university of groningen

faculty of mathematics and natural sciences

Calculating the size of the $[\neg, \vee]$ fragment of intuitionistic logic

Bachelor's thesis

28 October 2014

Student: R.S.R. Zijlstra

Primary supervisor: Prof. Dr. G.R. Renardel de Lavalette

Secondary supervisor: Dr. A. Meijster

Abstract

The Dedekind Numbers d_n,named after the German mathematician Richard Dedekind (October 6, 1831 - February 12, 1916), is a sequence of integers which grows very fast. The number d_n are the number of monotone subsets of the powerset of a set with n elements. The latest known number in this series is $d_8 = 56,130,437,228,687,557,907,788$. Recently A.T. Zijlstra recreated the method of computing this number with the method of D. Wiedemann. The method was implemented in C++ and parallelized using MPI.
This thesis will focus on the calculation of a different sequence. Intuitionistic logic is a logic which differs from classical logic. The number of non-equivalent formulae in this logic is infinite but when we restrict the connectives used in the formulae the number of non-equivalent formulae becomes finite. We will focus only on the $\neg$ and $\vee$ connectives. We will calculate the size of this logic utilizing an extension of Wiedemann's method.

Contents

1 Introduction **3**

2 Preliminaries **4**

3 Calculating Dedekind Numbers **5**
- 3.1 Calculating D_{n+1} from D_n . . . 5
- 3.2 Calculating d_{n+2} from D_n . . . 6
- 3.3 Calculating d_{n+2} from D_n and R_n . . . 8

4 Intuitionistic logic **10**

5 Calculating $\#[\neg, \vee, p, q, r]_{int}$ **11**
- 5.1 First method . . . 12
  - 5.1.1 Symmetries in δ . . . 13
  - 5.1.2 Computing δ . . . 14
- 5.2 Second method . . . 16
  - 5.2.1 calculation δ' . . . 17

6 Implementation **19**
- 6.1 Helper Functions . . . 19
  - 6.1.1 Bitset . . . 19
  - 6.1.2 UInt128 . . . 19
  - 6.1.3 Subsets . . . 20
  - 6.1.4 Concatenations . . . 20
  - 6.1.5 Power Set . . . 20
  - 6.1.6 Permutation . . . 21
  - 6.1.7 Equivalence classes . . . 22
  - 6.1.8 Sorting bitset . . . 22
  - 6.1.9 generateIndexes . . . 22
  - 6.1.10 generateGammas method 1 . . . 23
  - 6.1.11 generateGammas method 2 . . . 24
- 6.2 Main Functions . . . 24
  - 6.2.1 Calculating D_{n+1} from D_n . . . 24
  - 6.2.2 Calculating R_n . . . 25
  - 6.2.3 Calculating d_{n+2} using D_n and R_n . . . 25
  - 6.2.4 Calculation ζ_n Method 1 . . . 26
  - 6.2.5 Calculation ζ_n Method 2 . . . 28

7 Results and Discussion **30**

Appendix

1 Introduction

ζ_n is the number of non-equivalent formulae of the fragment $[\neg, \vee, P_n]_{int}$ of the intuitionistic logic where $P_n = \{p_0, p_1, ..., p_{n-1}\}$. This sequence is growing at an enormous rate. A fragment is a subset of a logic with restrictions on the connectives. In our case a smaller set of connectives: $\neg$ and $\vee$.
The goal of this thesis is to calculate the size of $[\neg, \vee, P_n]_{int}$ to gain knowledge of its structure. The computation requires us to analyse the structure of this fragment to further increase our knowledge. Furthermore the computation of ζ_n will also be useful for calculate the size of similar fragments.
Computation of ζ_n can be achieved using a similar calculation to the one for the Dedekind number d_{2^n}. Currently d_0 through d_8 have been calculated. Using the calculation of d_8 it is possible to calculate ζ_3 in a reasonable amount of time. The methods used to calculate d_n will be explained. After that the method to calculate ζ_n using this technique will be explained. Furthermore the calculation for ζ_n will be implemented in C++ and parallelised using the Message Passing Interface.

n	d_n
0	2
1	3
2	6
3	20
4	168
5	7,581
6	7,828,354
7	2,414,682,040,998
8	56,130,437,228,687,557,907,788

n	ζ_n
1	7
2	385
3	191,589,906,593,484,837,139,681

2 Preliminaries

First we shall introduce some mathematical structures. The power set $\wp(S)$ of a set S is defined as usual by

$$\wp(S) = \{P \mid P \subseteq S\}$$

and we define

$$Q(n) = \wp(\{0, 1, ..., n-1\})$$

A partial ordering over a set X is a binary relation on X, usually denoted by $\leq$ or $\subseteq$, which is reflexive, antisymmetric, and transitive. So we have $\forall a, b, c \in X$,

$$\begin{aligned} &a \leq a \\ &a \leq b,\ b \leq a \Rightarrow a = b \\ &a \leq b,\ b \leq c \Rightarrow a \leq c \end{aligned}$$

A subset S of a partially ordered set X is called monotonic in X whenever S is upward closed

$$\forall t \in S, \forall u \in X\ (t \leq u \Rightarrow u \in S)$$

We introduce a notation for the set of all monotonic subsets.

$$\begin{aligned} \mathcal{M}(X) &= \{Y \subseteq X \mid Y \text{ monotonic in } X\} \\ \mathcal{M}^{+}(X) &= \mathcal{M}(X) - \{\emptyset\} \\ \mathcal{M}^{\pm}(X) &= \mathcal{M}(X) - \{\emptyset, X\} \end{aligned}$$

We introduce two new operators for adding or removing an element to all elements of a set of sets.

$$\begin{aligned} S \oplus n &= \{t \cup \{n\} \mid t \in S\} \\ S \ominus n &= \{t - \{n\} \mid t \in S, n \in t\} \end{aligned}$$

The $\ominus$ operator is a bit tricky. It should not be confused with $\{t - \{n\} \mid t \in S\}$. We illustrate the difference with an example:

$$\begin{aligned} &S = \{\{1\}, \{1, 2\}, \{2, 3\}\} \\ &\{t - \{1\} \mid t \in S\} = \{\emptyset, \{2\}, \{2, 3\}\} \\ &S \ominus 1 = \{\emptyset, \{2\}\} \end{aligned}$$

If S is monotonic in $Q(n)$, then $S \ominus (n-1)$ is monotonic in $Q(n-1)$ and $S \oplus n$ is monotonic in $Q(n+1)$.

3 Calculating Dedekind Numbers

The sequence of Dedekind Numbers (d_n) is a fast growing series of integers. The number d_n is equal to the number of monotonic subsets of Q(n). The problem of calculating these numbers is called Dedekind's Problem. Let us define

$$D_n = \{S \mid S \text{ is monotonic in } Q(n)\}$$

then

$$d_n = \#D_n$$

As an example we will show D_0, D_1 and D_2, First we observe

$$\begin{aligned} Q(0) &= \{\emptyset\} \\ Q(1) &= \{\emptyset, \{0\}\} \\ Q(2) &= \{\emptyset, \{0\}, \{1\}, \{0,1\}\} \end{aligned}$$

The notation of sets in this way yields unnecessary large expressions. Therefore we will use a shorter form. The set $\{a, b, c\}$ will be denoted by abc. (Note that this would give problems with sets $\{\{10, 1, 0\}\}$ or $\{1, \{1\}\}$ but that is not an issue for our purpose). Note that $Q(n)$ is a set of sets. This means that D_n is a set of sets of sets. Also note that there is a difference between $\{\emptyset\}$ and $\emptyset$. The former is a set with one element, the empty set, while the latter is a set with no elements. We have

$$\begin{aligned} D_0 &= \{\{\emptyset\}, \emptyset\} \\ D_1 &= \{\{\emptyset, 0\}, \{0\}, \emptyset\} \\ D_2 &= \{\{\emptyset, 0, 1, 01\}, \{0, 1, 01\}, \{0, 01\}, \{1, 01\}, \{01\}, \emptyset\} \end{aligned}$$

As you can see $\#D_0 = 2$, $\#D_1 = 3$ and $\#D_2 = 6$. Because D_8 is an enormous set, it would take days to generate the needed D_8 in order to calculate $\#[\neg, \vee, p, q, r]_{int}$. To do the calculation more efficiently, we shall use the method of Wiedemann (1991). This method only needs D_{n-2} to compute d_n. This means that we only need D_6 for the computation of d_8

3.1 Calculating D_{n+1} from D_n

We need to construct D_n. To do this a method is used to construct D_{k+1} from D_k. When we apply this multiple times, we obtain D_n. To do this we observe that we can split every $S \in D_{k+1}$ in two parts, $T, U \in D_k$ as follows:

$$\begin{aligned} T &= Q(k) \cap S \\ U &= Q(k)S \ominus k \end{aligned}$$

Conversely: when $T, U \in D_k$ we have that $T \cup (U \oplus k) \subseteq Q(k+1)$ and when $T \subseteq U$ we have that $T \cup (U \oplus k)$ is monotonic. If we go through all combinations of T and U we can generate all elements of D_{k+1} So we have

$$D_{k+1} = \{T \cup (U \oplus k) \mid T, U \in D_k, T \subseteq U\} \tag{1}$$

Let us use this method to generate D_2 from $D_1 = \{\{\emptyset, 0\}, \{0\}, \emptyset\}$. We let T and U run over the elements of D_1 comparing T with all elements U of D_1 to see whether $T \subseteq U$, in which case we add 1 to the elements of U and take the union with T. Thus we obtain all elements of D_2 so we can construct it.

T	$U \oplus k$		
	$\{\emptyset, 0\} \oplus 1 = \{1, 01\}$	$\{0\} \oplus 1 = \{01\}$	$\emptyset \oplus 1 = \emptyset$
$\{\emptyset, 0\}$	$\{\emptyset, 0, 1, 01\}$	$T \not\subseteq U$	$T \not\subseteq U$
$\{0\}$	$\{0, 1, 01\}$	$\{0, 01\}$	$T \not\subseteq U$
$\emptyset$	$\{1, 01\}$	$\{01\}$	$\emptyset$

$$D_2 = \{\{\emptyset, 0, 1, 01\}, \{0, 1, 01\}, \{0, 01\}, \{1, 01\}, \{01\}, \emptyset\}$$

D_3 and higher can be constructed in the same way.

3.2 Calculating d_{n+2} from D_n

The following method does not construct the entire D_{n+2} for computing $d_{n+2} = \#Dn + 2$. We will use a method which will only construct D_n and use it to calculate d_{n+2} without explicit construction D_{n+2}. To calculate d_{n+2} from D_n we split each $S \in D_{n+2}$ into four components $A, B, C, D \in D_n$:

$$\begin{aligned} A &= Q(n) \cap S \\ B &= Q(n) \cap (S \ominus n) \\ C &= Q(n) \cap (S \ominus (n+1)) \\ D &= (S \ominus n) \ominus (n+1) \end{aligned} \tag{2}$$

In order to reconstruct S from A, B, C, D, we define $f : \wp(Q(n))^4 \to \wp(Q(n+2))$ by:

$$f(A, B, C, D) = A \cup (B \oplus n) \cup (C \oplus (n+1)) \cup ((D \oplus n) \oplus (n+1))$$

Now we have

$$S = f(A, B, C, D)$$

Because S is monotonic we can derive some additional properties:

$$A \subseteq B,\ A \subseteq C,\ C \subseteq D,\ B \subseteq D$$

A, B, C and D are also monotonic. To generate D_{n+2} we take all possible combinations of A, B, C, D that satisfy these conditions and add the elements back.

$$D_n = \{f(A, B, C, D) \mid A, B, C, D \in D_{n-2}, A \subseteq B \subseteq D, A \subseteq C \subseteq D\}$$

Observe that we are in fact applying (1) twice. To actually get a speed up, some information must be ignored. We only want the number of monotonic subsets in $Q(n+2)$. A lot of time is wasted generating the entire D_{n+2}.
The next method will only run through every combination of B and C, from which we calculate the number of possible A's and D's and multiply these numbers. This will give us the number of elements from D_{n+2} that can be constructed from a particular B and C. We do this for every combination and sum them to get d_{n+2}.
To calculate the number of A's and D's for all B and C with a single function we introduce the notion of duality. The dual S^* of a set $S \subseteq Q(n)$ is defined as

$$\begin{aligned} \cdot^* &: \wp(Q(n)) \to \wp(Q(n)) \\ S^* &= \{t^c \mid t \in S\}^c \\ &= \{\{0, ..., n-1\} - t \mid t \in S\}^c \\ &= Q(n) - \{\{0, ..., n-1\} - t \mid t \in S\} \end{aligned}$$

S^* has some interesting properties:

$$\begin{aligned} S \in D_n &\Leftrightarrow S^* \in D_n \\ S \subseteq T &\Leftrightarrow T^* \subseteq S^* \\ (S \cup T)^* &= S^* \cap T^* \end{aligned}$$

