```
QuotientToMatrix[gens_, d_] :=
 (* Returns a matrix M such that s(M) = <gens>/d,
 using the proofs of 2.1 and 2.5 *)
 Module[{len, M1, Diag},
  Catch[
   len = Length[gens];
   M1 = Inverse[SmithDecomposition[{gens}][[3]]];
   (* M1 will be such that s(M1)=<gens>; see 2.5 *)
   Diag = DiagonalMatrix[
     Join[{1}, Table[d, {i, 1, len - 1}]]];
   Throw[Diag.M1]
  ]
 ]
```

In[•]:=
```
MatrixForm[QuotientToMatrix[{21, 23}, 2]]
MatrixForm /@
 SmithDecomposition[QuotientToMatrix[{21, 23}, 2]]
```

Out[•]//MatrixForm=

$$\begin{pmatrix} 21 & 23 \\ -22 & -24 \end{pmatrix}$$

Out[•]=

$$\left\{\begin{pmatrix} -1 & -1 \\ 22 & 21 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}, \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}\right\}$$

In[•]:=
```
AddMatrices[M1_, M2_] :=
 (* Computes M such that s(M) = s(M1)+s(M2),
 using the proof of 1.7 *)
 Module[{l1, M},
  Catch[
   l1 = Dimensions[M1][[1]];
   M = ArrayFlatten[{{M1, 0}, {0, M2}}];
   (* Creates the block matrix;
   pi will be the sum of the 1st and (l1+1)st
    coordinates *)
   M[[1]] = M[[1]] + M[[1 + l1]];
   (* Transformation that allows pi to be
    projection onto the first coordinate *)
   Throw[M]
  ]
 ]
```

In[•]:=
```
M1 = QuotientToMatrix[{23, 25}, 2];
M2 = QuotientToMatrix[{29, 31}, 2];
M = AddMatrices[M1, M2];
MatrixForm[M]
MatrixForm /@ SmithDecomposition[M]
```

Out[•]//MatrixForm=

$$\begin{pmatrix} 23 & 25 & 29 & 31 \\ -24 & -26 & 0 & 0 \\ 0 & 0 & 29 & 31 \\ 0 & 0 & -30 & -32 \end{pmatrix}$$

Out[•]=

$$\left\{\begin{pmatrix} -1 & -1 & 1 & 0 \\ 0 & 0 & -1 & -1 \\ -24 & -23 & 22 & -2 \\ -24 & -23 & 54 & 29 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}, \begin{pmatrix} 1 & -1 & 1 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 1 & -1 & 1 \end{pmatrix}\right\}$$

In[•]:=
```
CheckElt[list_, i_] :=
  Length[FrobeniusSolve[list, i, 1]] == 1;
(* Returns Boolean of whether i is an element
  of the semigroup <list> *)
```

In[•]:=
```
CheckElt[{3, 4}, 5]
CheckElt[{3, 4}, 7]
```

Out[•]= False

Out[•]= True

```
FindQuotient[M_, gens_, r_, tries_] :=
 (* Applies main theorem to compute s(M) as a
  quotient. r is the blow up factor used in
  Lemma 4.5. tries is the number of times it
  will try a random matrix before giving up. gens
  must be the generators of s(M) *)
 Catch[
  Module[{Frob, psFrob, semigroup, len, tot, M2,
    M3, M4, M5, qgens, d, newsemigroup},
   Frob = FrobeniusNumber[gens];
   psFrob = {};
   (* The following loop will compute the pseudo-
     Frobenius number for <gens> *)
   Do[
    If[! CheckElt[gens, i] &&
      Apply[And, Table[CheckElt[gens, i + j],
        {j, gens}]], psFrob = Append[psFrob, i]],
    {i, 1, Frob}];
   checklist = Join[gens, psFrob];
   (* checklist includes the generators and pseudo-
```

```
 Frobenius numbers, that is,
everything that must be checked to see if s
  (M)=<gens> *)
semigroup = Table[CheckElt[gens, i],
  {i, checklist}];
(*semigroup is a list of Booleans of whether
  that element of checklist is in the
  semigroup
  (it will be True for the generators and
      False for the pseudo-Frobenius numbers).
   Another semigroup will be the same as <
 gens> iff its corresponding table is
  identical *)
len = Dimensions[M][[1]];
tot = M.Transpose[Table[1, {i, 1, len}]];
M2 = r M - Transpose[Table[tot, {i, 1, len}]];
(* M2 is the Matrix M_r in Lemma 4.5 *)
M3 = SmithDecomposition[M2][[2]][[len, len]] *
  Inverse[M2];
(* M3 is the smallest multiple of M2^(-1)
    that is integral...  the proof uses adj(M2)=
 det(M2)*M2^(-1), but this is smaller *)
Do[
 M4 = M3 + Table[RandomInteger[{0, 1}], {i, 1, len},
    {j, 1, len}];
 (* M4 is the matrix B in the main proof
  (last few paragraphs of manuscript) that
  we hope has SNF (1,...,1,d) *)
 M5 = Det[M4] * Inverse[M4];
 (*M5=adj(M4) is A in the main proof... we
    hope s(M5)=<gens>*)
```

```
    qgens = M5[[1]];
    d = SmithDecomposition[M5][[2]][[len, len]];
    (* if Theorem 2.8 hypothesis holds,
    s(M5)=<qgens>/d *)
    If[Min[qgens] > 0 && Apply[GCD, qgens] == 1,
     (* Checking that 2.8 hypothesis holds *)
     newsemigroup = Table[CheckElt[qgens, i * d],
       {i, checklist}];
     If[semigroup == newsemigroup,
      (* Checking that s(M5)=<gens>/d *)
      Throw[StringJoin["It is ", ToString[qgens],
        " divided by ", ToString[d], "."]]],
     {i, 1, tries}
    ];
    Throw["Fail"]
   ]
  ]

gens = {23, 24, 25, 29, 30, 31};
r = 150;
tries = 10 000;
FindQuotient[M, gens, r, tries]
```

