Some details about the calculation of Kazhdan-Lusztig polynomials for split E_8

Marc van Leeuwen

Laboratoire de Mathématiques et Applications
Université de Poitiers

March 19, 2007 / Atlas workshop, MIT

Outline

Outline

How Big is that matrix?

- A coarse upper bound
  453060×453060 entries, each a polynomial of degree < 32 whose integral coefficients fit in 4 bytes.
  So $453060^2 \times 32 \times 4 \approx 26$ trillion bytes suffice.
- A coarse lower bound
  Some 6 billion "interesting" matrix entries, 4 bytes needed for each to distinguish their values: ≈ 24 billion bytes
  More than 1 billion distinct polynomials, average size > 10: more than 10 billion coefficients
  So at least about 34 billion bytes seem necessary

$2^{32} \approx 4.3$ billion < 34 billion.
So a 32-bit computer cannot store the matrix

How Big is that matrix?

- A coarse upper bound
  453060×453060 entries, each a polynomial of degree < 32 whose integral coefficients fit in 4 bytes.
  So $453060^2 \times 32 \times 4 \approx 26$ trillion bytes suffice.
- A coarse lower bound
  Some 6 billion "interesting" matrix entries, 4 bytes needed for each to distinguish their values: ≈ 24 billion bytes
  More than 1 billion distinct polynomials, average size > 10: more than 10 billion coefficients
  So at least about 34 billion bytes seem necessary

$2^{32} \approx 4.3$ billion < 34 billion.
So a 32-bit computer cannot store the matrix

How Big is that matrix?

- A coarse upper bound
  453060×453060 entries, each a polynomial of degree < 32 whose integral coefficients fit in 4 bytes.
  So $453060^2 \times 32 \times 4 \approx 26$ trillion bytes suffice.
- A coarse lower bound
  Some 6 billion "interesting" matrix entries, 4 bytes needed for each to distinguish their values: ≈ 24 billion bytes
  More than 1 billion distinct polynomials, average size > 10: more than 10 billion coefficients
  So at least about 34 billion bytes seem necessary

$2^{32} \approx 4.3$ billion < 34 billion.
So a 32-bit computer cannot store the matrix

How Big is that matrix?

- A coarse upper bound
  453060×453060 entries, each a polynomial of degree < 32 whose integral coefficients fit in 4 bytes.
  So $453060^2 \times 32 \times 4 \approx 26$ trillion bytes suffice.
- A coarse lower bound
  Some 6 billion "interesting" matrix entries, 4 bytes needed for each to distinguish their values: ≈ 24 billion bytes
  More than 1 billion distinct polynomials, average size > 10: more than 10 billion coefficients
  So at least about 34 billion bytes seem necessary

$2^{32} \approx 4.3$ billion < 34 billion.
So a 32-bit computer cannot store the matrix

How Big is that matrix?

- A coarse upper bound
  453060×453060 entries, each a polynomial of degree < 32 whose integral coefficients fit in 4 bytes.
  So $453060^2 \times 32 \times 4 \approx 26$ trillion bytes suffice.
- A coarse lower bound
  Some 6 billion "interesting" matrix entries, 4 bytes needed for each to distinguish their values: ≈ 24 billion bytes
  More than 1 billion distinct polynomials, average size > 10: more than 10 billion coefficients
  So at least about 34 billion bytes seem necessary

$2^{32} \approx 4.3$ billion < 34 billion.
So a 32-bit computer cannot store the matrix

It's not just about bulk storage

Other considerations:

- Deep recurrence needs data in *Random Access Memory*
- Data must be accessible. Some choices:
  - Array of items
  - Linked tree structure using pointers
  - Array of pointers
  - Array of ID numbers, identified data looked up

  Choices imply additional overhead
  - In arrays, fixed size slots waste space at small items
  - Arrays need reallocation as data grows
  - Each pointer uses 8 bytes (64 bits machine!)
  - ID Numbers may fit in 4 bytes
- Speed cannot be forgotten
  - Linear search is no option
  - Nor is linear insertion into ordered array

It's not just about bulk storage

Other considerations:

- Deep recurrence needs data in *Random Access Memory*
- Data must be accessible. Some choices:
  - Array of items
  - Linked tree structure using pointers
  - Array of pointers
  - Array of ID numbers, identified data looked up
- Choices imply additional overhead
  - In arrays, fixed size slots waste space at small items
  - Arrays need reallocation as data grows
  - Each pointer uses 8 bytes (64 bits machine!)
  - ID Numbers may fit in 4 bytes
- Speed cannot be forgotten
  - Linear search is no option
  - Nor is linear insertion into ordered array

It's not just about bulk storage

Other considerations:

- Deep recurrence needs data in *Random Access Memory*
- Data must be accessible. Some choices:
  - Array of items
  - Linked tree structure using pointers
  - Array of pointers
  - Array of ID numbers, identified data looked up

  Choices imply additional overhead
  - In arrays, fixed size slots waste space at small items
  - Arrays need reallocation as data grows
  - Each pointer uses 8 bytes (64 bits machine!)
  - ID Numbers may fit in 4 bytes
- Speed cannot be forgotten
  - Linear search is no option
  - Nor is linear insertion into ordered array

It's not just about bulk storage

Other considerations:

- Deep recurrence needs data in *Random Access Memory*
- Data must be accessible. Some choices:
  - Array of items
  - Linked tree structure using pointers
  - Array of pointers
  - Array of ID numbers, identified data looked up

  Choices imply additional overhead
  - In arrays, fixed size slots waste space at small items
  - Arrays need reallocation as data grows
  - Each pointer uses 8 bytes (64 bits machine!)
  - ID Numbers may fit in 4 bytes
- Speed cannot be forgotten
  - Linear search is no option
  - Nor is linear insertion into ordered array

What can be shared

Not every item needs separate storage.

- Recurrence makes many matrix entries copy a of some “previous” one.
  The remaining ones are called primitive; only these entries need any form of storage.
- Almost half of them are zero.
  The remaining ones are called strongly primitive.
- Among strongly primitive entries, many are the same polynomial.
  Entries may references a unique copy of polynomial.
- Maybe even distinct polynomials have parts in common

What can be shared

Not every item needs separate storage.

- Recurrence makes many matrix entries copy a of some "previous" one.
  The remaining ones are called primitive; only these entries need any form of storage.
- Almost half of them are zero.
  The remaining ones are called strongly primitive.
- Among strongly primitive entries, many are the same polynomial.
  Entries may references a unique copy of polynomial.
- Maybe even distinct polynomials have parts in common

What can be shared

Not every item needs separate storage.

- Recurrence makes many matrix entries copy a of some "previous" one.
  The remaining ones are called primitive; only these entries need any form of storage.
- Almost half of them are zero.
  The remaining ones are called strongly primitive.
- Among strongly primitive entries, many are the same polynomial.
  Entries may references a unique copy of polynomial.
- Maybe even distinct polynomials have parts in common

What can be shared

Not every item needs separate storage.

- Recurrence makes many matrix entries copy a of some "previous" one.
  The remaining ones are called primitive; only these entries need any form of storage. Already exploited
- Almost half of them are zero. Nothing stored for them.
  The remaining ones are called strongly primitive.
- Among strongly primitive entries, many are the same polynomial.
  Entries may references a unique copy of polynomial.
  Already exploited
- Maybe even distinct polynomials have parts in common
  Who knows?

What can be separated?

Since memory is a bottleneck, can we split work into parts?

Unfortunately, computing later rows requires results of (almost) all previous rows.
(But several threads of computation can work in parallel.)

Fortunately, computation involves only $+$, $-$, $\times$ of polynomials, and extraction of coefficients in *specific* degrees.

So we may separately compute remainders modulo fixed n of all coefficients.

Remainders modulo $n_1, \ldots, n_k$ determine remainder modulo $N = \mathrm{lcm}(n_1, \ldots, n_k)$ ("Chinese Remainder Theorem")

When N sufficiently large, these remainders give the result.

What can be separated?

Since memory is a bottleneck, can we split work into parts?

Unfortunately, computing later rows requires results of (almost) all previous rows.

(But several threads of computation can work in parallel.)

Fortunately, computation involves only $+$, $-$, $\times$ of polynomials, and extraction of coefficients in *specific* degrees.

So we may separately compute remainders modulo fixed n of all coefficients.

Remainders modulo $n_1, \ldots, n_k$ determine remainder modulo $N = \mathrm{lcm}(n_1, \ldots, n_k)$ ("Chinese Remainder Theorem")

When N sufficiently large, these remainders give the result.

What can be separated?