We define $\eta(T)$ as the number of monotonic subsets of $Q(n)$ which are contained in T

$$\begin{aligned} \eta &: D_n \to \mathbb{N} \\ \eta(T) &= \#\{S \in D_n \mid S \subseteq T\} \\ &= \#(D_n \cap \{S \mid S \subseteq T\}) \\ &= \#(D_n \cap \wp(T)) \end{aligned}$$

Now we can calculate the number of possible A's and D's for any given B and C as follows. The number of possible A's equals $\eta(C \cap B)$, and the number of

possible D's equals $\eta(C^* \cap B^*)$, for which we have

$$\begin{aligned}\#\{S \in D_n \mid (C \cup B) \subseteq S\} &= \#\{S^* \in D_n \mid S^* \subseteq (C \cup B)^*\} \\ &= \#\{S \in D_n \mid S \subseteq (C^* \cap B^*)\} \\ &= \eta(C^* \cap B^*)\end{aligned}$$

Observe that we never calculate all the possible A's or D's explicitly, but only their number with a given B and C. This saves an enormous amount of time. Now that we have all the parts we need to calculate d_{n+2}, we can put it all together and obtain d_{n+2}.

$$d_{n+2} = \sum_{C \in D_n} \sum_{B \in D_n} \eta(C \cap B) \cdot \eta(C^* \cap B^*) \tag{3}$$

3.3 Calculating d_{n+2} from D_n and R_n

The method so far is still not efficient enough for computing d_{n+2}. The current method in (3) has a double loop over D_n. If we take a look at D_n we can see that it contains a lot of symmetries. Wiedemann (1991) also noticed these symmetries.

$$\begin{aligned}&\texttt{PERM} = \{\pi \mid \pi : \wp(Q(n)) \to \wp(Q(n)) \text{ is bijective}\} \\ &A \sim B \text{ iff } \exists \text{ permutation } \pi \text{ such that } \pi(A) = B\end{aligned}$$

With this we can define equivalence classes $[A]_\sim$ of D_n.

$$\begin{aligned}[A]_\sim &= \{B \in D_n \mid A \sim B\} \\ D_n/_\sim &= \{[A]_\sim \mid A \in D_n\}\end{aligned}$$

Let R_n contain exactly one element of every equivalence class in $D_n/_\sim$, so

$$\forall A \in D_n \ \#(R_n \cap [A]_\sim) = 1$$

Every element in D_n can then be mapped to a unique element of R_n by using some permutation π. Let

$$p : R_n \to \wp(\texttt{PERM})$$

$p(K)$ is the set which for every $A \in [K]_\sim$ holds one permutation π such that $\pi(K) = A$. So

$$\begin{aligned}\#p(K) &= \#[K]_\sim \\ \forall A \in [K]_\sim \ \exists \pi \in p(K) \ \pi(K) &= A\end{aligned}$$

Now we have

$$
\begin{aligned}
d_{n+2} = & \sum_{C \in D_n} \sum_{B \in D_n} \eta(C \cap B) \cdot \eta(C^* \cap B^*) \\
= & \ \langle D_n = \{C \mid \exists K \in R_n \ C \sim K\} \ \rangle \\
& \sum_{K \in R_n} \sum_{C \sim K} \sum_{B \in D_n} \eta(C \cap B) \cdot \eta(C^* \cap B^*) \\
= & \ \langle \text{ applying the permutations in } p(K) \rangle \\
& \sum_{K \in R_n} \sum_{\pi \in p(K)} \sum_{B \in D_n} \eta(\pi(K) \cap B) \cdot \eta(\pi(K)^* \cap B^*) \\
= & \ \langle \forall \pi \ D_n = \{\pi(S) \mid S \in D_n\} = \pi[D_n] \rangle \\
& \sum_{K \in R_n} \sum_{\pi \in p(K)} \sum_{B \in D_n} \eta(\pi(K) \cap \pi(B)) \cdot \eta(\pi(K)^* \cap \pi(B)^*) \\
= & \ \langle \pi(X) \cap \pi(Y) = \pi(X \cap Y) \rangle \\
& \sum_{K \in R_n} \sum_{\pi \in p(K)} \sum_{B \in D_n} \eta(\pi(K \cap B)) \cdot \eta(\pi(K^* \cap B^*)) \\
= & \ \langle \eta(X) = \eta(\pi(X)) \rangle \\
& \sum_{K \in R_n} \sum_{B \in D_n} \gamma(K) \cdot \eta(K \cap B) \cdot \eta(K^* \cap B^*) \\
= & \ \langle \text{ rearranging} \rangle \\
& \sum_{K \in R_n} \gamma(K) \cdot \sum_{B \in D_n} \eta(K \cap B) \cdot \eta(K^* \cap B^*)
\end{aligned}
$$

where $\gamma(K) = \#p(K)$. Note that due to the many symmetries in D_n we have that $\#R_n$ is significantly lower than $\#D_n$. We do need some preprocessing to calculate $\gamma(K)$ for all $K \in R_n$. Now we are able to calculate d_n. See Zijlstra (2013) for more details.

4 Intuitionistic logic

We need to know something about intuitionistic logic as well. Unlike classical logic, it does not focus on a two-valued concept of truth. In classical logic a proposition is true or false even if we do not have direct evidence for it. In intuitionistic logic this is not the case. It can be such that a proposition can not be assigned any value due to a lack of proof. Several axioms of the classical logic do not hold in intuitionistic logic. The first one is the law of excluded middle: $p \vee \neg p$, which allows a proposition not to be assigned a value true or false. However this does not mean this is a three-valued or many-valued logic. The second axiom that does not hold is the double negation elimination: $\neg\neg p \rightarrow p$. The converse: $p \rightarrow \neg\neg p$ still holds. The exclusion of these axioms makes some other well-known identities invalid as well. Only three of the four De Morgan's laws hold in intuitionistic logic. $\neg(p \wedge q) \rightarrow \neg q \vee \neg p$ is no longer valid in intuitionistic logic. This does not mean that we can not use our knowledge from classical logic. Glivenko's Theorem states that a normal prove in classical logic leads to a proof under double negation in intuitionistic logic: $T \models_{clas} S \Leftrightarrow T \models_{int} \neg\neg T$. We also have that intuistionistic logic acts the same as classical logic under negation: $T \models_{clas} \neg S \Leftrightarrow T \models_{int} \neg S$ For more information the reader is referred to (van Atten, 2014).

5 Calculating $\#[\neg, \vee, p, q, r]_{int}$

We know that intuitionistic logic acts like classical logic under negation. This is useful for construction of a normal form for intuitionistic logic. For this we will first take a look at classical logic. A normal form of the classical fragment $[\neg, \vee, \wedge, P]$ is the conjunctive normal form $C_P(X)$. This normal form uses the fact that all possible formulae in $[\neg, \vee, \wedge, P]$ can be written as conjunctions of disjunctions of literals. Here we take $X \in \wp(\wp(P))$. X represents all combinations of atoms for which the formula is true. We have

$$C_P(X) = \bigwedge_{Q \in X} (\bigvee_{p \in Q} p \ \vee \bigvee_{p \in P-Q} \neg p)$$

Using De Morgan's law, $p \wedge q \equiv \neg(\neg p \vee \neg q)$, we can eliminate all occurrences of $\wedge$ and have a normal form $D_P(X)$ using only the connectives $\vee$ and $\neg$.

$$D_P(X) = \neg \bigvee_{Q \in X} \neg(\bigvee_{p \in Q} p \ \vee \bigvee_{p \in P-Q} \neg p)$$

We can use this normal form to construct a normal form for $[\neg, \vee, P]_{int}$. We take $Q \subseteq P$ and $\mathcal{A} \in \wp(\wp(\wp(P)))$. As the part under negation acts as classical logic under negation we can use that for our normal form

$$N_P(Q, \mathcal{A}) = \bigvee_{p \in Q} p \ \vee \bigvee_{X \in \mathcal{A}} D_P(X)$$

The $N_P(Q, \mathcal{A})$ are the unique normal forms for $[\neg, \vee]$ if Q and $\mathcal{A}$ are chosen such that $Q \cup \mathcal{A}$ is an upward closed subset in $S(P) = P \cup \wp(\wp(P))$ with the following order $\leq$. Let $p, q \in P$ and $X, Y \in \mathcal{A}$, then

$$\begin{aligned} X \leq Y &\equiv X \subseteq Y \\ p \leq X &\equiv X = \wp(P) \\ X \leq p &\equiv X \subseteq \overline{p} \\ p \leq q &\equiv p = q \end{aligned}$$

where

$$\begin{aligned} \bar{\cdot} &: P \to \wp(\wp(P)) \\ \overline{p} &= \{R \mid \{p\} \subseteq R \subseteq P\} \end{aligned} \tag{4}$$

The set $\overline{p}$ is the of all subsets of P that contain p. When we look at $C_P(\overline{p})$ we can see that $C_P(\overline{p}) \equiv p$.
The set P has n elements, so $\wp(P)$ has 2^n elements. If we label these elements from 0 to $2^n - 1$ we can see that $\wp(\wp(P)) \cong Q(2^n)$. For calculating $\#[\neg, \vee, P_n]_{int}$ we need to add some elements to the powerset $Q(2^n)$ used for the computation

of the Dedekind number d_{2^n}. The elements that are added, $a_0, ...a_{n-1}$, have a relation with α's defined as:

$$\alpha_n(i) = \{k < 2^n \mid \text{bit}_n(i,k) = 1\} \text{ for } i = 0...n-1 \tag{5}$$

where $bit_n(i,k)$ is the i'th in the binary bit representation of k, where we number bit 0 as the least significant bit. We now construct $C(n)$ as:

$$C(n) = Q(2^n) \cup \{a_0, .., a_{n-1}\}$$

Note that $\{a_0, ..., a_{n-1}\} \sim P$ and $\alpha_n(0), ..., \alpha_n(n-1) \in Q(2^n)$. We extend the order $\leq$ by

$$\emptyset \leq a_i \leq \alpha_n(i) \text{ for } i = 0...n-1$$

We define

$$E(n) = \mathcal{M}^+(C(n))$$

such that $\#[\neg, \vee, P_n]_{int} = \#E(n)$.
There are two methods for the calculation of $\#E(n)$ which will be discussed here.

5.1 First method

We have

$$\begin{aligned}
E(n) = &\quad \mathcal{M}^+(C(n)) \\
= &\ \langle \text{ Definition of } C(n))\rangle \\
&\quad \mathcal{M}^+(Q(2^n) \cup \{a_0, .., a_{n-1}\}) \\
= &\ \langle \text{ Upwards closed in } C(n)\rangle \\
&\quad \{X \cup Y \mid X \in \mathcal{M}^+(Q(2^n)), Y \subseteq \{a_0, .., a_{n-1}\}, \\
&\qquad \forall i < n((\emptyset \in X \to a_i \in Y) \& (a_i \in Y \to \alpha_n(i) \in X))\} \\
= &\quad \{C(n)\} \cup \{X \cup Y \mid X \in \mathcal{M}^{\pm}(Q(2^n)), Y \subseteq \{a_i \mid \alpha_n(i) \in X\}\}
\end{aligned}$$

knowing this we can do the following calculation

$$\begin{aligned}
\#[\neg, \vee, P_n]_{int} &= \#(\{C(n)\} \cup \{X \cup Y \mid X \in \mathcal{M}^{\pm}(Q(2^n)), Y \subseteq \{a_i \mid \alpha_n(i) \in X\}\}) \\
&= 1 + \sum_{X \in \mathcal{M}^{\pm}(Q(2^n))} 2^{\#\{i \mid \alpha_n(i) \in X\}} \\
&= \sum_{X \in D(2^n)} 2^{\#\{i \mid \alpha_n(i) \in X\}} - 2^n \\
&= \sum_{k=0}^{n} \binom{n}{k} 2^k \delta(n,k) - 2^k
\end{aligned} \tag{6}$$

where

$$\delta(n,k) = \#\{X \in D(2^n) \mid \forall i < n(\alpha_n(i) \in X \leftrightarrow i < k)\} \tag{7}$$

We can also use δ to calculate d_{2^n}.

$$d_{2^n} = \sum_{k=0}^{n} \binom{n}{k} \delta(n,k)$$

This can be used to check the computed values of $\delta(n,k)$

5.1.1 Symmetries in δ

A consequence of the definition of $\delta(n,k)$ is that it has some symmetries. We claim

$$\delta(n,k) = \delta(n,n-k) \tag{8}$$

To prove this we start with a few definitions.

