
 {1, 0, 0, 0, 0, 0, 0, 1, 1, 0}, {1, 1, 0, 0, 1, 0, 1, 0, 1, 0}, {1, 0, 0, 1, 1, 1, 1, 0, 1, 0},
 {1, 0, 1, 1, 1, 1, 0, 0, 1, 0}, {0, 1, 1, 1, 1, 0, 1, 0, 0, 1}, {0, 0, 0, 0, 0, 0, 1, 1, 1, 0},
 {0, 1, 1, 1, 1, 1, 1, 1, 0, 1}, {1, 0, 1, 0, 1, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 1, 0, 0, 1, 1}}
```

In[1828]:=
```
computeW[a_] := Module[{ad, b, genAD, weights},
  ad = FromDigits[#, 2] & /@ a;
  b = Tuples[{0, 1}, a // Length];
  genAD = Fold[BitXor] /@ ((Times[ad, #] &) /@ b) // Union;
  weights = 𝒲D /@ genAD;
  weights // Tally
 ]
```

In[1829]:=
```
w = computeW[A]
```

Out[1829]=
```
{{0, 1}, {1, 10}, {2, 45}, {3, 120}, {4, 210},
 {5, 252}, {6, 210}, {7, 120}, {8, 45}, {9, 10}, {10, 1}}
```