Since memory is a bottleneck, can we split work into parts?

Unfortunately, computing later rows requires results of (almost) all previous rows.
(But several threads of computation can work in parallel.)

Fortunately, computation involves only $+$, $-$, $\times$ of polynomials, and extraction of coefficients in *specific* degrees.

So we may separately compute remainders modulo fixed n of all coefficients.

Remainders modulo $n_1, \ldots, n_k$ determine remainder modulo $N = \mathrm{lcm}(n_1, \ldots, n_k)$ ("Chinese Remainder Theorem")

When N sufficiently large, these remainders give the result.

What can be separated?

Since memory is a bottleneck, can we split work into parts?

Unfortunately, computing later rows requires results of (almost) all previous rows.
(But several threads of computation can work in parallel.)

Fortunately, computation involves only $+$, $-$, $\times$ of polynomials, and extraction of coefficients in *specific* degrees.

So we may separately compute remainders modulo fixed n of all coefficients.

Remainders modulo $n_1, \ldots, n_k$ determine remainder modulo $N = \operatorname{lcm}(n_1, \ldots, n_k)$ ("Chinese Remainder Theorem")

When N sufficiently large, these remainders give the result.

What can be separated?

Since memory is a bottleneck, can we split work into parts?

Unfortunately, computing later rows requires results of (almost) all previous rows.
(But several threads of computation can work in parallel.)

Fortunately, computation involves only $+$, $-$, $\times$ of polynomials, and extraction of coefficients in *specific* degrees.

So we may separately compute remainders modulo fixed n of all coefficients.

Remainders modulo $n_1, \ldots, n_k$ determine remainder modulo $N = \mathrm{lcm}(n_1, \ldots, n_k)$ ("Chinese Remainder Theorem")

When N sufficiently large, these remainders give the result.

Where is the excess fat?

For storage, `atlas` uses ***vector*** and ***set*** structures from the `C++` standard library.

These are optimised for speed, not memory use:

- ***vector*** uses 3 pointers to access an array
- ***set*** uses 3 pointers plus one bit for each node

Memory overhead

- ***vector***: 24 bytes per vector
- ***set***: is 32 bytes per element
- Unknown additional overhead for memory management
- Some loss due to memory fragmentation (7%?)

Estimated memory requirement for split E_8 using standard `atlas` software: at least 161 GiB

Where is the excess fat?

For storage, `atlas` uses ***vector*** and ***set*** structures from the `C++` standard library.

These are optimised for speed, not memory use:

- ***vector*** uses 3 pointers to access an array
- ***set*** uses 3 pointers plus one bit for each node

Memory overhead

- ***vector***: 24 bytes per vector
- ***set***: is 32 bytes per element
- Unknown additional overhead for memory management
- Some loss due to memory fragmentation (7%?)

Estimated memory requirement for split E_8 using standard `atlas` software: at least 161 GiB

Where is the excess fat?

For storage, `atlas` uses ***vector*** and ***set*** structures from the `C++` standard library.
These are optimised for speed, not memory use:

- ***vector*** uses 3 pointers to access an array
- ***set*** uses 3 pointers plus one bit for each node

Memory overhead

- ***vector***: 24 bytes per vector
- ***set***: is 32 bytes per element
- Unknown additional overhead for memory management
- Some loss due to memory fragmentation (7%?)

Estimated memory requirement for split E_8 using standard `atlas` software: at least 161 GiB

Where is the excess fat?

For storage, `atlas` uses ***vector*** and ***set*** structures from the `C++` standard library.
These are optimised for speed, not memory use:

- ***vector*** uses 3 pointers to access an array
- ***set*** uses 3 pointers plus one bit for each node

Memory overhead

- ***vector***: 24 bytes per vector
- ***set***: is 32 bytes per element
- Unknown additional overhead for memory management
- Some loss due to memory fragmentation (7%?)