$$\begin{aligned}
Q^c &= P - Q \\
X^c &= \wp(P) - X \\
\mathcal{A}^c &= \wp(\wp(P)) - \mathcal{A} \\
X^{\underline{c}} &= \{Q \mid Q^c \in X\} \\
\mathcal{A}^{\underline{c}} &= \{X \mid X^c \in \mathcal{A}\} \\
\mathcal{A}^{\underline{\underline{c}}} &= \{X \mid X^{\underline{c}} \in X\} \\
\mathcal{A}^u &= \{X \mid \exists Y \subseteq X, Y \in \mathcal{A}\} \\
\overline{Q} &= \{\overline{p} \mid p \in Q\}
\end{aligned}$$

Also recall the definition of $\overline{p}$ from (4). If we take a closer look at these operations we see that $\cdot^c$, $\cdot^{\underline{c}}$ and $\cdot^{\underline{\underline{c}}}$ are involutions:

$$(\mathcal{A}^c)^c = (\mathcal{A}^{\underline{c}})^{\underline{c}} = (\mathcal{A}^{\underline{\underline{c}}})^{\underline{\underline{c}}} = \mathcal{A}$$

These same operations also commute. Furthermore we claim,

$$\begin{aligned}
\mathcal{A}^{u\ c\ \underline{c}\ u} &= \mathcal{A}^{u\ c\ \underline{c}} \\
\mathcal{A}^{u\ \underline{\underline{c}}} &= \mathcal{A}^{\underline{\underline{c}}\ u} \\
\overline{p}^{c\ \underline{c}} &= \overline{p}
\end{aligned}$$

Now we are going to prove a slightly more general case than our original problem,

$$\#\{\mathcal{A} \in D(2^n) \mid \mathcal{A} \cap \overline{P}\} = \#\{\mathcal{A} \in D(2^n) \mid \mathcal{A} \cap \overline{P} = \overline{P - Q}\} \tag{9}$$

For this we define a function $f : \wp(\wp(\wp(P))) \to \wp(\wp(\wp(P)))$. It is defined by $f(X) = X^{c\ \underline{c}\ \underline{\subseteq}}$. By the properties above, f is also an involution. Another thing to note is $\mathcal{A} = \mathcal{A}^u \Rightarrow f(\mathcal{A}) = f(\mathcal{A})^u$, for

$$f(\mathcal{A}) = \mathcal{A}^{c\ \underline{c}\ \underline{\subseteq}} = \mathcal{A}^{u\ c\ \underline{c}\ \underline{\subseteq}} = \mathcal{A}^{u\ c\ \underline{c}\ u\ \underline{\subseteq}} = \mathcal{A}^{u\ c\ \underline{c}\ \underline{\subseteq}\ u} = \mathcal{A}^{c\ \underline{c}\ \underline{\subseteq}\ u} = f(\mathcal{A})^u$$

So f is a bijection on $D(2^n)$. From $\overline{p}^{c\ \underline{c}} = \overline{p}$ we can get

$$\overline{p} \in \mathcal{A} \Leftrightarrow \overline{p} \in \mathcal{A}^{\underline{c}\ \underline{\subseteq}}$$

And finally we have

$$X \cap \overline{P} = \overline{Q} \Leftrightarrow X \cap \overline{P} = \overline{P - Q}$$

This last statement is what we need to conclude the proof of (9). By choosing the appropriate P we can prove (8) from (9). As a consequence, we only need to calculate half of the δs.

5.1.2 Computing δ

To compute δ we split up $S \in D_{2^n}$ in the same way as (2). There are a few important things to note here. The sets C and D are the only sets with elements that contain $2^n - 1$. We also observe that the elements in C do not contain $2^n - 2$. Now we take a look at the $\alpha_n(i)$ which are defined in (5). The bit representation of $2^n - 1$ is rather simple, it is just a sequence of n 1's. From this we can conclude that $2^n - 1$ is always in $\alpha_n(i)$. With similar reasoning we observe that 0 never appears in $\alpha_n(i)$. To make use of these facts we introduce a permutation $\sigma : \{0, 1, ..., n-1\} \to \{0, 1, ..., n-1\}$, defined by

$$\sigma(x) = \begin{cases} 0 & \text{if } x = 2^n - 2 \\ 2^n - 2 & \text{if } x = 0 \\ x & \text{otherwise} \end{cases}$$

We lift σ to sets al follows: $\sigma[T] = \{\sigma(t) \mid t \in T\}$. With this we will now permute $\alpha_n(i)$ to $\alpha'_n(i)$.

$$\alpha'_n(i) = \sigma[\alpha_n(i)]$$

Since $0 \notin \alpha_n(i)$, we have that $2^n - 2 \notin \alpha'_n(i)$. With this information and the definition of C we can conclude that $\alpha'_n(i) \ominus 2^n - 1$ will always be in C if $\alpha_n(i) \in f(A, B, C, D)$. Now we define

$$A_n = \{\alpha_n(i) \mid i < n\}$$
$$A'_n = \{\alpha'_n(i) \mid i < n\}$$
$$A'^{-}_n = \{\alpha'_n(i) \ominus 2^n - 1 \mid i < n\}$$

By intersecting these sets with $X \in D_2^n$ we can check which $\alpha' s$ are contained in X.

$$
\begin{aligned}
\delta(n,k) = \quad & \#\{X \in D_{2^n} \mid X \cap A_n = \{\alpha(i) \mid i < k\}\} \\
= \ & \langle \text{ permuting 0 and } 2^n - 1 \rangle \\
& \#\{X \in D_{2^n} \mid X \cap A'_n = \{\alpha'(i) \mid i < k\}\} \\
= \ & \langle \text{ applying (2)} \rangle \\
& \#\{(A,B,C,D) \in (D_{2^n-2})^4 \mid A \subseteq (B \cap C), D \subseteq (B^* \cap C^*) \\
& \qquad f(A,B,C,D) \cap A'_n = \{\alpha'(i) \mid i < k\}\} \\
= \ & \langle \forall \alpha' \ \alpha' \ominus (2^n - 1) \in C \rangle \\
& \#\{(A,B,C,D) \in (D_{2^n-2})^4 \mid A \subseteq (B \cap C), D \subseteq (B^* \cap C^*) \\
& \qquad C \cap A'^{-}_n = \{\alpha'(i) \ominus (2^n - 1) \mid i < k\}\} \\
= \ & \langle \text{ dismissing } A \text{ and } D \rangle \\
& \sum_{C \in D'_{2^n-2}(k)} \sum_{B \in D_{2^n-2}} \eta(C \cap B) \cdot \eta(C^* \cap B^*) \\
= \ & \langle D_n = \{C \mid \exists K \in R_n, C \sim K\} \ \rangle \\
& \sum_{K \in R_{2^n-2}} \sum_{\pi \in p'(K,k)} \sum_{B \in D_{2^n-2}} \eta(\pi(K) \cap B) \cdot \eta(\pi(K)^* \cap B^*) \\
= \ & \langle \text{ applying the permutations in } p(K) \rangle \\
& \sum_{K \in R_{2^n-2}} \sum_{\pi \in p'(K,k)} \sum_{B \in D_{2^n-2}} \eta(\pi(K) \cap \pi(B)) \cdot \eta(\pi(K)^* \cap \pi(B)^*) \\
= \ & \langle \forall \pi \ D_n = \{\pi(S) \mid S \in D_n\} = \pi[D_n] \rangle \\
& \sum_{K \in R_{2^n-2}} \sum_{\pi \in p'(K,k)} \sum_{B \in D_{2^n-2}} \eta(\pi(K \cap B)) \cdot \eta(\pi(K^* \cap B^*)) \\
= \ & \langle \eta(X) = \eta(\pi(X)) \rangle \\
& \sum_{K \in R_{2^n-2}} \sum_{B \in D_{2^n-2}} \gamma(K,k) \cdot \eta(K \cap B) \cdot \eta(K^* \cap B^*) \\
= \ & \langle \text{ rearranging} \rangle \\
& \sum_{K \in R_{2^n-2}} \gamma(K,k) \cdot \sum_{B \in D_{2^n-2}} \eta(K \cap B) \cdot \eta(K^* \cap B^*)
\end{aligned}
$$

where

$$
\begin{aligned}
D'_{2^n-2}(k) &= \{A \in D_{2^n-2} \mid \alpha'(i) \ominus (2^n - 1) \in A, i < k, \alpha(j)' \ominus (2^n - 1) \notin Aj \geq k\} \\
p'(K,k) &= \{\pi \in p(K) \mid \alpha'(i) \ominus (2^n - 1) \in \pi(K), i < k, \alpha(j)' \ominus (2^n - 1) \notin \pi(K)j \geq k\} \\
\gamma(K,k) &= \#\{A \sim K \mid \alpha'(i) \ominus (2^n - 1) \in A, i < k, \alpha(j)' \ominus (2^n - 1) \notin Aj \geq k\}
\end{aligned}
$$

With the δ's calculated we can now compute ζ_n. To do this we use the formula (6) which only needs some simple computations.

$$\zeta_n = \sum_{k=0}^{n} \binom{n}{k} 2^k \delta(n,k) - 2^k$$

5.2 Second method

The second method is easier to understand. If X is monotonic in $Q(2^n)$ then there is an easy way to tell how many elements it will generate for $E(n)$. We know that $\emptyset$ is the only subset of a_i for all $i = 1...n-1$. So. if $\alpha_n(i) \notin X$ then $X \cup \{a_i\}$ is monotonic in $C(n)$. This holds for all a's. This means we need to count how many α's are not in X.
As seen in the first method we have

$$\begin{aligned}
\#[\neg, \vee, P_n]_{int} &= \#(\{C(n)\} \cup \{X \cup Y \mid X \in \mathcal{M}^{\pm}(Q(2^n)), Y \subseteq \{a_i \mid \alpha_n(i) \in X\}\}) \\
&= 1 + \sum_{X \in \mathcal{M}^{\pm}(Q(2^n))} 2^{\#\{i \mid \alpha_n(i) \in X\}} \\
&= \sum_{X \in D(2^n)} 2^{\#\{i \mid \alpha_n(i) \in X\}} - 2^n
\end{aligned}$$

Now we will define δ' so it fits the description above.

$$\delta'(n,k) = \#\{X \in D(2^n) \mid \#\{X \cap A_n\} = k\}$$

Using this we get

$$\begin{aligned}
\#[\neg, \vee, P_n]_{int} &= \sum_{X \in D(2^n)} 2^{\#\{i \mid \alpha_n(i) \in X\}} - 2^n \\
&= \sum_{k=0}^{n} 2^{n-k} \cdot \delta'(n,k) - 2^n
\end{aligned}$$

where With this δ' we can calculate d_{2^n} as well. $d_{2^n} = \sum_{k=0}^{n} \delta'(n,k)$.

5.2.1 calculation δ'

We will split up D_{2^n} into four parts just as we did before. The whole calculation is pretty similar.

$$\begin{aligned}
\delta'(n,k) = \quad & \#\{X \in D_{2^n} \mid \#(X \cap A_n) = k\} \\
= \ & \langle \text{ permuting 0 and } 2^n - 1 \rangle \\
& \#\{X \in D_{2^n} \mid \#(X \cap A'_n) = k\} \\
= \ & \langle \text{ applying (2)} \rangle \\
& \#\{(A,B,C,D) \in (D_{2^n-2})^4 \mid A \subseteq (B \cap C), D \subseteq (B^* \cap C^*) \\
& \qquad \#(f(A,B,C,D) \cap A'_n) = k\} \\
= \ & \langle \forall \alpha' \ \alpha' \ominus (2^n - 1) \in C \rangle \\
& \#\{(A,B,C,D) \in (D_{2^n-2})^4 \mid A \subseteq (B \cap C), D \subseteq (B^* \cap C^*) \\
& \qquad \#(C \cap A'^-_n) = n\} \\
= \ & \langle \text{ dismissing } A \text{ and } D \rangle \\
& \sum_{C \in D''_{2^n-2}(k)} \sum_{B \in D_{2^n-2}} \eta(C \cap B) \cdot \eta(C^* \cap B^*) \\
= \ & \langle D_n = \{C \mid \exists K \in R_n, C \sim K\} \ \rangle \\
& \sum_{K \in R_{2^n-2}} \sum_{\pi \in p''(K,k)} \sum_{B \in D_{2^n-2}} \eta(\pi(K) \cap B) \cdot \eta(\pi(K)^* \cap B^*) \\
= \ & \langle \text{ applying the permutations in } p(K) \rangle \\
& \sum_{K \in R_{2^n-2}} \sum_{\pi \in p''(K,k)} \sum_{B \in D_{2^n-2}} \eta(\pi(K) \cap \pi(B)) \cdot \eta(\pi(K)^* \cap \pi(B)^*) \\
= \ & \langle \forall \pi \ D_n = \{\pi(S) \mid S \in D_n\} = \pi[D_n] \rangle \\
& \sum_{K \in R_{2^n-2}} \sum_{\pi \in p''(K,k)} \sum_{B \in D_{2^n-2}} \eta(\pi(K \cap B)) \cdot \eta(\pi(K^* \cap B^*)) \\
= \ & \langle \eta(X) = \eta(\pi(X)) \rangle \\
& \sum_{K \in R_{2^n-2}} \sum_{B \in D_{2^n-2}} \gamma'(K,k) \cdot \eta(K \cap B) \cdot \eta(K^* \cap B^*) \\
= \ & \langle \text{ rearranging} \rangle \\
& \sum_{K \in R_{2^n-2}} \gamma'(K,k) \cdot \sum_{B \in D_{2^n-2}} \eta(K \cap B) \cdot \eta(K^* \cap B^*)
\end{aligned}$$

where

$$\begin{aligned}
D''_{2^n-2}(k) &= \{A \in D_{2^n-2} \mid \#(A \cap A'^-_n) = k\} \\
p''(K,k) &= \{\pi \in p(K) \mid \#(\pi(K) \cap A'^-_n) = k)\} \\
\gamma'(K,k) &= \#\{A \sim K \mid \#(A \cap A'^-_n) = k\}
\end{aligned}$$

Now we have all the things we need to calculate ζ_n:

$$\zeta_n = \sum_{k=0}^{n} 2^{n-k}\delta'(n,k) - 2^k$$

6 Implementation

The implementation is a continuation of the implementation to calculate d_n by Zijlstra (2013). We will take a look at some helper functions in section (6.1). Using these we can then take a look at the main functionality in section (6.2).