```
weightsHistorgram = BarChart[#[[;;, 2]], ChartLabels→#[[;;,1]]]&;
weightsHistorgram@w
```

Out[1831]=

Now we have one **computeW** function for solving our task.

Touch the propability

In[1832]:=
```
w
```

Out[1832]=
```
{{0, 1}, {1, 10}, {2, 45}, {3, 120}, {4, 210},
 {5, 252}, {6, 210}, {7, 120}, {8, 45}, {9, 10}, {10, 1}}
```

In[1839]:=
```
Transpose[{{0, 1}, {1, 10}, {2, 45}, {3, 120},
   {4, 210}, {5, 252}, {6, 210}, {7, 120}, {8, 45}, {9, 10}, {10, 1}}]
```

Out[1839]=
```
{{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, {1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1}}
```

In[1850]:=
```
UnTally = Fold[Join][(PadLeft[{}, #[[2]], #[[1]]]) & /@ #] &;
```

In[1870]:=
```
wAll = UnTally [w];
wAll[[ ;; 400]]
```

Out[1871]=
```
{0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3,
 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5}
```

In[1866]:=
```
pDistrib = EstimatedDistribution[wAll, NormalDistribution[α, β]]
```

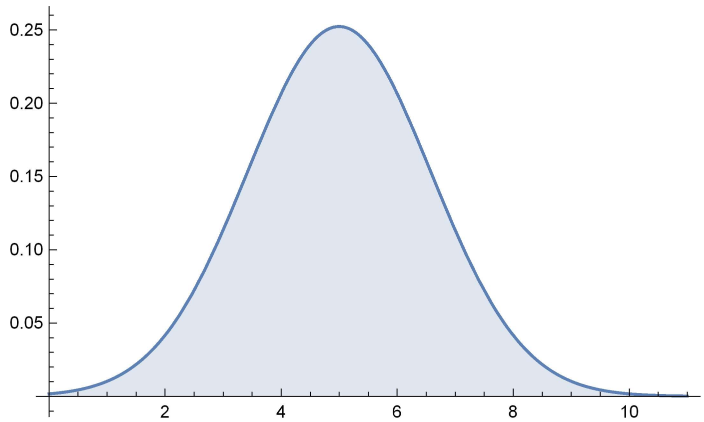
Out[1866]=
```
NormalDistribution[5., 1.58114]
```

Plot estimated Normal Distribution

Let check the normal distribution likeness.

In[1876]:=
```
Plot[PDF[pDistrib, x] // Evaluate, {x, 0, Length@w}, Filling → Axis]
DistributionFitTest[wAll, pDistrib]
```

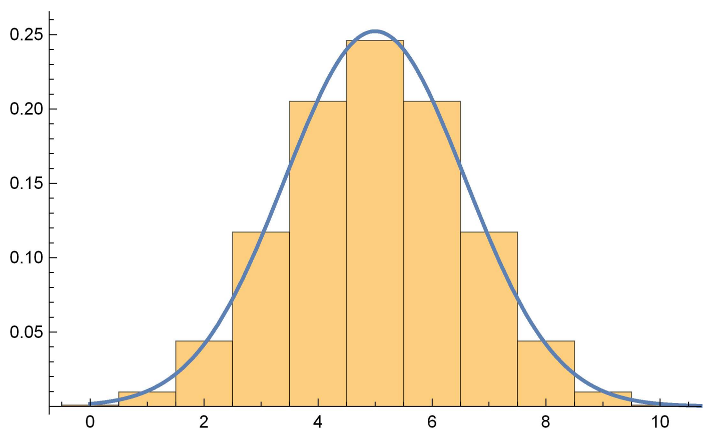
Out[1876]=



Out[1877]= 4.32956×10^{-8}

In[1878]:=
```
Show[Histogram[wAll, Automatic, "ProbabilityDensity"],
 Plot[PDF[pDistrib, x], {x, 0, Length@w}, PlotStyle → Thick]]
```

Out[1878]=



In[1910]:=
```
weightsDistribHistorgram[w_] := Module[{wAll, pDistrib},
   wAll = UnTally [w];
   pDistrib  = EstimatedDistribution[wAll, NormalDistribution[α, β]];
   Show[Histogram[wAll, Automatic, "ProbabilityDensity"],