Estimated memory requirement for split E_8 using standard `atlas` software: at least 161 GiB

Outline

Using modular arithmetic

- Computing in $\mathbf{Z}/n\mathbf{Z}$ is easy; slightly slower than in $\mathbf{Z}$
- But no worry about overflow/underflow: speed gain
- `C++` "modular integer" class can be defined,
  and used as drop-in replacement for certain integers
- `atlas` defined type ***KLCoeff***, allowing easy replacement
- 1 day work, gain 41 GiB

Using modular arithmetic

- Computing in $\mathbf{Z}/n\mathbf{Z}$ is easy; slightly slower than in $\mathbf{Z}$
- But no worry about overflow/underflow: speed gain
- `C++` "modular integer" class can be defined,
  and used as drop-in replacement for certain integers
- `atlas` defined type ***KLCoeff***, allowing easy replacement
- 1 day work, gain 41 GiB

Using modular arithmetic

- Computing in $\mathbf{Z}/n\mathbf{Z}$ is easy; slightly slower than in $\mathbf{Z}$
- But no worry about overflow/underflow: speed gain
- `C++` "modular integer" class can be defined, and used as drop-in replacement for certain integers
- `atlas` defined type ***KLCoeff***, allowing easy replacement
- 1 day work, gain 41 GiB

Using modular arithmetic

- Computing in $\mathbf{Z}/n\mathbf{Z}$ is easy; slightly slower than in $\mathbf{Z}$
- But no worry about overflow/underflow: speed gain
- `C++` "modular integer" class can be defined, and used as drop-in replacement for certain integers
- `atlas` defined type ***KLCoeff***, allowing easy replacement
- 1 day work, gain 41 GiB

Representing all known polynomials

`Atlas` used: ***set<vector<KLCoeff> >*** *store*. For computed *P*:

- look up *P* in *store* (binary search)
- if not present, insert a copy of *P* into *store*
- in matrix, store pointer to node found/inserted; discard *P*

The ***set*** structure is suited, but gives more than is needed.

Instead use a hash table of polynomials. For computed *P*:

- search ID for *P* near *table[hash(P)]*
- if not present: copy *P* to storage vector, assigning new ID; store ID near *table[hash(P)]*
- in matrix, store the ID found/assigned; discard *P*

4 days work, gain 19 GiB (but some 20 GiB unused allocation)

Representing all known polynomials

`Atlas` used: ***set<vector<KLCoeff> >*** *store*. For computed *P*:

- look up *P* in *store* (binary search)
- if not present, insert a copy of *P* into *store*
- in matrix, store pointer to node found/inserted; discard *P*

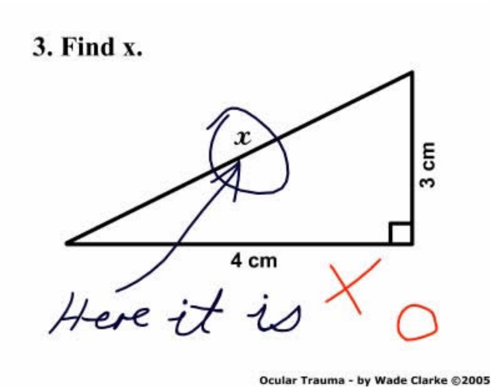
The ***set*** structure is suited, but gives more than is needed.

Instead use a hash table of polynomials. For computed *P*:

- search ID for *P* near *table[hash(P)]*
- if not present: copy *P* to storage vector, assigning new ID; store ID near *table[hash(P)]*
- in matrix, store the ID found/assigned; discard *P*

4 days work, gain 19 GiB (but some 20 GiB unused allocation)

Representing all known polynomials



Representing all known polynomials

`Atlas` used: ***set<vector<KLCoeff> >*** *store*. For computed *P*:

- look up *P* in *store* (binary search)
- if not present, insert a copy of *P* into *store*
- in matrix, store pointer to node found/inserted; discard *P*

The ***set*** structure is suited, but gives more than is needed.

Instead use a hash table of polynomials. For computed *P*:

- search ID for *P* near *table[hash(P)]*
- if not present: copy *P* to storage vector, assigning new ID; store ID near *table[hash(P)]*
- in matrix, store the ID found/assigned; discard *P*

4 days work, gain 19 GiB (but some 20 GiB unused allocation)

Representing all known polynomials

`Atlas` used: ***set<vector<KLCoeff> >*** *store*. For computed *P*:

- look up *P* in *store* (binary search)
- if not present, insert a copy of *P* into *store*
- in matrix, store pointer to node found/inserted; discard *P*

The ***set*** structure is suited, but gives more than is needed.