6.1 Helper Functions

6.1.1 Bitset

It is important to know the structures used for all different forms of sets. A monotonic subset is implemented as a `std::bitset` of the size 2^n. This is done in such a way that each bit represents an element in $Q(n)$. As an example, for $n = 2$ we have $Q(n) = \{01, 1, 0, \emptyset\}$. Then the monotonic subset 0101 would be $\{1, \emptyset\}$. Also note $Q(n)$ is sorted lexicographically. A useful effect of the ordering is that we can find the index of a specific element really easily. An element $\{a_0, a_1, ..., a_n\}$ within $Q(n)$ can be found at index $\sum_{i=0}^{n} 2^{a_i}$.

6.1.2 UInt128

The number ζ_3 is larger than d_8, but it is still less than 2^{128}. This means that the result will need a 128-bit number as well. To do this we use the `UInt128` class. This is a class containing two `uint_fast64_t`'s as it is the fastest 64-bits data type. A previous implementation only implemented the + and the << (for printing the number) operator.
We will need two extra operations. The first one we need is the − operation. At the end 2^n needs to be subtracted. As 2^n is a lot smaller then 2^{64} we only need to implement `UInt128` minus `uint_fast64_t`. The implementation is really simple. We only need to check if we need to borrow from the higher part.
The other operator we need is the $*$ operation for multiplication. We only need to implement the `UInt128` times `uint_fast64_t` variant as we won't need to multiply by large numbers. A very simple approach is taken for multiplication as it is called less than n times when calculation ζ_n. We iterate over every bit of the number in `uint_fast64_t` and multiply by using bit shifting and addition. As `UInt128` is stored in two `uint_fast64_t` we have to pay attention to overflowing from the lower part. We need to add the overflow of the lower part to the higher part.

Listing 1: `project/uint128/operatormultiply.cpp`

```
1  #include "uint128.ih"
2
3  namespace Dedekind
4  {
5      UInt128 &UInt128::operator*=(uint_fast64_t other)
6      {
7          UInt128 temp(*this);
```

```
8           d_lo =0; d_hi=0;
9           for(uint_fast64_t i=0;i<64;i++){
10              if((other&1)!=0){
11                  *this+= (temp.d_lo<<i);
12                  d_hi+= (temp.d_hi<<i);
13                  if ( i!= 0) //can't bitshift 64 bits
14                      d_hi+= (temp.d_lo>>(64-i));
15              }
16              other>>=1;
17          }
18          return *this;
19      }
20  }
```

6.1.3 Subsets

The <= operator for sets checks whether the left-hand side is a subset of the right-hand side. We have implemented the sets as bitsets using one bit for every element. For lhs to be a subset of rhs we then need to validate that every set bit for the lhs is also set for the rhs. This translates to $\neg lhs \vee rhs$ being true for all bits.

Listing 2: `project/dedekind/bitsetoperleq.h`

```
1  bool operator<=(std::bitset<size> lhs, std::bitset<size> const &rhs)
2  {
3      return (lhs.flip() | rhs).all();
4  }
```

6.1.4 Concatenations

Concatenation of bitsets is needed in generating D_n. There is no default operator for this. To concatenate two bitsets we convert the bitsets to strings and then concatenate the strings. Then a new bitset is constructed using the bitset constructor and the new string.

Listing 3: `project/dedekind/operwiedemann.h`

```
1  template <size_t size>
2  std::bitset<(size << 1)> concatenate(std::bitset<size> const &lhs,
3          std::bitset<size> const &rhs)
4  {
5      std::string lhs_str = lhs.to_string();
6      std::string rhs_str = rhs.to_string();
7
8      return std::bitset<(size << 1)>(lhs_str + rhs_str);
9  }
```

6.1.5 Power Set

The construction of power sets is done using template-meta-programming. This decreases the running time by doing parts of the calculation at compile time.

Listing 4: `project/dedekind/powersetbin.h`

```
template <size_t size>
struct PowerSet
{
    static std::vector<std::bitset<size>> powerSetBin();
};

template <size_t size>
std::vector<std::bitset<size>> PowerSet<size>::powerSetBin()
{
    auto current = PowerSet<size - 1>::powerSetBin();

    std::vector<std::bitset<size>> result;
    for (auto iter = current.begin(); iter != current.end(); ++iter)
    {
        std::bitset<size> tmp((*iter).to_ulong() + (1 << (size - 1)));
        result.push_back(tmp);
    }

    for (auto iter = current.begin(); iter != current.end(); ++iter)
    {
        std::bitset<size> tmp((*iter).to_ulong());
        result.push_back(tmp);
    }

    return result;
}


template <>
struct PowerSet<0>
{
    static std::vector<std::bitset<0>> powerSetBin();
};

std::vector<std::bitset<0>> PowerSet<0>::powerSetBin()
{
    return std::vector<std::bitset<0>>({ std::bitset<0>() });
}
```

6.1.6 Permutation

Permutations are implemented using an array of indexes. This array holds the indexes which will be permuted.

Listing 5: `project/dedekind/permutations.h`

```
template <size_t size>
std::bitset<size> permute(std::array<size_t, size> const &permutation,
        std::bitset<size> const &elem)
{
    std::bitset<size> result;
    for (size_t idx = 0; idx != result.size(); ++idx)
    {
        result[idx] = elem[permutation[idx]];
    }
    return result;
}
```

6.1.7 Equivalence classes

To generate all equivalent elements one needs to do all permutations over an element. We do not want to have copies so we need a set as a container.

Listing 6: `project/dedekind/permutations.h`

```
template <size_t size>
std::set<std::bitset<size>, BitSetLess> equivalences(
        std::bitset<size> const &bset,
        std::vector<std::array<size_t, size>> const &perms)
{
    std::set<std::bitset<size>, BitSetLess> result;
    for (auto iter = perms.begin(); iter!= perms.end(); ++iter)
    {
        std::bitset<size> temp = permute(*iter, bset);
        result.insert(temp);
    }
    return result;
}
```

6.1.8 Sorting bitset

Bitsets are sorted on their integer value. This way the highest value will be output first when printing a bitset. This follows the structure of lattices.

Listing 7: `project/dedekind/bitsetless.h`

```
class BitSetLess
{
    public:
        template<size_t size>
        bool operator()(std::bitset<size> const &lhs,
                std::bitset<size> const &rhs) const
        {
            return lhs.to_ulong() > rhs.to_ulong();
        }
};
```

6.1.9 generateIndexes

The generateIndexes function has to generate the indexes of the α_n's. The index of a set $\{a_0, a_1, ..., a_n\}$ can be found is $\sum_{i=0}^{n} 2^{a_i}$. To recall our choice of α's

$$\alpha_n(i) = \{k < 2^n \mid \mathrm{bit}_n(i,k) = 1\}$$

To calculate the indices belonging to these sets we loop from 0 to $2^n - 1$, check whether the number is an element of the set and if so add the appropriate amount to the index. Checking if a bit is set is a rather cheap operation. Another thing we need to consider is that the α's are permuted once. The element 0 and $2^n - 1$ are permuted and we have to make up for that. To do this a special case is made which adds 1 (2^0) to the index if $2^n - 1$ is found.

Listing 8: project/dedekind/dedekind.h

```
std::vector<size_t> generateIndexes(size_t n = 2)
{
    std::vector<size_t> indexes(n,0);
    for(size_t x = 1; x<n;x++)
    {
        indexes[x]=0;
    }
    for(size_t x = 0; x<(1<<n)-1;x++)
    {
        for(size_t i = 0;i<n;i++)
        {
            if(x & (1<<i))
            {
                if( x == ((1<<n)-2)) indexes[i]+=1;
                else indexes[i] += (1<<x);
            }
        }
    }
    return indexes;
}
```

6.1.10 generateGammas method 1

To calculate the γ's for the first method we need to do two things. First we need to count how many α's in a row are in the element of R_n. The second thing we need to check is whether the rest of the α's are not in the element.

Listing 9: project/dedekind/dedekind.h

```
template <size_t size>
std::vector<size_t> calculateGammas(std::vector<std::bitset<size>> const
     elem,size_t n, std::vector<size_t> indexes){
    size_t length = n/2 + 1;
    std::vector<size_t>  result(length,0);
    for(auto iter = elem.begin(); iter != elem.end(); ++iter)
    {
        bool valid = true;
        size_t counter = 0;
        size_t secondcounter = 0;
        for (auto it = indexes.begin(); it != indexes.end(); ++it)
        {
            if ((*iter).test(*it))
            {
                if (counter == secondcounter) counter++;
                else valid = false;
            }
            secondcounter++;
        }
        if( valid == true && counter <= length-1 )
        {
            result[counter]++;
        }
    }
    return result;
}
```

6.1.11 generateGammas method 2

The way to calculate the γ's for the second method is way easier. It is simply counting how many α's are contained by each element.

Listing 10: `project-alt/dedekind/dedekind.h`

```
template <size_t size>
std::vector<size_t> calculateGammas(std::vector<std::bitset<size>> const
     elem,size_t n, std::vector<size_t> indexes){
    size_t length = n + 1;
    std::vector<size_t>  result(length,0);
    for(auto iter = elem.begin(); iter != elem.end(); ++iter)
    {
        size_t counter =0;
        for (size_t i = 0; i < n; i++)
        {
            if ((*iter).test(indexes[i]))
            {
                counter++;
            }
        }
        result[counter]++;
    }
    return result;
}
```

6.2 Main Functions

6.2.1 Calculating D_{n+1} from D_n

To generate D_{n+1} from D_n we use the method we explained earlier in section 4.1. To do this we have to loop twice over D_n. For this we can use Iterators as D_n is a vector. Furthermore we need to test if two elements of D_n are qualified for creating an element in D_{n+1}. We can check whether these are subsets using the $<=$ operator defined earlier. If it is a subset then we can use the concatenation of the two elements as a new element of D_{n+1}.

Listing 11: `project/dedekind/dedekind.h`

```
template <size_t size>
std::vector<std::bitset<(size << 1)> > generate(
        std::vector<std::bitset<size> > const &dn)
{
    std::vector<std::bitset<(size << 1)> > dn1;

    for (auto iter = dn.begin(); iter != dn.end(); ++iter)
    {
        for (auto iter2 = dn.begin(); iter2 != dn.end(); ++iter2)
        {
            if (*iter <= *iter2)
            {
                dn1.push_back(Internal::concatenate(*iter, *iter2));
            }
        }
    }

```

```
18          return dn1;
19      }
```

6.2.2 Calculating R_n

R_n is mostly about permutations. For this we use the permutation functions defined earlier. First of all we need to find all permutations using these functions. Once all the permutations are generated we can make the equivalent classes. For each element of D_n the equivalent class is generated, which is stored in a vector. This vector will then be added to R_n. The whole vector is added, as the elements are useful for generation the η-values. A set of all processed subsets is also made to keep track of which subsets have already been processed. This is needed to prevent duplicate entries.