    Plot[PDF[pDistrib, x], {x, 0, Max@wAll}, PlotStyle → Thick]]
  ];
```

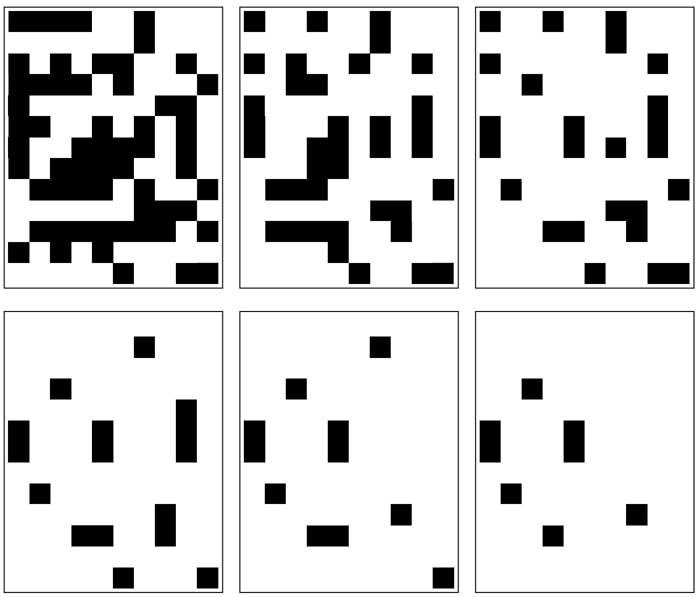
Increase zeros or ones in generation set

ToPValue replaced original **v** with **v2** with p probability.

In[1883]:=
```
ToPValue[v_, v2_: 0, p_: 0.5] := If[Random[] ≤ p, v2, v]
```

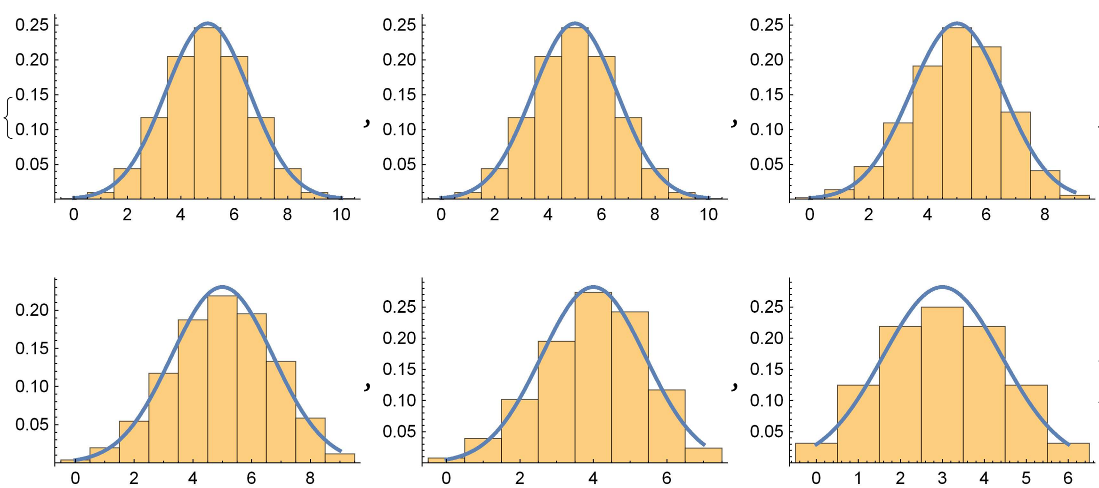
In[1952]:=
```
ANew = NestList[Map[ToPValue[#, 0, 0.3] &, #, {2}] &, A, 5];
GraphicsGrid[Partition[ArrayPlot /@ ANew, 3]]
```

Out[1953]=



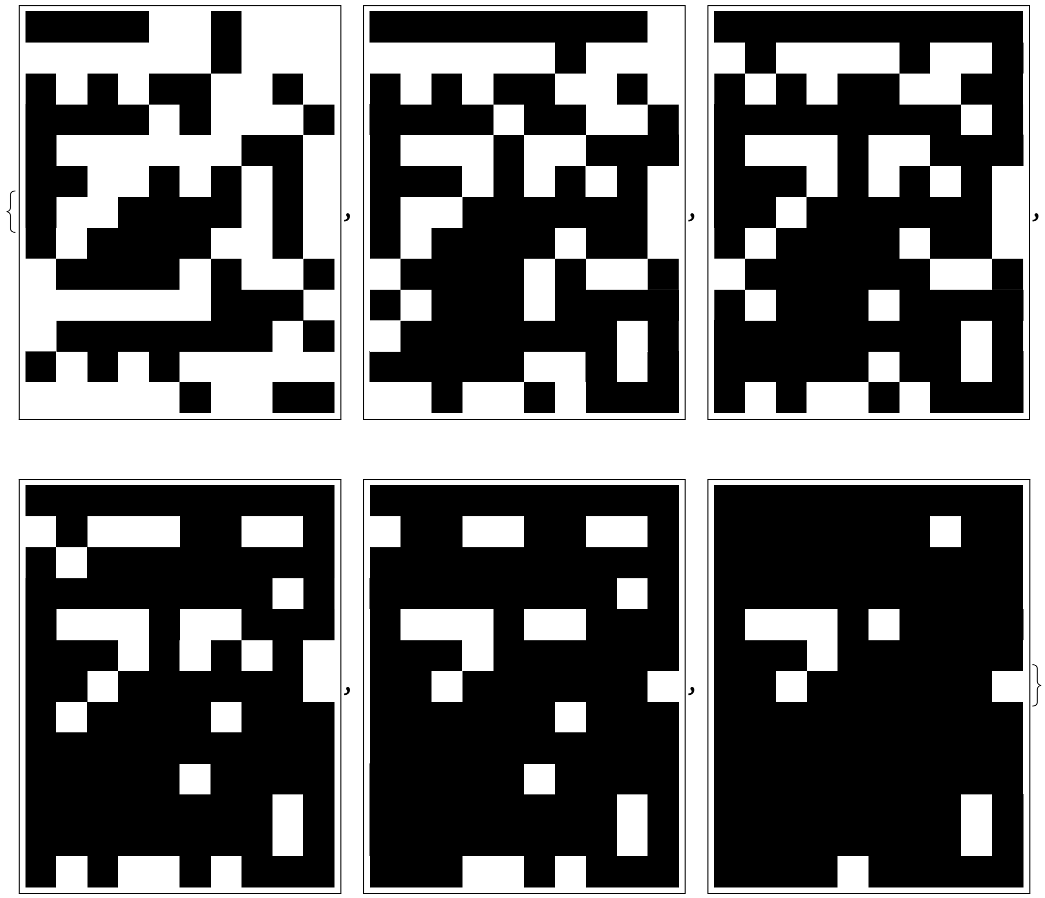
In[1954]:=
```
weightsDistribHistorgram /@ (computeW /@ ANew)
```

Out[1954]=



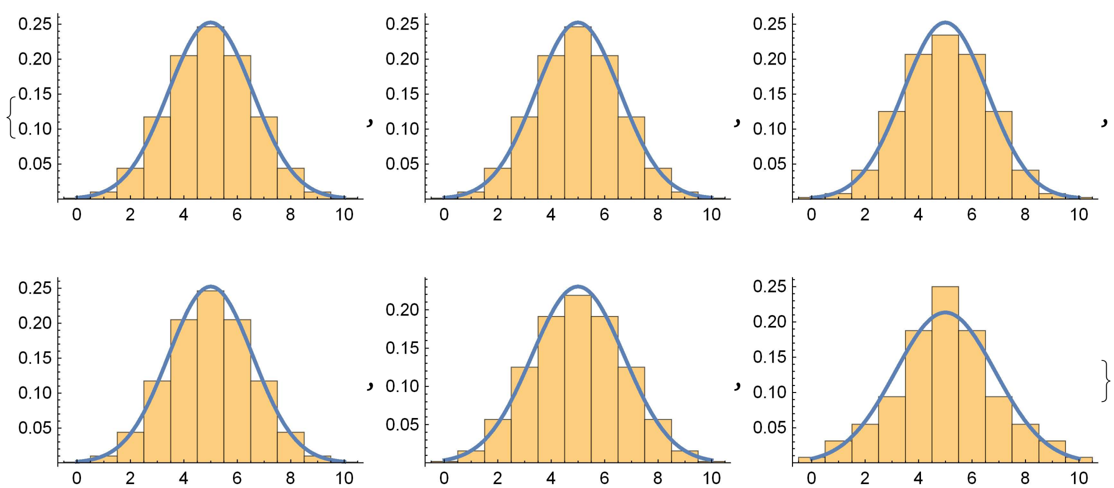
In[1955]:=
```
ANew = NestList[Map[ToPValue[#, 1, 0.3] &, #, {2}] &, A, 5];
ArrayPlot /@ ANew
```

Out[1956]=



In[1957]:=
```
weightsDistribHistorgram /@ (computeW /@ ANew)
```

Out[1957]=

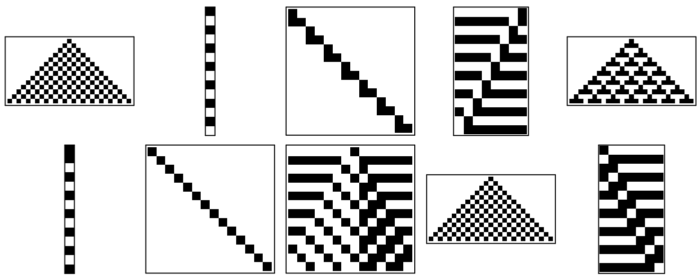