Instead use a hash table of polynomials. For computed *P*:

- search ID for *P* near *table[hash(P)]*
- if not present: copy *P* to storage vector, assigning new ID; store ID near *table[hash(P)]*
- in matrix, store the ID found/assigned; discard *P*

4 days work, gain 19 GiB (but some 20 GiB unused allocation)

Representing all known polynomials

`Atlas` used: ***set<vector<KLCoeff> >*** *store*. For computed *P*:

- look up *P* in *store* (binary search)
- if not present, insert a copy of *P* into *store*
- in matrix, store pointer to node found/inserted; discard *P*

The ***set*** structure is suited, but gives more than is needed.

Instead use a hash table of polynomials. For computed *P*:

- search ID for *P* near *table[hash(P)]*
- if not present: copy *P* to storage vector, assigning new ID; store ID near *table[hash(P)]*
- in matrix, store the ID found/assigned; discard *P*

4 days work, gain 19 GiB (but some 20 GiB unused allocation)

Overhead of polynomial structure

Each polynomial P is a ***vector***.
Its coefficient array is stored, can grow/shrink and be discarded separately.

When P is found to be new, this is no longer needed.
By copying coefficients of P to common array (pool),
the use of 3 pointers can be reduced to 1.

Use of stored polynomials in arithmetic and in matching (hash table) needs rewriting.

2 days work, gain 31 GiB

Overhead of polynomial structure

Each polynomial P is a ***vector***.
Its coefficient array is stored, can grow/shrink and be discarded separately.

When P is found to be new, this is no longer needed.
By copying coefficients of P to common array (pool),
the use of 3 pointers can be reduced to 1.

Use of stored polynomials in arithmetic and in matching (hash table) needs rewriting.

2 days work, gain 31 GiB

Overhead of polynomial structure

Each polynomial P is a ***vector***.
Its coefficient array is stored, can grow/shrink and be discarded separately.

When P is found to be new, this is no longer needed.
By copying coefficients of P to common array (pool),
the use of 3 pointers can be reduced to 1.

Use of stored polynomials in arithmetic and in matching (hash table) needs rewriting.

2 days work, gain 31 GiB

Overhead of polynomial structure

Each polynomial P is a ***vector***.
Its coefficient array is stored, can grow/shrink and be discarded separately.

When P is found to be new, this is no longer needed.
By copying coefficients of P to common array (pool),
the use of 3 pointers can be reduced to 1.

Use of stored polynomials in arithmetic and in matching (hash table) needs rewriting.

2 days work, gain 31 GiB

Not every polynomial needs a pointer

For stored P, need to locate first coefficient in pool.
Final coefficient is determined by start of next polynomial.

Each polynomial has length ≤ 32.
Pointer to first coefficient (8 bytes) contains less than 1 byte of real information.

But storing *only* differences would make locating coefficients too slow.

So decided: store 5 byte pool index once every 16 polynomials, and a 1 byte difference for each of next 15 polynomials.

2 days work, gain 15 GiB

Not every polynomial needs a pointer

For stored P, need to locate first coefficient in pool.
Final coefficient is determined by start of next polynomial.

Each polynomial has length ≤ 32.
Pointer to first coefficient (8 bytes) contains less than 1 byte of real information.

But storing *only* differences would make locating coefficients too slow.

So decided: store 5 byte pool index once every 16 polynomials, and a 1 byte difference for each of next 15 polynomials.

2 days work, gain 15 GiB

Not every polynomial needs a pointer

For stored P, need to locate first coefficient in pool.
Final coefficient is determined by start of next polynomial.

Each polynomial has length ≤ 32.
Pointer to first coefficient (8 bytes) contains less than 1 byte of real information.

But storing *only* differences would make locating coefficients too slow.

So decided: store 5 byte pool index once every 16 polynomials, and a 1 byte difference for each of next 15 polynomials.

2 days work, gain 15 GiB

Not every polynomial needs a pointer

For stored P, need to locate first coefficient in pool.
Final coefficient is determined by start of next polynomial.

Each polynomial has length ≤ 32.
Pointer to first coefficient (8 bytes) contains less than 1 byte of real information.

But storing *only* differences would make locating coefficients too slow.

So decided: store 5 byte pool index once every 16 polynomials, and a 1 byte difference for each of next 15 polynomials.

2 days work, gain 15 GiB

Not every polynomial needs a pointer

For stored P, need to locate first coefficient in pool.
Final coefficient is determined by start of next polynomial.