Listing 12: `project/dedekind/dedekind.h`

```
1   template <size_t Number, size_t Power>
2   std::vector<std::vector<std::bitset<Power>>> generateRn(
3           std::vector<std::bitset<Power>> const &dn)
4   {
5       auto permutations = Internal::permutations<Number, Power>();
6
7       std::vector<std::vector<std::bitset<Power>>> rn;
8       std::set<std::bitset<Power>, BitSetLess> processed;
9       for (auto iter = dn.begin(); iter != dn.end(); ++iter)
10      {
11          if (processed.find(*iter) == processed.end())
12          {
13              auto equivs = Internal::equivalences(*iter, permutations);
14
15              std::vector<std::bitset<Power>> permuted;
16              copy(equivs.begin(), equivs.end(), std::back_inserter(
                    permuted));
17              for (auto perm = equivs.begin(); perm != equivs.end(); ++perm
                    )
18              {
19                  processed.insert(*perm);
20              }
21
22              rn.push_back(permuted);
23          }
24      }
25
26      return rn;
27  }
```

6.2.3 Calculating d_{n+2} using D_n and R_n

Now we have everything to calculate d_{n+2} using MPI. First of we can do the preprocessing. The η and dual values will be needed multiple times so is faster to calculate them beforehand. To store the values a map is used. Maps have a fast lookup speed which is the most important thing for us. Both maps have the corresponding `vector<bitset>` as key. The η map returns an integer value

and the dual map return a `vector<bitset>`. The function as a whole can not return just an `int`. The number d_8 needs more than 64 bits. To do this a class `UInt128` is used which is a 128-bit unsigned integer. The first loop is the loop over R_n. The second loop loops over D_n.
Parallelization is done in the outer loop. This choice is made for it simplicity. It appears that this does not cause a severe efficiency loss as the difference between the fastest and slowest process is measured to be 5%. To do the parallelization static scheduling is used. Instead of increasing the iterator in the loop by one it is increased by the number of processes. This way each processor gets about equal load.

Listing 13: `bachelor-project/dedekind/dedekind.h`

```
1   template <size_t size>
2   UInt128 enumerate(std::vector<std::bitset<size>> const &dn,
3           std::vector<std::vector<std::bitset<size>>> const &rn,
4           size_t rank = 0, size_t nprocs = 1)
5   {
6       std::map<std::bitset<size>, std::bitset<size>, BitSetLess> duals;
7       std::map<std::bitset<size>, size_t, BitSetLess> etas;
8
9       // Preprocess duals and eta's of all elements
10      for (auto iter = rn.begin(); iter != rn.end(); ++iter)
11      {
12          auto elem = (*iter).begin();
13          size_t tmp = Internal::eta(*elem, dn);
14          for (; elem != (*iter).end(); ++elem)
15          {
16              etas[*elem] = tmp;
17              duals[*elem] = Internal::dual(*elem);
18          }
19      }
20      // Preprocessing complete
21
22
23      UInt128 result;
24      for (size_t idx = rank; idx < rn.size(); idx += nprocs)
25      {
26          auto iter(rn[idx].begin());
27          for (auto iter2 = dn.begin(); iter2 != dn.end(); ++iter2)
28          {
29              auto first = *iter & *iter2;
30              auto second = duals[*iter] & duals[*iter2];
31
32              result += rn[idx].size() * etas[first] * etas[second];
33          }
34      }
35
36      return result;
37  }
```

6.2.4 Calculation ζ_n Method 1

The methods used for calculating ζ_n are very similar to those of d_n. A few extra things need to be calculated to get the correct result. First off we need

to check whether certain bitsets are contained within all of the indexes of $A_n'^-$ can easily be calculated due to the lexicographical order. This is done in `generateIndexes`. This is a vector containing the indexes of all the α's. Note that γ is not the size of an element of R_n any longer as was the case with the calculation of d_n. Now it also needs to check if some sets are contained in it. For this a new function is used `calculateGammas`. This will return a vector that will take an element of R_n and will calculate all the γ-values for that element.

Listing 14: `project/dedekind/dedekind.h`

```
template <size_t size>
UInt128* enumerate(std::vector<std::bitset<size> > const &dn,
        std::vector<std::vector<std::bitset<size> > > const &rn,
        size_t rank = 0, size_t nprocs = 1, size_t n= 2 )
{
    std::map<std::bitset<size>, std::bitset<size>, BitSetLess> duals;
    std::map<std::bitset<size>, size_t, BitSetLess> etas;
    size_t length = n/2 + 1;
    // Preprocess gamma's, duals and eta's of all elements
    std::vector<size_t> indexes = generateIndexes(n);
    for (auto iter = rn.begin(); iter != rn.end(); ++iter)
    {

        auto elem = (*iter).begin();
        size_t tmp = Internal::eta(*elem, dn);
        for (; elem != (*iter).end(); ++elem)
        {
            etas[*elem] = tmp;
            duals[*elem] = Internal::dual(*elem);
        }
    }


    UInt128* result;
    result = new UInt128[length];
    for (size_t idx = rank; idx < rn.size(); idx += nprocs)
    {
        auto iter(rn[idx].begin());
        std::vector<size_t> gamma =  calculateGammas(rn[idx],n,indexes);
        for (auto iter2 = dn.begin(); iter2 != dn.end(); ++iter2)
        {
            auto first = *iter & *iter2;
            auto second = duals[*iter] & duals[*iter2];

            for(size_t i=0;i<length;i++)
            {
            result[i] += gamma[i] * etas[first] * etas[second];
            }
        }
    }

    return result;
}
```

At the end all the answers need to be summed up using the correct factors (which were all 1 in d_n cases). The calculation of d_n is also made, because it can be done in little amount of time and it adds a check for correctness.

Listing 15: `project/main.cpp`

```
            for(size_t i=0;i<=n;i++)
            {
                uint_fast64_t temp = (factorial(n) / (factorial(n-i)*
                    factorial(i)));

                size_t index = (i>n/2 ? n-i:i);
                dedekindnumber += (result[index]) * temp;
                othernumber += (result[index]) * (temp*(1<<i));
            }
            othernumber -= (1<<n);
```

6.2.5 Calculation ζ_n Method 2

Method 2 is nearly the same as method 1. The main difference is the γ's. The function `calculateGammas` is much easier with this method as it is simply counting how many α's occur. It does not matter which ones.

Listing 16: `project-alt/dedekind/dedekind.h`

```
    template <size_t size>
    UInt128* enumerate(std::vector<std::bitset<size> > const &dn,
            std::vector<std::vector<std::bitset<size> > > const &rn,
            size_t rank = 0, size_t nprocs = 1, size_t n= 2 )
    {
        std::map<std::bitset<size>, std::bitset<size>, BitSetLess> duals;
        std::map<std::bitset<size>, size_t, BitSetLess> etas;
        size_t length = n + 1;
        // Preprocess gamma's, duals and eta's of all elements
        std::vector<size_t> indexes = generateIndexes(n);
        for (auto iter = rn.begin(); iter != rn.end(); ++iter)
        {

            auto elem = (*iter).begin();
            size_t tmp = Internal::eta(*elem, dn);
            for (; elem != (*iter).end(); ++elem)
            {
                etas[*elem] = tmp;
                duals[*elem] = Internal::dual(*elem);
            }
        }


        UInt128* result;
        result = new UInt128[length];
        for (size_t idx = rank; idx < rn.size(); idx += nprocs)
        {
            auto iter(rn[idx].begin());
            std::vector<size_t> gamma =  calculateGammas(rn[idx],n,indexes);
            for (auto iter2 = dn.begin(); iter2 != dn.end(); ++iter2)
            {
                auto first = *iter & *iter2;
                auto second = duals[*iter] & duals[*iter2];

                for(size_t i=0;i<length;i++)
                {
                result[i] += gamma[i] * etas[first] * etas[second];
                }
            }
```

```
40          }
41          return result;
42      }
```

The summation at the end does differ as the theory would also suggest.

Listing 17: `project-alt/main.cpp`

```
1               for(size_t i=0;i<=n;i++)
2               {
3                   cout << result[i] << "\n";
4                   dedekindnumber += result[i];
5                   othernumber += result[i] *(1<<(n-i));
6               }
7               othernumber -= (1<<n);
```

7 Results and Discussion

The two separate programs have been run on the millipede cluster of the University of Groningen. This cluster consists of 235 nodes with 12 2.6 GHz AMD Opteron cores and 24GB of memory. The cluster also has 16 nodes with 24 cores and one with 64. The original code for calculating d_8 took 12537 seconds using 12 cores. Using 144 cores resulted in a time of 2670 seconds. The first method to calculate ζ_3 took 2787 seconds on 144 cores. On 12 it was 14147 seconds. The second method took 3165 seconds on 144 cores and 17889 seconds on 12 cores. Both methods yielded the same result. The number $\zeta_3 = 191,589,906,593,484,837,139,681$ was calculated with both methods. To check we also took a look at the calculated d_8 value. With both methods we found the correct answer, $d_8 = 56,130,437,228,687,557,907,788$. There is room for significant reduction of the execution time. The current implementation is using distributed memory. With shared memory the time used on the preprocessing can be lowered significantly. With shared memory the computation of γ and η can be divided over all processors. A different strategy to parallelization could also be tested. One could dynamically divide blocks of calculations. This might improve it a bit even though it would cause some extra overhead. The current static division has about a 5% difference between the first and last finished process.

At this point it is not possible to calculate ζ_4 in a reasonable amount of time. This calculation would be similar to the calculation of d_{16}. As of today nothing higher then d_8 has been calculated. Calculating d_9 is in the realm of possibility at this point in time but would take a huge amount of resources which is not quite worth it. Historically there has been somewhere between 10 and 20 years between the calculation of successive Dedekind numbers. If this trend is extrapolated then it would last between 50 and 100 years until d_{14} is calculated.

Even though ζ_4 is not possible there are other things we can calculate. There are structures which are similar to the structure of $\#[\neg, \vee, P_n]_{int}$. A different fragment, $\#[\neg, \vee, \wedge, P_n]_{int}$, can be calculated with an altered form of the second method.

References

Mark van Atten. The development of intuitionistic logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2014 edition, 2014.

Doug Wiedemann. A computation of the eighth Dedekind number. *Order*, 8 (1):5–6, 1991.

Arjen Teijo Zijlstra. *Calculating the 8th Dedekind Number*. Bachelors thesis, University of Groningen, 2013.

Appendix

Source Code

The source code is based on the code of Zijlstra (2013) which can be found at: `github.com/arjenzijlstra`.
The source code is available at: `github.com/Seremath/bachelor-project`. `main.cpp` had been altered to fit the new computation while retaining the same structure. In `dedekind.h` two new function were added: `calculateGamma` and `generateIndexes`. The `enumerate` function has been changed to fit the new methods. Furthermore `operatorminus` and `operatormultiply` have been added to `UInt128`. The rest of the code remains largely unchanged from the code of Zijlstra (2013).

Method 1

Listing 18: `project/main.ih`

```
1
2  #include <iostream>
3  #include <sstream>
4  #include <mpi.h>
5  #include <sys/time.h>
6
7  #include "dedekind/dedekind.h"
8  #include "uint128/uint128.h"
9
10 using namespace std;
11
12
13 struct timeval tim2;
14
15
16 typedef Dedekind::UInt128* (*fptr)(size_t, size_t);
17
18 template <size_t a = 3>
19 fptr findFunction(size_t b)
20 {
21     if (a == b)
22     {
23         return Dedekind::monotoneSubsets<a>;
24     }
25     else
26     {
27         return findFunction<a - 1>(b);
28     }
29 }
30
31 template <>
32 fptr findFunction<1>(size_t b)
33 {
34     return Dedekind::monotoneSubsets<1>;
35 }
```