Out[•]= It is {13775465, 14996610,
18887728, 20196837} divided by 109340422.

```
FindQuotientBrute[gens_, k_, d_] :=
 (* Exhaustive search to decide if <gens>
  can be written as some
```

```
 <a1,a2,...,am>/d for m ≤ k. If yes,
it prints one solution and returns "True". If no,
it returns "False". For maximum efficiency,
gens should be written in increasing order. *)
Catch[
 Module[{a, Frob, psFrob},
  Frob = FrobeniusNumber[gens];
  psFrob = {};
  (* The loop below will create a list of all
     pseudo-Frobenius numbers for <
   gens>. A semigroup will be equal to <gens>
   iff it includes each element of gens and
    doesn't include any element of psFrob *)
  Do[
   If[! CheckElt[gens, i] &&
     Apply[And, Table[CheckElt[gens, i + j],
       {j, gens}]], psFrob = Append[psFrob, i]],
   {i, 1, Frob}];
  Do[ (* This loop will search through all
    options for the smallest generator, a1,
   of the quotient semigroup. The RunLoop
     function will allow us to recursively
     nest loops to check all possible <a1,
   a2,...,ak>/d. *)
   If[RunLoop[gens, psFrob, k, d, {a}], Throw[True]],
   {a, gens[[1]], d * gens[[1]]}
  ];
  Throw[False]
 ]
]

RunLoop[gens_, psFrob_, k_, d_, currentGens_] :=
```

```
(* This function calls itself recursively. Given
   a potential subset of generators currentGens=<a1,
 a2,...,am}, with m ≤ k,
this function returns True (and prints a solution)
   if currentGens can be extended to up to k
   generators to obtain <gens> = <a1,a2,...,
ak>/d. Otherwise, it returns False *)
Catch[
 Module[{a, flag, aMax, gen},
  If[Apply[Or, Table[CheckElt[currentGens, i*d],
     {i, psFrob}]], Throw[False]];
  (* The above checks if any of the pseudo-
   Frobenius numbers of <gens> are in <
   currentGens>/d. If so,
  it is impossible to extend to a correct quotient,
  so it returns False and doesn't nest more
   loops *)
  flag = True;
  Do[ If[! CheckElt[currentGens, d*gen],
    aMax = d*gen;
    flag = False; Break[]],
   {gen, gens}];
  (* If one of the generators, gen,
  in gens is /not/ in <currentGens>/d,
  then a correct quotient must have an additional
    generator that is at most aMax= d*gen,
  because somehow d*gen must be obtained in
    the numerator of
   <a1,a2,...ak>/d. This sets the upper bound
    on the nested loop below that will look
    for this additional generator. *)
```

```
 If[flag, Print[currentGens, " / ", d];
  Throw[True]];
 (* If flag == True,
 then every element of gens is in <currentGens>/d,
 and we already know that none of the pseudo-
  Frobenius numbers are, so this would mean that
   <gens> = <currentGens>/d has been found. So
    it prints the result and stops the looping *)
 If[k == Length[currentGens], Throw[False],
  (* If currentGens already has k elements,
  then we've failed to find a quotient and
   don't nest any more loops *)
  Do[
   (* Otherwise, we test all possible values,
   a, that we might append to currentGens to
    get the correct answer.  Since we assume
    we are writing the generators of the
    quotient in increasing order,
   the lower bound for a is the largest
    element of currentGens plus 1,
   and the upper bound is aMax as determined
    above *)
   If[RunLoop[gens, psFrob, k, d,
     Append[currentGens, a]], Throw[True]],
   {a, Last[currentGens] + 1, aMax}]
  (* We nest a new loop inside by calling
   RunLoop. If it returns a correct answer,
  then we stop. *)
 ];
 Throw[False]
]
```

```
]
```

```
FindQuotientBrute[{3, 4, 5}, 2, 2]
```

```
{3, 5} / 2
```

Out[•]= True

```
FindQuotientBrute[{5, 11, 12, 13}, 3, 2]
```

```
{10, 11, 13} / 2
```

Out[•]= True

```
FindQuotientBrute[{23, 24, 25, 29, 30, 31, 32}, 4, 6]
```

```
{58, 64, 69, 75} / 6
```

Out[•]= True

```
Do[Print[d];
 Print[FindQuotientBrute[{6, 7, 8}, 2, d]],
 {d, 2, 7}]
```

```
2
```

```
False
```

```
3
```

```
False
```

```
4
```

```
False
```

```
5
```

```
False
```

```
6
```

```
False
```

```
7
```

```
{6, 25} / 7
```

```
True
```

```
Do[Print[d];
 Print[FindQuotientBrute[{23, 24, 25, 29, 30, 31},
   4, d]], {d, 2, 10}]
```

```
2
False
3
False
4
False
5
False
6
False
7
False
8
False
9
False
10
False
```