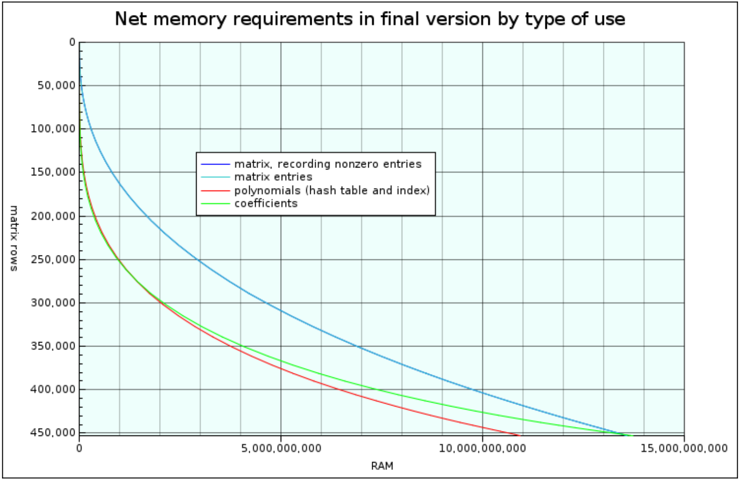
Each polynomial has length ≤ 32.
Pointer to first coefficient (8 bytes) contains less than 1 byte of real information.

But storing *only* differences would make locating coefficients too slow.

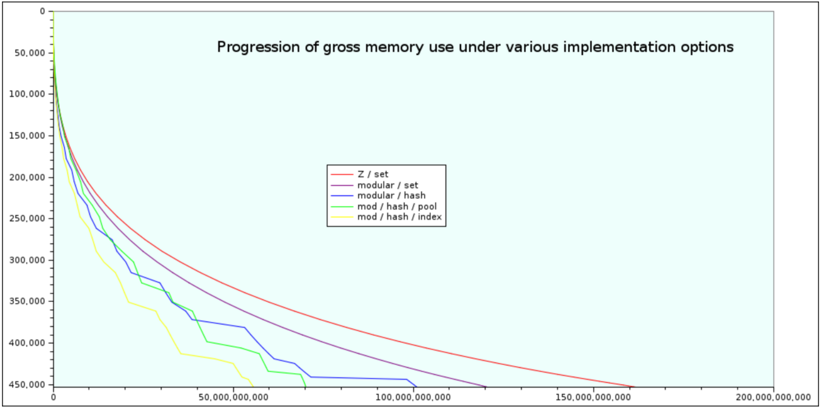
So decided: store 5 byte pool index once every 16 polynomials, and a 1 byte difference for each of next 15 polynomials.

2 days work, gain 15 GiB

Final rows weigh in heavy



What difference does it make?



Outline

1. Surveying the Challenge
   - First estimates
   - How is Memory Used?
2. Reducing Memory Use
   - Modular Reduction of Coefficients
   - The Set of Known Polynomials
   - Individual Polynomials
3. Writing and Processing the Result
   - Output Format
   - Processing Strategy
4. What went Wrong (until it was fixed)
   - Mysterious Malfunctions
   - Safe but Slow Solutions
   - Precision Problems

Output requirements

- Compact (so retain sharing)
- Processing oriented
- Allow random access to polynomials
- Cater for variations between moduli in zeros and in polynomial numbering

Choice:

- Separate matrix and polynomial files
- Binary format
- Prefer simplicity over extreme compactness

Output requirements

- Compact (so retain sharing)
- Processing oriented
- Allow random access to polynomials
- Cater for variations between moduli in zeros and in polynomial numbering

Choice:

- Separate matrix and polynomial files
- Binary format
- Prefer simplicity over extreme compactness

Output requirements

- Compact (so retain sharing)
- Processing oriented
- Allow random access to polynomials
- Cater for variations between moduli
  in zeros and in polynomial numbering

Choice:

- Separate matrix and polynomial files
- Binary format
- Prefer simplicity over extreme compactness

Output requirements

- Compact (so retain sharing)
- Processing oriented
- Allow random access to polynomials
- Cater for variations between moduli
  in zeros and in polynomial numbering

Choice:

- Separate matrix and polynomial files
- Binary format
- Prefer simplicity over extreme compactness

Output requirements

- Compact (so retain sharing)
- Processing oriented
- Allow random access to polynomials
- Cater for variations between moduli in zeros and in polynomial numbering

Choice:

- Separate matrix and polynomial files
- Binary format
- Prefer simplicity over extreme compactness

Output format

For polynomials record:

- their number,
- for each one the offset of its first coefficient
- all the coefficients.