Nothing interesting.

Lets add CellularAutomaton

In[1962]:=
```
cellA = CellularAutomaton[#, {{1}, 0}, 13] & /@ Range[50, 60];
GraphicsGrid[Partition[ArrayPlot /@ cellA, 5]]
```

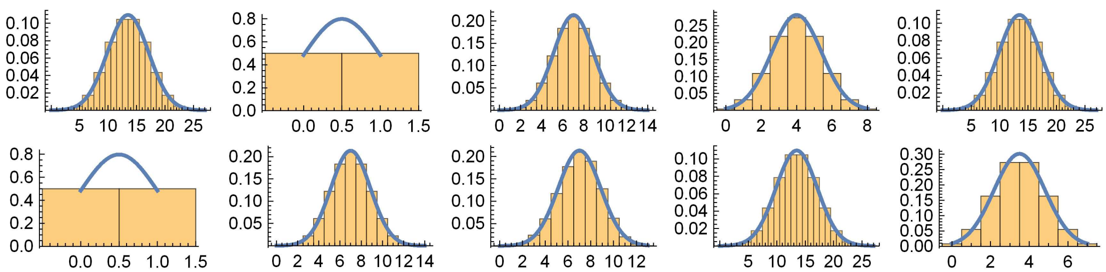
Out[1963]=



In[1964]:=
```
cw = computeW /@ cellA;
```

In[1966]:=
```
GraphicsGrid[Partition[weightsDistribHistorgram /@ cw, 5]]
```

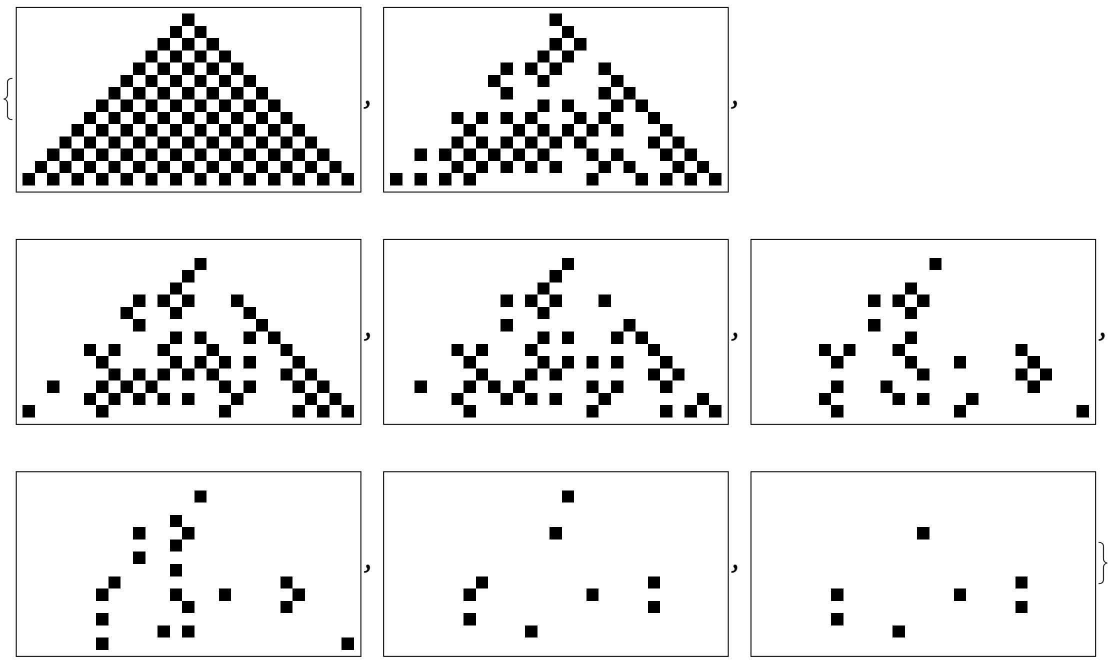
Out[1966]=



In[1970]:=
```
ANew = NestList[Map[ToPValue[#, 0, 0.3] &, #, {2}] &, cellA[[1]], 7];
```

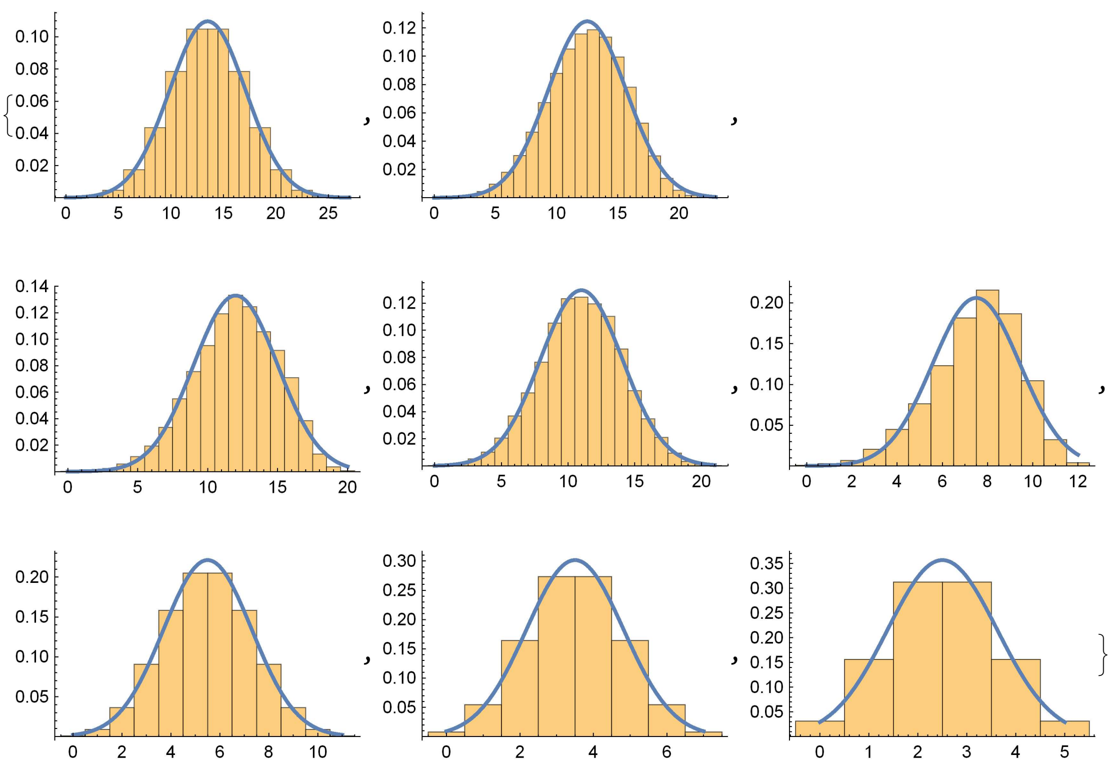
In[1971]:=
```
ArrayPlot /@ ANew
```

Out[1971]=



In[1972]:=
```
weightsDistribHistorgram /@ (computeW /@ ANew)
```

Out[1972]=



Good. We get not a Normal Distribution using cell auto basis.

Map – Reduce method

Now lets make classic parallelization using Map-Reduce method. Let see on computeW function:

```
computeW[a_] := Module[{ad, b, genAD, weights},
  ad = FromDigits[#, 2] & /@ a;
  b = Tuples[{0, 1}, a // Length];
  genAD = Fold[BitXor] /@ ((Times[ad, #] &) /@ b) // Union;
  weights = 𝒲D /@ genAD;
  weights // Tally
 ]
```

```
In[2112]:= computeW2[a_] := Module[{ad, b1, b2, genAD, genAD2, weights, weights2},
  ad = FromDigits[#, 2] & /@ a;
  b1 = Join[{0}, #] & /@ Tuples[{0, 1}, (a // Length) - 1];
  b2 = Join[{1}, #] & /@ Tuples[{0, 1}, (a // Length) - 1];

  genAD = Fold[BitXor] /@ ((Times[ad, #] &) /@ b1) // Union;
  genAD2 = Fold[BitXor] /@ ((Times[ad, #] &) /@ b2) // Union;

  weights = 𝒲D /@ genAD;
  weights2 = 𝒲D /@ genAD2;

  {{genAD, weights}, {genAD2, weights2}}
 ]
```

```
In[2113]:= A = RandomChoice[{0, 1}, {7, 6}]
w = computeW2[A]

Out[2113]= {{0, 1, 0, 1, 0, 0}, {1, 0, 0, 1, 0, 1}, {0, 0, 0, 0, 0, 0},
 {0, 0, 1, 1, 0, 1}, {0, 0, 1, 1, 1, 1}, {0, 0, 1, 0, 1, 0}, {1, 0, 1, 0, 1, 0}}

Out[2114]= {{{0, 2, 5, 7, 8, 10, 13, 15, 32, 34, 37, 39, 40, 42, 45, 47},
  {0, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 2, 3, 4, 5}},
 {{17, 19, 20, 22, 25, 27, 28, 30, 49, 51, 52, 54, 57, 59, 60, 62},
  {2, 3, 2, 3, 3, 4, 3, 4, 3, 4, 3, 4, 4, 5, 4, 5}}}
```

```
In[2117]:= A = RandomChoice[{0, 1}, {7, 6}]
w = computeW2[A]

Out[2117]= {{0, 0, 0, 1, 0, 0}, {1, 0, 1, 1, 0, 0}, {1, 1, 1, 1, 1, 1},
 {0, 1, 0, 1, 1, 1}, {1, 1, 1, 1, 1, 1}, {1, 0, 0, 0, 0, 0}, {1, 1, 1, 1, 1, 1}}

Out[2118]= {{{0, 4, 8, 12, 19, 23, 27, 31, 32, 36, 40, 44, 51, 55, 59, 63},
  {0, 1, 1, 2, 3, 4, 4, 5, 1, 2, 2, 3, 4, 5, 5, 6}},
 {{0, 4, 8, 12, 19, 23, 27, 31, 32, 36, 40, 44, 51, 55, 59, 63},
  {0, 1, 1, 2, 3, 4, 4, 5, 1, 2, 2, 3, 4, 5, 5, 6}}}
```

Reduce detail in Map – Reduce.

We have to work with XOR-linear independence basis vectors to make map-reduce effective. The last example shows us that we have to join weights checking if numbers(vectors) are not the same.

```
(*In our case it will be useful to calc only
 basic vectors and find their WEIGHTS on the last step*)
mergeW2[gen1_, gen2_] := Module[{gen, w},
  gen = gen1[[1]] ~ Join ~ gen2[[1]] // Union;
  {gen, 𝒲D /@ gen}
 ]
mergeW2[w[[1]], w[[2]]]
```

```
Out[2124]= {{0, 4, 8, 12, 19, 23, 27, 31, 32, 36, 40, 44, 51, 55, 59, 63},
  {0, 1, 1, 2, 3, 4, 4, 5, 1, 2, 2, 3, 4, 5, 5, 6}}
```

FIX: to fix w recalculation

ComputeW function have to work with any basis vectors prefix. Not only {0} or {1}