Listing 19: project/main.cpp

```
1
2  #include "main.ih"
3
4  uint_fast64_t factorial(size_t n){
5          uint_fast64_t tmp =1;
6          for(size_t i=2;i<=n;i++)
7              tmp *= i;
8          return tmp;
9  }
10
11 int main(int argc, char **argv)
12 {
13     gettimeofday(&tim2, NULL);
14
15     MPI::Init(argc, argv);
16     MPI::COMM_WORLD.Set_errhandler(MPI::ERRORS_THROW_EXCEPTIONS);
17
18     size_t rank = 0;
19     size_t size = 1;
20     try
21     {
22         rank = MPI::COMM_WORLD.Get_rank();
23         size = MPI::COMM_WORLD.Get_size();
24     }
25     catch (MPI::Exception const &exception)
26     {
27         cerr << "MPI␣error:␣" << exception.Get_error_code() << "␣-␣"
28                 << exception.Get_error_string() << endl;
29     }
30
31
32     if (argc == 3 && string(argv[1]) == "-d")
33     {
34         size_t n = 2;
35         stringstream ss(argv[2]);
36         ss >> n;
37
38         double start = MPI::Wtime();
39
40         // findFunction makes it very slow, this can be solved by replacing
               it
41         // with the following, replacing N for the Dedekind number to compute
               .
42         //Dedekind::UInt128* result = Dedekind::monotoneSubsets<2>(rank, size
               );
43         Dedekind::UInt128* result = findFunction(n)(rank, size);
44
45         double end = MPI::Wtime();
46         cerr << "Rank␣" << rank << "␣done!" /* Result: " << result*/ << "␣in␣
               "
47               << end - start << "s\n";
48         size_t length = n/2+1;
49         // reduce over all processes
50         if (rank == 0)
51         {
52             size_t toReceive = (size-1)*length+1;
53             while (--toReceive)
54             {
55                 // send the high and the low part of the result
56                 uint_fast64_t lohi[3];
57                 for( size_t i=0;i<3;i++){
```

```
58                      lohi[i]=0;
59                  }
60                  MPI::Status status;
61                  MPI::COMM_WORLD.Recv(lohi, 3, MPI::UNSIGNED_LONG,
62                          MPI::ANY_SOURCE, Dedekind::BIGINTTAG, status);
63
64                  Dedekind::UInt128 tmp(lohi[0], lohi[1]);
65                  result[lohi[2]] += tmp;
66              }
67
68              double final = MPI::Wtime();
69              Dedekind::UInt128 dedekindnumber=0;
70              Dedekind::UInt128 othernumber =0;
71              for(size_t i=0;i<=n;i++)
72              {
73                  uint_fast64_t temp = (factorial(n) / (factorial(n-i)*
                        factorial(i)));
74
75                  size_t index = (i>n/2 ? n-i:i);
76                  dedekindnumber += (result[index]) * temp;
77                  othernumber += (result[index]) * (temp*(1<<i));
78              }
79              othernumber -= (1<<n);
80              cout << "d" << (1<<n) << "␣=␣" << dedekindnumber << "␣and␣zeta"
                        << n << "␣=␣" << othernumber
81                  << "␣(in␣" << final - start << "s)\n";
82          }
83          else
84          {
85              // send the high and the low part of the result
86              uint_fast64_t lohi[3];
87              for(size_t i=0;i<length;i++){
88                  lohi[0] = result[i].lo();
89                  lohi[1] = result[i].hi();
90                  lohi[2] = i;
91                  MPI::COMM_WORLD.Bsend(&lohi, 3, MPI::UNSIGNED_LONG, 0,
92                          Dedekind::BIGINTTAG);
93              }
94
95              double final = MPI::Wtime();
96              cerr << "Rank␣" << rank << "␣exiting!␣Total:␣"
97                      << final - start << "s\n";
98          }
99      }
100     else
101     {
102         cout << "Usage:␣mpirun␣-np␣N␣./project␣-d␣X␣\n"
103             << "Where␣X␣in␣[2..n)␣is␣the␣Dedekind␣Number␣to␣calculate.\n"
104             << "And␣N␣is␣the␣number␣of␣processes␣you␣would␣like␣to␣use.\n\n"
105             << "Note:␣The␣program␣will␣also␣work␣when␣running␣normally␣"
106             << "(without␣using␣mpirun).\n"
107             << "In␣that␣case␣the␣program␣will␣just␣run␣on␣1␣core.\n";
108     }
109     MPI::Finalize();
110 }
```

Listing 20: project/dedekind/dedekind.h

```
1
2 #ifndef DEDEKIND_H_
3 #define DEDEKIND_H_
4
```

```
5  #include <algorithm>
6  #include <bitset>
7  #include <iostream>
8  #include <map>
9  #include <set>
10 #include <vector>
11
12 #include "../uint128/uint128.h"
13
14 #include "bitsetless.h"
15 #include "bitsetoperleq.h"
16 #include "operwiedemann.h"
17 #include "permutations.h"
18 #include "powerof2.h"
19 #include "powersetbin.h"
20 #include "vectoroperinsert.h"
21
22 namespace Dedekind
23 {
24     enum
25     {
26         BIGINTTAG
27     };
28
29     template <size_t Number, size_t Power>
30     std::vector<std::vector<std::bitset<Power>>> generateRn(
31             std::vector<std::bitset<Power>> const &dn)
32     {
33         auto permutations = Internal::permutations<Number, Power>();
34
35         std::vector<std::vector<std::bitset<Power>>> rn;
36         std::set<std::bitset<Power>, BitSetLess> processed;
37         for (auto iter = dn.begin(); iter != dn.end(); ++iter)
38         {
39             if (processed.find(*iter) == processed.end())
40             {
41                 auto equivs = Internal::equivalences(*iter, permutations);
42
43                 std::vector<std::bitset<Power>> permuted;
44                 copy(equivs.begin(), equivs.end(), std::back_inserter(
                        permuted));
45                 for (auto perm = equivs.begin(); perm != equivs.end(); ++perm
                        )
46                 {
47                     processed.insert(*perm);
48                 }
49
50                 rn.push_back(permuted);
51             }
52         }
53
54         return rn;
55     }
56
57     template <size_t size>
58     std::vector<size_t> calculateGammas(std::vector<std::bitset<size>> const
          elem,size_t n, std::vector<size_t> indexes){
59         size_t length = n/2 + 1;
60         std::vector<size_t>  result(length,0);
61         for(auto iter = elem.begin(); iter != elem.end(); ++iter)
62         {
63             bool valid = true;
```

```
64              size_t counter = 0;
65              size_t secondcounter = 0;
66              for (auto it = indexes.begin(); it != indexes.end(); ++it)
67              {
68                  if ((*iter).test(*it))
69                  {
70                      if (counter == secondcounter) counter++;
71                      else valid = false;
72                  }
73                  secondcounter++;
74              }
75              if( valid == true && counter <= length-1 )
76              {
77                  result[counter]++;
78              }
79          }
80          return result;
81      }
82
83      std::vector<size_t> generateIndexes(size_t n = 2)
84      {
85          std::vector<size_t> indexes(n,0);
86          for(size_t x = 1; x<n;x++)
87          {
88              indexes[x]=0;
89          }
90          for(size_t x = 0; x<(1<<n)-1;x++)
91          {
92              for(size_t i = 0;i<n;i++)
93              {
94                  if(x & (1<<i))
95                  {
96                      if( x == ((1<<n)-2)) indexes[i]+=1;
97                      else indexes[i] += (1<<x);
98                  }
99              }
100         }
101         return indexes;
102     }
103
104     template <size_t size>
105     UInt128* enumerate(std::vector<std::bitset<size> > const &dn,
106             std::vector<std::vector<std::bitset<size> > > const &rn,
107             size_t rank = 0, size_t nprocs = 1, size_t n= 2 )
108     {
109         std::map<std::bitset<size>, std::bitset<size>, BitSetLess> duals;
110         std::map<std::bitset<size>, size_t, BitSetLess> etas;
111         size_t length = n/2 + 1;
112         // Preprocess gamma's, duals and eta's of all elements
113         std::vector<size_t> indexes = generateIndexes(n);
114         for (auto iter = rn.begin(); iter != rn.end(); ++iter)
115         {
116
117             auto elem = (*iter).begin();
118             size_t tmp = Internal::eta(*elem, dn);
119             for (; elem != (*iter).end(); ++elem)
120             {
121                 etas[*elem] = tmp;
122                 duals[*elem] = Internal::dual(*elem);
123             }
124         }
125
```

```
126
127         UInt128* result;
128         result = new UInt128[length];
129         for (size_t idx = rank; idx < rn.size(); idx += nprocs)
130         {
131             auto iter(rn[idx].begin());
132             std::vector<size_t> gamma =  calculateGammas(rn[idx],n,indexes);
133             for (auto iter2 = dn.begin(); iter2 != dn.end(); ++iter2)
134             {
135                 auto first = *iter & *iter2;
136                 auto second = duals[*iter] & duals[*iter2];
137
138                 for(size_t i=0;i<length;i++)
139                 {
140                 result[i] += gamma[i] * etas[first] * etas[second];
141                 }
142             }
143         }
144
145         return result;
146     }
147
148     template <size_t size>
149     std::vector<std::bitset<(size << 1)> > generate(
150             std::vector<std::bitset<size> > const &dn)
151     {
152         std::vector<std::bitset<(size << 1)> > dn1;
153
154         for (auto iter = dn.begin(); iter != dn.end(); ++iter)
155         {
156             for (auto iter2 = dn.begin(); iter2 != dn.end(); ++iter2)
157             {
158                 if (*iter <= *iter2)
159                 {
160                     dn1.push_back(Internal::concatenate(*iter, *iter2));
161                 }
162             }
163         }
164
165         return dn1;
166     }
167
168     namespace Internal
169     {
170         template <size_t Number>
171         struct MonotoneSubsets
172         {
173             static std::vector<std::bitset<PowerOf2<Number>::value>> result;
174         };
175
176         template <size_t Number>
177         std::vector<std::bitset<PowerOf2<Number>::value>>
178                 MonotoneSubsets<Number>::result(generate(
179                         MonotoneSubsets<(Number - 1)>::result));
180
181         template <>
182         struct MonotoneSubsets<0>
183         {
184             static std::vector<std::bitset<1>> result;
185         };
186
```

```
187         std::vector<std::bitset<1>> MonotoneSubsets<0>::result({std::bitset
                <1>(0),
188                 std::bitset<1>(1)});
189     }
190 
191     template <size_t Number>
192     UInt128* monotoneSubsets(size_t rank = 0, size_t size = 1)
193     {
194         double start = MPI::Wtime();
195         auto dn = Internal::MonotoneSubsets<(1<<Number) - 2>::result;
196 
197         std::cerr << "Rank␣" << rank << "␣is␣done␣generating␣D"
198                 << (1<<Number) - 2 << ":␣" << dn.size() <<  '\n';
199 
200         auto rn = generateRn<(1<<Number) - 2>(dn);
201 
202         std::cerr << "Rank␣" << rank << "␣is␣done␣generating␣R"
203                 << (1<<Number) - 2 << ":␣" << rn.size() << '\n';
204         double end = MPI::Wtime();
205         std::cout << "Generated␣D␣and␣R" <<(1<<Number)-2 << "␣in␣" << end-
                start<< "s\n";
206             UInt128* result = enumerate(dn, rn, rank, size,Number);
207 
208             return result;
209     }
210 }
211 
212 
213 #endif // end of guard DEDEKIND_H_
```

Method 2

Listing 21: project-alt/main.ih

```
1 
2 #include <iostream>
3 #include <sstream>
4 #include <mpi.h>
5 #include <sys/time.h>
6 
7 #include "dedekind/dedekind.h"
8 #include "uint128/uint128.h"
9 
10 using namespace std;
11 
12 
13 struct timeval tim2;
14 
15 
16 typedef Dedekind::UInt128* (*fptr)(size_t, size_t);
17 
18 template <size_t a = 3>
19 fptr findFunction(size_t b)
20 {
21     if (a == b)
22     {
23         return Dedekind::monotoneSubsets<a>;
24     }
25     else
26     {
27         return findFunction<a - 1>(b);
28     }
```

```
}

template <>
fptr findFunction<1>(size_t b)
{
    return Dedekind::monotoneSubsets<1>;
}
```

Listing 22: project-alt/main.cpp

```

#include "main.ih"

int main(int argc, char **argv)
{
    gettimeofday(&tim2, NULL);

    MPI::Init(argc, argv);
    MPI::COMM_WORLD.Set_errhandler(MPI::ERRORS_THROW_EXCEPTIONS);

    size_t rank = 0;
    size_t size = 1;
    try
    {
        rank = MPI::COMM_WORLD.Get_rank();
        size = MPI::COMM_WORLD.Get_size();
    }
    catch (MPI::Exception const &exception)
    {
        cerr << "MPI␣error:␣" << exception.Get_error_code() << "␣-␣"
                << exception.Get_error_string() << endl;
    }


    if (argc == 3 && string(argv[1]) == "-d")
    {
        size_t n = 2;
        stringstream ss(argv[2]);
        ss >> n;

        double start = MPI::Wtime();

        // findFunction makes it very slow, this can be solved by replacing
            it
        // with the following, replacing N for the Dedekind number to compute
            .
        //Dedekind::UInt128* result = Dedekind::monotoneSubsets<2>(rank, size
            );
        Dedekind::UInt128* result = findFunction(n)(rank, size);

        double end = MPI::Wtime();
        cerr << "Rank␣" << rank << "␣done!" /* Result: " << result*/ << "␣in␣
            "
             << end - start << "s\n";
        size_t length = n+1;
        // reduce over all processes
        if (rank == 0)
        {
            size_t toReceive = (size-1)*length+1;
            while (--toReceive)
            {
                // send the high and the low part of the result
```

```
                uint_fast64_t lohi[3];
                for( size_t i=0;i<3;i++){
                    lohi[i]=0;
                }
                MPI::Status status;
                MPI::COMM_WORLD.Recv(lohi, 3, MPI::UNSIGNED_LONG,
                        MPI::ANY_SOURCE, Dedekind::BIGINTTAG, status);

                Dedekind::UInt128 tmp(lohi[0], lohi[1]);
                result[lohi[2]] += tmp;
            }

            double final = MPI::Wtime();
            Dedekind::UInt128 dedekindnumber=0;
            Dedekind::UInt128 othernumber =0;
            for(size_t i=0;i<=n;i++)
            {
                cout << result[i] << "\n";
                dedekindnumber += result[i];
                othernumber += result[i] *(1<<(n-i));
            }
            othernumber -= (1<<n);
            cout << "d" << (1<<n) << "␣=␣" << dedekindnumber << "␣and␣zeta"
                 << n << "␣=␣" << othernumber
                << "␣(in␣" << final - start << "s)\n";
        }
        else
        {
            // send the high and the low part of the result
            uint_fast64_t lohi[3];
            for(size_t i=0;i<length;i++){
                lohi[0] = result[i].lo();
                lohi[1] = result[i].hi();
                lohi[2] = i;
                MPI::COMM_WORLD.Bsend(&lohi, 3, MPI::UNSIGNED_LONG, 0,
                        Dedekind::BIGINTTAG);
            }

            double final = MPI::Wtime();
            cerr << "Rank␣" << rank << "␣exiting!␣Total:␣"
                    << final - start << "s\n";
        }
    }
    else
    {
        cout << "Usage:␣mpirun␣-np␣N␣./project␣-d␣X␣\n"
            << "Where␣X␣in␣[2..n)␣is␣the␣Dedekind␣Number␣to␣calculate.\n"
            << "And␣N␣is␣the␣number␣of␣processes␣you␣would␣like␣to␣use.\n\n"
            << "Note:␣The␣program␣will␣also␣work␣when␣running␣normally␣"
            << "(without␣using␣mpirun).\n"
            << "In␣that␣case␣the␣program␣will␣just␣run␣on␣1␣core.\n";
    }
    MPI::Finalize();
}
```