For matrices record sequence of rows, with

- a bitmap indicating, among the primitive entries, the strongly primitive ones
- the ID numbers of the nonzero entries

Output format

For polynomials record:

- their number,
- for each one the offset of its first coefficient
- all the coefficients.

For matrices record sequence of rows, with

- a bitmap indicating, among the primitive entries, the strongly primitive ones
- the ID numbers of the nonzero entries

Making polynomial numberings agree

Numbering of polynomials may differ between moduli, because

- modular reduction could make polynomials equal, or zero;
- multi-threading randomly perturbs assignment of ID's.

Therefore, matrices must be compared before polynomials.

Traverse corresponding matrix entries for all k moduli:

- if entry is strong only for some moduli, take 0 at others;
- look up if k-tuple if ID's is new; if so assign new canonical ID to tuple, otherwise use ID from table;
- to do this, use a hash table;
- write the canonical ID to a new matrix file.

Then write k-tuples of modular ID's to k renumbering files

Making polynomial numberings agree

Numbering of polynomials may differ between moduli, because

- modular reduction could make polynomials equal, or zero;
- multi-threading randomly perturbs assignment of ID's.

Therefore, matrices must be compared before polynomials.

Traverse corresponding matrix entries for all k moduli:

- if entry is strong only for some moduli, take 0 at others;
- look up if k-tuple if ID's is new; if so assign new canonical ID to tuple, otherwise use ID from table;
- to do this, use a hash table;
- write the canonical ID to a new matrix file.

Then write k-tuples of modular ID's to k renumbering files

Making polynomial numberings agree

Numbering of polynomials may differ between moduli, because

- modular reduction could make polynomials equal, or zero;
- multi-threading randomly perturbs assignment of ID's.

Therefore, matrices must be compared before polynomials.

Traverse corresponding matrix entries for all k moduli:

- if entry is strong only for some moduli, take 0 at others;
- look up if k-tuple if ID's is new; if so assign new canonical ID to tuple, otherwise use ID from table;
- to do this, use a hash table;
- write the canonical ID to a new matrix file.

Then write k-tuples of modular ID's to k renumbering files

Modular lifting

Now translating polynomial ID's through renumbering, look up corresponding modular polynomials. Must solve:

Given remainders r_1 mod n_1, and r_2 mod n_2,
find r such that $r \equiv r_1 \pmod{n_1}$ and $r \equiv r_2 \pmod{n_2}$.

- We must adjust r_1 by multiple m of n_1 with $r_1 + m \equiv r_2 \pmod{n_2}$.
- Let m_0 be multiple of n_1 such that $m_0 \equiv d = \gcd(n_1, n_2) \pmod{n_2}$ (exists by "Bezout").
- If $r_2 - r_1$ not multiple of d, no solution exists.
- Otherwise $m = m_0 \frac{r_2 - r_1}{d}$ works.
- The number m_0 is independent of r_1 and r_2.

Modular lifting

Now translating polynomial ID's through renumbering, look up corresponding modular polynomials. Must solve:

Given remainders r_1 mod n_1, and r_2 mod n_2,
find r such that $r \equiv r_1 \pmod{n_1}$ and $r \equiv r_2 \pmod{n_2}$.

- We must adjust r_1 by multiple m of n_1 with $r_1 + m \equiv r_2 \pmod{n_2}$.
- Let m_0 be multiple of n_1 such that $m_0 \equiv d = \gcd(n_1, n_2) \pmod{n_2}$ (exists by "Bezout").
- If $r_2 - r_1$ not multiple of d, no solution exists.
- Otherwise $m = m_0 \frac{r_2 - r_1}{d}$ works.
- The number m_0 is independent of r_1 and r_2.

Modular lifting

Now translating polynomial ID's through renumbering, look up corresponding modular polynomials. Must solve:

Given remainders r_1 mod n_1, and r_2 mod n_2,
find r such that $r \equiv r_1 \pmod{n_1}$ and $r \equiv r_2 \pmod{n_2}$.