```
In[2157]:= computeWpre[a_, prefix_: {}] := Module[
  {l = Length@prefix, lneed = Length@a, b = {}, times = {}, time = 0, genAD = {}, w = {}, ad},
  ad = FromDigits[#, 2] & /@ a;
  b = Join[prefix, #] & /@ Tuples[{0, 1}, lneed - l];
  genAD = Map[Fold[BitXor], Map[Times[ad, #] &, b]];
  genAD = genAD // Union // Sort;
  w = 𝒲D /@ genAD;
  {genAD, w}
 ]
A = RandomChoice[{0, 1}, {7, 6}];
computeWpre[A] == (computeWpre[A, {1}] ~ mergeW2 ~ computeWpre[A, {0}])
```

```
Out[2159]= True
```

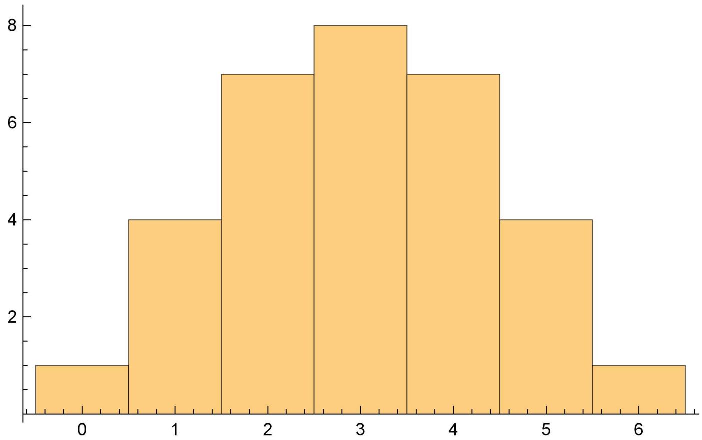
```
In[2153]:= bPrefixs = Tuples[{0, 1}, 3]
```

```
Out[2153]= {{0, 0, 0}, {0, 0, 1}, {0, 1, 0}, {0, 1, 1}, {1, 0, 0}, {1, 0, 1}, {1, 1, 0}, {1, 1, 1}}
```

In[2164]:=
```
{numbers, w} = Fold[mergeW2][Parallelize[computeWpre[A, #] & /@ bPrefixs]]
Histogram@w
```

Out[2164]=
```
{{0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28,
  30, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63},
 {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6}}
```

Out[2165]=



We used **Parallelize** to compute basis vectors with prefix on independent nodes (if possible).

Parallelize Map – Reduce

In[2178]:=
```
computeParallelMapReduce[fmap_, freduce_, data_, prefix_: {{}}] :=
 Fold[freduce,
  Parallelize[Map[fmap[data, #] &, prefix]]
 ]
```

In[2169]:=
```
computeParallelMapReduce[computeWpre, mergeW2, A, bPrefixs]
```

Out[2169]=
```
{{0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28,
  30, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63},
 {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6}}
```

In[2185]:=
```
computeParallelMapReduce[computeWpre, mergeW2, {{}}]
```

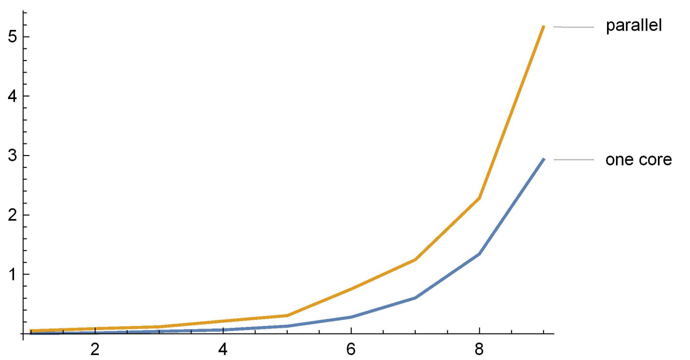
Out[2185]=
```
{{0}, {0}}
```

If there is any computational acceleration in such realization?

```
times = {};
For[i = 1, i < 10, i++,
 time = {};
 A = RandomChoice[{0, 1}, {9 + i, 5 + i}] // Union;
 res = computeW[A] // AbsoluteTiming;
 time = AppendTo[time, res[[1]]];
 res = computeParallelMapReduce[computeWpre, mergeW2, A, bPrefixs] // AbsoluteTiming;
 time = AppendTo[time, res[[1]]];
 times = AppendTo[times, time];
 (*Print[i];*)
]
```

In[2210]:=
```
ListLinePlot[times // Transpose, PlotRange → All, PlotLabels → {"one core", "parallel"}]
```

Out[2210]=



Our parallel implementation using left **fold** and **merging** is slower then “one core” computeW. **FIX in the future.**
To optimise we need to make linear-xor independent basis and then parallize the computing without mention same numbers at different Map executions.

Reading input file

In[2255]:=
```
fileIn = "E:\\data\\in.txt";
fileOut = "E:\\data\\out.txt"