Listing 23: project-alt/dedekind/dedekind.h

```

#ifndef DEDEKIND_H_
#define DEDEKIND_H_

#include <algorithm>
```

```
6  #include <bitset>
7  #include <iostream>
8  #include <map>
9  #include <set>
10 #include <vector>
11
12 #include "../uint128/uint128.h"
13
14 #include "bitsetless.h"
15 #include "bitsetoperleq.h"
16 #include "operwiedemann.h"
17 #include "permutations.h"
18 #include "powerof2.h"
19 #include "powersetbin.h"
20 #include "vectoroperinsert.h"
21
22 namespace Dedekind
23 {
24     enum
25     {
26         BIGINTTAG
27     };
28
29     template <size_t Number, size_t Power>
30     std::vector<std::vector<std::bitset<Power>>> generateRn(
31             std::vector<std::bitset<Power>> const &dn)
32     {
33         auto permutations = Internal::permutations<Number, Power>();
34
35         std::vector<std::vector<std::bitset<Power>>> rn;
36         std::set<std::bitset<Power>, BitSetLess> processed;
37         for (auto iter = dn.begin(); iter != dn.end(); ++iter)
38         {
39             if (processed.find(*iter) == processed.end())
40             {
41                 auto equivs = Internal::equivalences(*iter, permutations);
42
43                 std::vector<std::bitset<Power>> permuted;
44                 copy(equivs.begin(), equivs.end(), std::back_inserter(
                        permuted));
45                 for (auto perm = equivs.begin(); perm != equivs.end(); ++perm
                        )
46                 {
47                     processed.insert(*perm);
48                 }
49
50                 rn.push_back(permuted);
51             }
52         }
53
54         return rn;
55     }
56
57     template <size_t size>
58     std::vector<size_t> calculateGammas(std::vector<std::bitset<size>> const
         elem,size_t n, std::vector<size_t> indexes){
59         size_t length = n + 1;
60         std::vector<size_t>  result(length,0);
61         for(auto iter = elem.begin(); iter != elem.end(); ++iter)
62         {
63             size_t counter =0;
64             for (size_t i = 0; i < n; i++)
```

```
65              {
66                  if ((*iter).test(indexes[i]))
67                  {
68                      counter++;
69                  }
70              }
71              result[counter]++;
72          }
73          return result;
74      }
75
76      //template <size_t size>
77      std::vector<size_t> generateIndexes(size_t n = 2)
78      {
79          std::vector<size_t> indexes(n,0);
80          for(size_t x = 1; x<n;x++)
81          {
82              indexes[x]=0;
83          }
84          for(size_t x = 0; x<(1<<n)-1;x++)
85          {
86              for(size_t i = 0;i<n;i++)
87              {
88                  if(x & (1<<i))
89                  {
90                      if( x == ((1<<n)-2)) indexes[i]+=1;
91                      else indexes[i] += (1<<x);
92                  }
93              }
94          }
95          return indexes;
96      }
97
98      template <size_t size>
99      UInt128* enumerate(std::vector<std::bitset<size> > const &dn,
100             std::vector<std::vector<std::bitset<size> > > const &rn,
101             size_t rank = 0, size_t nprocs = 1, size_t n= 2 )
102     {
103         std::map<std::bitset<size>, std::bitset<size>, BitSetLess> duals;
104         std::map<std::bitset<size>, size_t, BitSetLess> etas;
105         size_t length = n + 1;
106         // Preprocess gamma's, duals and eta's of all elements
107         std::vector<size_t> indexes = generateIndexes(n);
108         for (auto iter = rn.begin(); iter != rn.end(); ++iter)
109         {
110
111             auto elem = (*iter).begin();
112             size_t tmp = Internal::eta(*elem, dn);
113             for (; elem != (*iter).end(); ++elem)
114             {
115                 etas[*elem] = tmp;
116                 duals[*elem] = Internal::dual(*elem);
117             }
118         }
119
120
121         UInt128* result;
122         result = new UInt128[length];
123         for (size_t idx = rank; idx < rn.size(); idx += nprocs)
124         {
125             auto iter(rn[idx].begin());
126             std::vector<size_t> gamma =  calculateGammas(rn[idx],n,indexes);
```

```
127             for (auto iter2 = dn.begin(); iter2 != dn.end(); ++iter2)
128             {
129                 auto first = *iter & *iter2;
130                 auto second = duals[*iter] & duals[*iter2];
131 
132                 for(size_t i=0;i<length;i++)
133                 {
134                 result[i] += gamma[i] * etas[first] * etas[second];
135                 }
136             }
137         }
138         return result;
139     }
140 
141     template <size_t size>
142     std::vector<std::bitset<(size << 1)> > generate(
143             std::vector<std::bitset<size> > const &dn)
144     {
145         std::vector<std::bitset<(size << 1)> > dn1;
146 
147         for (auto iter = dn.begin(); iter != dn.end(); ++iter)
148         {
149             for (auto iter2 = dn.begin(); iter2 != dn.end(); ++iter2)
150             {
151                 if (*iter <= *iter2)
152                 {
153                     dn1.push_back(Internal::concatenate(*iter, *iter2));
154                 }
155             }
156         }
157 
158         return dn1;
159     }
160 
161     namespace Internal
162     {
163         template <size_t Number>
164         struct MonotoneSubsets
165         {
166             static std::vector<std::bitset<PowerOf2<Number>::value>> result;
167         };
168 
169         template <size_t Number>
170         std::vector<std::bitset<PowerOf2<Number>::value>>
171                 MonotoneSubsets<Number>::result(generate(
172                         MonotoneSubsets<(Number - 1)>::result));
173 
174         template <>
175         struct MonotoneSubsets<0>
176         {
177             static std::vector<std::bitset<1>> result;
178         };
179 
180         std::vector<std::bitset<1>> MonotoneSubsets<0>::result({std::bitset
                <1>(0),
181                 std::bitset<1>(1)});
182     }
183 
184     template <size_t Number>
185     UInt128* monotoneSubsets(size_t rank = 0, size_t size = 1)
186     {
187         double start = MPI::Wtime();
```

```
188         auto dn = Internal::MonotoneSubsets<(1<<Number) - 2>::result;
189
190         std::cerr << "Rank " << rank << " is done generating D"
191                 << (1<<Number) - 2 << ": " << dn.size() <<  '\n';
192
193         auto rn = generateRn<(1<<Number) - 2>(dn);
194
195         std::cerr << "Rank " << rank << " is done generating R"
196                 << (1<<Number) - 2 << ": " << rn.size() << '\n';
197         double end = MPI::Wtime();
198         std::cout << "Generated D and R" <<(1<<Number)-2 << " in " << end-
                start<< "s\n";
199             UInt128* result = enumerate(dn, rn, rank, size,Number);
200
201             return result;
202     }
203 }
204
205
206 #endif // end of guard DEDEKIND_H_
```

Bitset Operations

These files are the same for both methods

Listing 24: project/dedekind/bitsetless.h

```
1  #ifndef BITSETLESS_H_
2  #define BITSETLESS_H_
3
4  #include <bitset>
5  #include <cstddef>
6
7
8  namespace Dedekind
9  {
10
11
12 class BitSetLess
13 {
14     public:
15         template<size_t size>
16         bool operator()(std::bitset<size> const &lhs,
17                 std::bitset<size> const &rhs) const
18         {
19             return lhs.to_ulong() > rhs.to_ulong();
20         }
21 };
22
23 template <size_t size>
24 bool bitsetLess(std::bitset<size> const &lhs, std::bitset<size> const &rhs)
25 {
26     return lhs.to_ulong() > rhs.to_ulong();
27 }
28
29 } // namespace Dedekind
30
31
32 #endif
```

Listing 25: project/dedekind/bitsetoperleq.h

```
1
```

```
#ifndef BITSETOPERLEQ_H_
#define BITSETOPERLEQ_H_

#include <bitset>
#include <cstddef>


namespace Dedekind
{


template <size_t size>
bool operator<=(std::bitset<size> lhs, std::bitset<size> const &rhs)
{
    return (lhs.flip() | rhs).all();
}


} // namespace Dedekind


#endif
```

Listing 26: project/dedekind/operwiedemann.h

```

#ifndef OPERWIEDEMANN_H_
#define OPERWIEDEMANN_H_

#include <bitset>
#include <vector>
#include <string>

#include "bitsetoperleq.h"


namespace Dedekind
{

namespace Internal
{


template <size_t size>
std::bitset<size> reverse(std::bitset<size> const &bset)
{
    std::bitset<size> reverse;
    for (size_t iter = 0; iter != size; ++iter)
    {
        reverse[iter] = bset[size - iter - 1];
    }
    return reverse;
}

template <size_t size>
std::bitset<size> dual(std::bitset<size> const &bset)
{
    return reverse(bset).flip();
}

template <size_t size>
size_t eta(std::bitset<size> const &bset,
```

```
38                 std::vector<std::bitset<size>> const &dn)
39 {
40     size_t result = 0;
41     for (size_t idx = 0; idx < dn.size(); ++idx)
42     {
43         if (dn[idx] <= bset)
44         {
45             ++result;
46         }
47     }
48     return result;
49 }
50 
51 template <size_t size>
52 std::bitset<(size << 1)> concatenate(std::bitset<size> const &lhs,
53         std::bitset<size> const &rhs)
54 {
55     std::string lhs_str = lhs.to_string();
56     std::string rhs_str = rhs.to_string();
57 
58     return std::bitset<(size << 1)>(lhs_str + rhs_str);
59 }
60 
61 
62 } // namespace Internal
63 
64 } // namespace Dedekind
65 
66 
67 #endif
```

Listing 27: project/dedekind/permutations.h

```
1  
2  #ifndef PERMUTATIONS_H_
3  #define PERMUTATIONS_H_
4  
5  #include <algorithm>
6  #include <array>
7  #include <bitset>
8  #include <set>
9  #include <vector>
10 
11 #include "powersetbin.h"
12 
13 
14 namespace Dedekind
15 {
16 
17 namespace Internal
18 {
19 
20 
21 template <size_t size>
22 std::bitset<size> permute(std::array<size_t, size> const &permutation,
23         std::bitset<size> const &elem)
24 {
25     std::bitset<size> result;
26     for (size_t idx = 0; idx != result.size(); ++idx)
27     {
28         result[idx] = elem[permutation[idx]];
29     }
```

```
30      return result;
31  }
32
33  template <size_t Number, size_t Power>
34  std::array<size_t, Power> subsetPermutation(
35          std::array<size_t, Number> const &permutation,
36          std::vector<std::bitset<Number>> const &pset)
37  {
38      std::array<size_t, Power> result;
39      size_t idx = 0;
40      for (auto iter = pset.begin(); iter != pset.end(); ++iter)
41      {
42          std::bitset<Number> tmp = permute(permutation, *iter);
43          result[idx++] = find(pset.begin(), pset.end(), tmp) - pset.begin();
44      }
45      return result;
46  }
47
48  template <size_t Number, size_t Power>
49  std::vector<std::array<size_t, Power>> permutations()
50  {
51      std::vector<std::bitset<Number>> powerset =
52              Internal::PowerSet<Number>::powerSetBin();
53
54      std::array<size_t, Number> permutation;
55      for (size_t idx = 0; idx != Number; ++idx)
56      {
57          permutation[idx] = idx;
58      }
59
60      std::vector<std::array<size_t, Power>> result;
61      do
62      {
63          result.push_back(subsetPermutation<Number, Power>(permutation,
                powerset));
64      }
65      while (std::next_permutation(permutation.begin(), permutation.end()));
66
67      return result;
68  }
69
70  template <size_t size>
71  std::set<std::bitset<size>, BitSetLess> equivalences(
72          std::bitset<size> const &bset,
73          std::vector<std::array<size_t, size>> const &perms)
74  {
75      std::set<std::bitset<size>, BitSetLess> result;
76      for (auto iter = perms.begin(); iter!= perms.end(); ++iter)
77      {
78          std::bitset<size> temp = permute(*iter, bset);
79          result.insert(temp);
80      }
81      return result;
82  }
83
84
85  } // namespace Internal
86
87  } // namespace Dedekind
88
89
90  #endif
```