- We must adjust r_1 by multiple m of n_1 with $r_1 + m \equiv r_2 \pmod{n_2}$.
- Let m_0 be multiple of n_1 such that $m_0 \equiv d = \gcd(n_1, n_2) \pmod{n_2}$ (exists by "Bezout").
- If $r_2 - r_1$ not multiple of d, no solution exists.
- Otherwise $m = m_0 \frac{r_2 - r_1}{d}$ works.
- The number m_0 is independent of r_1 and r_2.

Modular lifting

Now translating polynomial ID's through renumbering, look up corresponding modular polynomials. Must solve:

Given remainders $r_1 \bmod n_1$, and $r_2 \bmod n_2$,
find r such that $r \equiv r_1 \pmod{n_1}$ and $r \equiv r_2 \pmod{n_2}$.

- We must adjust r_1 by multiple m of n_1 with $r_1 + m \equiv r_2 \pmod{n_2}$.
- Let m_0 be multiple of n_1 such that $m_0 \equiv d = \gcd(n_1, n_2) \pmod{n_2}$ (exists by "Bezout").
- If $r_2 - r_1$ not multiple of d, no solution exists.
- Otherwise $m = m_0 \frac{r_2 - r_1}{d}$ works.
- The number m_0 is independent of r_1 and r_2.

Modular lifting

Now translating polynomial ID's through renumbering, look up corresponding modular polynomials. Must solve:

Given remainders r_1 mod n_1, and r_2 mod n_2,
find r such that $r \equiv r_1 \pmod{n_1}$ and $r \equiv r_2 \pmod{n_2}$.

- We must adjust r_1 by multiple m of n_1 with $r_1 + m \equiv r_2 \pmod{n_2}$.
- Let m_0 be multiple of n_1 such that $m_0 \equiv d = \gcd(n_1, n_2) \pmod{n_2}$ (exists by "Bezout").
- If $r_2 - r_1$ not multiple of d, no solution exists.
- Otherwise $m = m_0 \frac{r_2 - r_1}{d}$ works.
- The number m_0 is independent of r_1 and r_2.

Modular lifting

Now translating polynomial ID's through renumbering, look up corresponding modular polynomials. Must solve:

Given remainders r_1 mod n_1, and r_2 mod n_2,
find r such that $r \equiv r_1 \pmod{n_1}$ and $r \equiv r_2 \pmod{n_2}$.

- We must adjust r_1 by multiple m of n_1 with $r_1 + m \equiv r_2 \pmod{n_2}$.
- Let m_0 be multiple of n_1 such that $m_0 \equiv d = \gcd(n_1, n_2) \pmod{n_2}$ (exists by "Bezout").
- If $r_2 - r_1$ not multiple of d, no solution exists.
- Otherwise $m = m_0 \frac{r_2 - r_1}{d}$ works.
- The number m_0 is independent of r_1 and r_2.

Outline

Surprising scenarios

Various subtleties of `C++` have caused headaches:

- defining too much automatic conversion inadvertently did: reference$\longrightarrow$copy$\longrightarrow$reference, and values evaporated;
- strange bit shifting semantics can bite: necessary $x \gg 32$ failed tests on 32-bit machine.

Why David was right after all

With so much data, one cannot afford inefficiency.

Lessons we learned the hard way:

- a bad choice of hash function gives congestion;
- counting bits in a bitmap can be costly;
- making threads safe can make them useless;
- don't over-display counters.

Why David was right after all

With so much data, one cannot afford inefficiency.

Lessons we learned the hard way:

- a bad choice of hash function gives congestion;
- counting bits in a bitmap can be costly;
- making threads safe can make them useless;
- don't over-display counters.

Watch your Width

When space is tight, one must be extra alert:

- width of operands, not of result, determine operation;
  $offset_{\langle 8\rangle} = base_{\langle 8\rangle} + 5 * renumbering_{\langle 4\rangle}[i_{\langle 8\rangle}]$
- modular lifting can easily overflow (tables can solve this).

Watch your Width

When space is tight, one must be extra alert:

- width of operands, not of result, determine operation;
  $\mathit{offset}_{\langle 8\rangle} = \mathit{base}_{\langle 8\rangle} + 5 * \mathit{renumbering}_{\langle 4\rangle}[i_{\langle 8\rangle}]$
- modular lifting can easily overflow (tables can solve this).

Summary

- Tailoring implementation of abstract data types can easily give substantial gain.
- Simple binary formats allow (relatively) rapid processing.
- Handling massive data is a challenge, but fun.