dataStr = ReadList[fileIn, String]
A = (Interpreter["Number"] /@ Characters@# &) /@ dataStr
```

Out[2256]= E:\data\out.txt

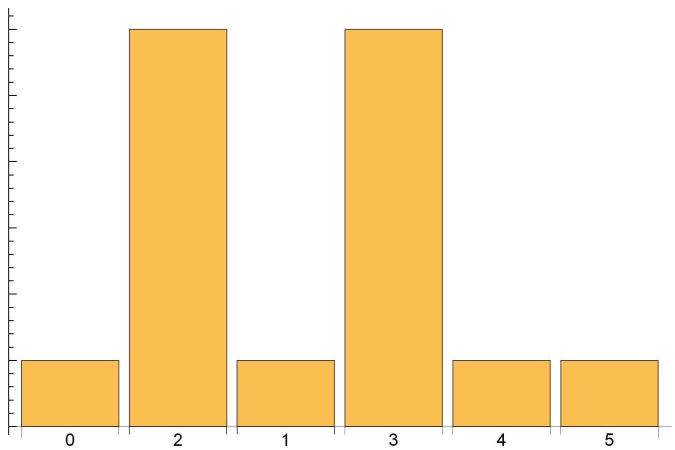
Out[2257]= {000101, 010100, 110000, 111000}

Out[2258]= {{0, 0, 0, 1, 0, 1}, {0, 1, 0, 1, 0, 0}, {1, 1, 0, 0, 0, 0}, {1, 1, 1, 0, 0, 0}}

In[2260]:=
```
w = computeW[A]
weightsHistorgram@w
```

Out[2260]= {{0, 1}, {2, 6}, {1, 1}, {3, 6}, {4, 1}, {5, 1}}

Out[2261]=



In[2269]:=
```
w = SortBy[w, Last] // Reverse
```

Out[2269]= {{3, 6}, {2, 6}, {5, 1}, {4, 1}, {1, 1}, {0, 1}}

```
stream = OpenWrite[fileOut];
WriteString[stream,
    StringTemplate["`w`\t`freq`\n"][<|"w" → #[[1]], "freq" → #[[2]]|>]] & /@ w;
Close[
 fileOut]
```

Out[2278]= E:\data\out.txt

In[2280]:=
```
ReadList[fileOut, String]
```

Out[2280]= {3	6, 2	6, 5	1, 4	1, 1	1, 0	1}

One page code

In[151]:=
```
fileIn = "E:\\data\\in.txt";
fileOut = "E:\\data\\out.txt";

dataStr = ReadList[fileIn, String];
A = (Interpreter["Number"] /@ Characters@# &) /@ dataStr;

𝒲D = DigitCount[#, 2, 1] &;
computeW[a_] := Module[{ad, b, genAD, weights},
   ad = FromDigits[#, 2] & /@ a;
   b = Tuples[{0, 1}, a // Length];
   genAD = Fold[BitXor] /@ ((Times[ad, #] &) /@ b) // Union;
   weights = 𝒲D /@ genAD;
   weights // Tally
  ]

w = computeW[A];
w = SortBy[w, Last] // Reverse;

stream = OpenWrite[fileOut];
WriteString[stream,
     StringTemplate["`w`\t`freq`\n"][<|"w" → #[[1]], "freq" → #[[2]]|>]] & /@ w;
Close[fileOut]
```

Out[161]= E:\data\out.txt

THE END

Optimization

1) Find xor-linear independent basis of A and use it instead of A.

2) Don’t use Fold in parallel version.

3) Reduce function in Map-Reduce have to know that we used independent basis.

4) If we have input basis with lot of “0” and few “1” we need to migrate to sparse arrays usage.

Algorithmic complexity

Let the source basis A is xor-linear independent and have K vectors with size N.
B have 2^K vectors.

Fair BitXor have O(N) comlexity .

The most complex function is:
genAD=Fold[BitXor]/@((Times[ad,#]&)/@b);

have O ((K*N)*(2^K * N)) complexity which equal to $O(N^2 * K * 2^K)$
Fine parallel version with P independent nodes will works at $O(N^2 * K * 2^K * (\log(P)/ P))$.

If vector size is constant then N^2 is *a* constant *C*. Result complexity = $O(K*2^K)$.
If vector basis size is constant then *K* is *a* constant *C*. Result complexity = $O(N^2)$.