Listing 28: project/dedekind/powerof2.h

```
1
2  #ifndef POWEROF2_H_
3  #define POWEROF2_H_
4
5  #include <cstddef>
6
7  namespace Dedekind
8  {
9
10 namespace Internal
11 {
12
13
14 template<size_t Number>
15 struct PowerOf2
16 {
17     static size_t const value;
18 };
19
20 template<size_t Number>
21 size_t const PowerOf2<Number>::value((PowerOf2<Number - 1>::value << 1));
22
23
24 template<>
25 struct PowerOf2<0>
26 {
27     static size_t const value;
28 };
29
30 size_t const PowerOf2<0>::value(1);
31
32
33
34 template<size_t Number>
35 struct LogOf2
36 {
37     static size_t const value;
38 };
39
40 template<size_t Number>
41 size_t const LogOf2<Number>::value(LogOf2<(Number >> 1)>::value + 1);
42
43
44 template<>
45 struct LogOf2<1>
46 {
47     static size_t const value;
48 };
49
50 size_t const LogOf2<1>::value(0);
51
52
53 } // namespace Internal
54
55 } // namespace Dedekind
56
57
58 #endif
```

Listing 29: project/dedekind/powersetbin.h

```
1
2  #ifndef POWERSETBIN_H_
3  #define POWERSETBIN_H_
4
5  #include <vector>
6  #include <bitset>
7
8  namespace Dedekind
9  {
10
11 namespace Internal
12 {
13
14
15 template <size_t size>
16 struct PowerSet
17 {
18     static std::vector<std::bitset<size>> powerSetBin();
19 };
20
21 template <size_t size>
22 std::vector<std::bitset<size>> PowerSet<size>::powerSetBin()
23 {
24     auto current = PowerSet<size - 1>::powerSetBin();
25
26     std::vector<std::bitset<size>> result;
27     for (auto iter = current.begin(); iter != current.end(); ++iter)
28     {
29         std::bitset<size> tmp((*iter).to_ulong() + (1 << (size - 1)));
30         result.push_back(tmp);
31     }
32
33     for (auto iter = current.begin(); iter != current.end(); ++iter)
34     {
35         std::bitset<size> tmp((*iter).to_ulong());
36         result.push_back(tmp);
37     }
38
39     return result;
40 }
41
42
43 template <>
44 struct PowerSet<0>
45 {
46     static std::vector<std::bitset<0>> powerSetBin();
47 };
48
49 std::vector<std::bitset<0>> PowerSet<0>::powerSetBin()
50 {
51     return std::vector<std::bitset<0>>({ std::bitset<0>() });
52 }
53
54
55 } // namespace Internal
56
57 } // namespace Dedekind
58
59
60 #endif
```

Listing 30: project/dedekind/vectoroperinsert.h

```
1
2  #ifndef VECTOROPERINSET_H_
3  #define VECTOROPERINSET_H_
4
5  #include <algorithm>
6  #include <array>
7  #include <bitset>
8  #include <iostream>
9  #include <sstream>
10 #include <string>
11 #include <vector>
12
13 #include "bitsetless.h"
14 #include "powerof2.h"
15 #include "powersetbin.h"
16
17 template <typename Type>
18 std::ostream &operator<<(std::ostream &out,
19     std::vector<Type> const &rhs)
20 {
21     out << '{';
22     for (auto iter = rhs.begin();
23             iter != rhs.end();
24                 ++iter)
25     {
26         out << ((iter != rhs.begin()) ? ", " : "") << *iter;
27     }
28     out << '}';
29
30     return out;
31 }
32
33 template <size_t size>
34 std::string bitsetToString(std::bitset<size> const &bset,
35         std::array<std::string, size> const &domain, std::string const &sep)
36 {
37     std::string result;
38     for (size_t idx = 0; idx != bset.size(); ++idx)
39     {
40         if (bset[idx])
41         {
42             std::stringstream ss(domain[idx]);
43             result.append(ss.str() + sep);;
44         }
45     }
46     if (bset.none())
47     {
48         result.append("e");
49     }
50     return result;
51 }
52
53 template <size_t size>
54 std::string powersetToString(std::vector<std::bitset<size>> const &pset)
55 {
56     std::array<std::string, size> domain;
57     for (size_t idx = 0; idx != size; ++idx)
58     {
59         std::stringstream ss;
60         ss << idx;
61         domain[idx] = ss.str();
```

```
    }

    std::string result;
    for (auto iter = pset.begin(); iter != pset.end(); ++iter)
    {
        result.append(bitsetToString((*iter), domain, "") + ",␣");
    }
    return result;
}

template <size_t size>
std::string subsetToString(std::vector<std::bitset<size>> const &sset)
{
    size_t const logof2(Dedekind::Internal::LogOf2<size>::value);
    std::vector<std::bitset<logof2>> powerset =
            Dedekind::Internal::PowerSet<logof2>::powerSetBin();


    std::sort(powerset.begin(), powerset.end(), Dedekind::bitsetLess<logof2>)
        ;

    std::array<std::string, logof2> domain1;
    for (size_t idx = 0; idx != logof2; ++idx)
    {
        std::stringstream ss;
        ss << idx;
        domain1[idx] = ss.str();
    }

    std::cout << domain1.size() << "␣" << size << "␣" << powerset.size() << '
        \n';

    std::array<std::string, size> domain2;
    for (size_t idx = 0; idx != powerset.size(); ++idx)
    {
        domain2[idx] = bitsetToString(powerset[idx], domain1, "");
    }

    std::string result;
    for (auto iter = sset.begin(); iter != sset.end(); ++iter)
    {
        result.append(bitsetToString((*iter), domain2, "␣") + "\n");
    }
    return result;
}

#endif
```

UInt128

Listing 31: project/uint128/uint128.ih

```
#include "uint128.h"

#include <limits>
#include <iostream>
#include <iomanip>
#include <cmath>

```

```
9  using namespace std;
```

Listing 32: project/uint128/uint128.h

```
1
2  #ifndef DEDEKIND_UINT_
3  #define DEDEKIND_UINT_
4
5  #include <cstdint>
6  #include <iosfwd>
7
8  namespace Dedekind
9  {
10     class UInt128
11     {
12         uint_fast64_t d_hi;
13         uint_fast64_t d_lo;
14
15         public:
16             friend std::ostream &operator<<(std::ostream &out,
17                     UInt128 const &uint128);
18
19             UInt128(UInt128 const &other) = default;
20             UInt128(uint_fast64_t lo = 0, uint_fast64_t hi = 0);
21
22             UInt128 &operator+=(uint_fast64_t other);
23             UInt128 &operator-=(uint_fast64_t other);
24             UInt128 &operator*=(uint_fast64_t other);
25             UInt128 &operator+=(UInt128 const &other);
26
27             uint_fast64_t hi() const;
28             uint_fast64_t lo() const;
29     };
30
31
32     inline UInt128 operator+(UInt128 const &lhs, UInt128 const &rhs)
33     {
34         UInt128 tmp(lhs);
35         return tmp += rhs;
36     }
37
38     inline UInt128 operator*(UInt128 const &lhs, uint_fast64_t rhs)
39     {
40         UInt128 tmp(lhs);
41         return tmp *= rhs;
42     }
43
44     inline uint_fast64_t UInt128::hi() const
45     {
46         return d_hi;
47     }
48
49     inline uint_fast64_t UInt128::lo() const
50     {
51         return d_lo;
52     }
53 }
54
55 #endif
```

Listing 33: project/uint128/uint128.cpp

```
1
2  #include "uint128.ih"
3
4  namespace Dedekind
5  {
6      UInt128::UInt128(uint_fast64_t lo, uint_fast64_t hi)
7      :
8          d_hi(hi),
9          d_lo(lo)
10     {
11     }
12 }
```

Listing 34: project/uint128/operatoraddassign1.cpp

```
1
2  #include "uint128.ih"
3
4  namespace Dedekind
5  {
6      UInt128 &UInt128::operator+=(UInt128 const &other)
7      {
8          if (d_lo > std::numeric_limits<unsigned long>::max() - other.d_lo)
9          {
10             ++d_hi;
11         }
12
13         d_lo += other.d_lo;
14         d_hi += other.d_hi;
15         return *this;
16     }
17 }
```

Listing 35: project/uint128/operatoraddassign2.cpp

```
1
2  #include "uint128.ih"
3
4  namespace Dedekind
5  {
6      UInt128 &UInt128::operator+=(uint_fast64_t other)
7      {
8          if (d_lo > std::numeric_limits<unsigned long>::max() - other)
9          {
10             ++d_hi;
11         }
12
13         d_lo += other;
14         return *this;
15     }
16 }
```

Listing 36: project/uint128/operatorinsert.cpp

```
1
2  #include "uint128.ih"
3
4  // http://stackoverflow.com/questions/4361441/c-print-a-biginteger-in-base-10
5
6  namespace Dedekind
```

```
7  {
8      std::ostream &operator<<(std::ostream &out, UInt128 const &uint128)
9      {
10         size_t d[39] = {0}; // a 128 bit number has at most 39 digits
11
12         // starting at the highest, for each bit
13         for (int iter = 63; iter != -1; --iter)
14         {
15             // increase the lowest digit if this bit is set
16             if ((uint128.d_hi >> iter) & 1)
17             {
18                 d[0]++;
19             }
20
21             // multiply by 2, since bits represent powers of 2
22             for (size_t idx = 0; idx < 39; ++idx)
23             {
24                 d[idx] *= 2;
25             }
26
27             // handle carries/overflow
28             for (size_t idx = 0; idx < 38; ++idx)
29             {
30                 d[idx + 1] += d[idx] / 10;
31                 d[idx] %= 10;
32             }
33         }
34
35         for (int iter = 63; iter > -1; --iter)
36         {
37             // increase the lowest digit if this bit is set
38             if ((uint128.d_lo >> iter) & 1)
39             {
40                 d[0]++;
41             }
42
43             // only multiply if more bits will follow
44             if (iter > 0)
45             {
46                 for (size_t idx = 0; idx < 39; ++idx)
47                 {
48                     d[idx] *= 2;
49                 }
50             }
51
52             // handle carries/overflow
53             for (size_t idx = 0; idx < 38; ++idx)
54             {
55                 d[idx + 1] += d[idx] / 10;
56                 d[idx] %= 10;
57             }
58         }
59
60         // find highest digit to be inserted in outputstream
61         int idx;
62         for (idx = 38; idx > 0; --idx)
63         {
64             if (d[idx] > 0)
65             {
66                 break;
67             }
68         }
```

```
69
70          // insert from here
71          for (; idx > -1; --idx)
72          {
73              out << d[idx];
74          }
75
76          return out;
77      }
78  }
```

Listing 37: project/uint128/operatorminus.cpp

```
1
2   #include "uint128.ih"
3
4   namespace Dedekind
5   {
6       UInt128 &UInt128::operator-=(uint_fast64_t other)
7       {
8           if (d_lo < other)
9           {
10              --d_hi;
11          }
12
13          d_lo -= other;
14          return *this;
15      }
16  }
```

Listing 38: project/uint128/operatormultiply.cpp

```
1   #include "uint128.ih"
2
3   namespace Dedekind
4   {
5       UInt128 &UInt128::operator*=(uint_fast64_t other)
6       {
7           UInt128 temp(*this);
8           d_lo =0; d_hi=0;
9           for(uint_fast64_t i=0;i<64;i++){
10              if((other&1)!=0){
11                  *this+= (temp.d_lo<<i);
12                  d_hi+= (temp.d_hi<<i);
13                  if ( i!= 0) //can't bitshift 64 bits
14                      d_hi+= (temp.d_lo>>(64-i));
15              }
16              other>>=1;
17          }
18          return *this;
19      }
20  }
```