University of Groningen

Bachelor of Science, Computing Science

Calculating the 8th Dedekind Number

by

Arjen Teijo Zijlstra

July 11, 2013

Supervisor

prof. dr. G.R. Renardel de Lavalette

2nd reader

dr. A. Meijster

Abstract

Dedekind Numbers ($d_n \mid n = 0, 1, 2, \dots$) are a rapidly growing sequence of integers: 2, 3, 6, 20, 168, 7 581, 7 828 354, 2 414 682 040 998, 56 130 437 228 687 557 907 788. d_n counts the number of monotone subsets of a power set on n elements. A subset is monotone when there are no elements in the power set, that contain an element of the subset, and are not an element of the subset themself. d_8 is the biggest computed Dedekind number so far. It was first computed in 1991 by Doug Wiedemann. This took him 200 hours on a Cray-2.

In this thesis, Wiedemanns strategy is explained, implemented in C/C++ and parallelised using the *Message Passing Interface*. The goal is to gather knowledge about the theory and to check the calculation. Another intention in this thesis is to speedup the calculation as much as possible. The first goal is accomplished and the result of d_n is exactly the same as Wiedemanns result. The shortest time to calculate d_8 is about 30 minutes. Other results are discussed in this thesis. Furthermore, some things are said about scaling this calculation to d_9.

Acknowledgements

I would like to express my gratitude to my supervisor prof. dr. Gerard R. Renardel de Lavalette, for giving me the chance to work on this project and for all his effort and time, explaining the math and working on new ideas with me.

Second, I would like to thank dr. Arnold Meijster, for all the useful comments, remarks and advices on my work. I have learned a lot from you, throughout this project and the rest of my degree.

Furthermore, I would like to thank all my fellow students and friends that helped me in any way. Thank you, Marc, Jorrit, Safet, Tycho, Dirk, Klaas, Tijmen, Johan, Paul, Matthew, Herbert, Robbert-Jan, Joost, Aloys.

Also, I would like to thank my family for giving me the chance to fully concentrate on working on this thesis and studying as a whole. I know for sure that I can always count on you and you will always be there to help me.

Moreover, I would like to thank everyone that was present at the presentation, for being there, paying attention and giving me time. I hope you liked it and that you understood what I worked on during my project.

Finally, I would like to thank everyone else from whom I have learned and everyone that helped me to develop myself. I would not have been this far without any of you.

Thank you all. My studying would not have been the same without you.

Table of Contents

Chapter 1

Introduction

The Dedekind numbers $(d_n \mid n = 0, 1, 2, \dots)$ are a rapidly growing sequence of integers, that count the number of monotone subsets of a power set on n elements. Doug Wiedemann computed d_8 in 1991. This is still the biggest computed Dedekind number so far. Calculating this numer took Wiedemann 200 hours on a Cray-2.

In this thesis, Wiedemanns strategy is explained by analysing the article in which Wiedemann published his findings, step by step. After that his strategy is followed when implementing a program in C/C++ to find d_8, and this program is parallelised using the *Message Passing Interface*. The goal of this, is to gather knowledge about the theory and the strategy Wiedemann used. The value of d_8 that Wiedemann computed is checked, and also looking at possibilities to scale up to d_9 is a goal of this thesis.

The currently computed values for the Dedekind numbers (d_n) are given in table 1.1.

n	d_n	
0	2	Dedekind (1897)
1	3	Dedekind (1897)
2	6	Dedekind (1897)
3	20	Dedekind (1897)
4	168	Dedekind (1897)
5	7581	Church (1940)
6	7828354	Ward (1946)
7	2414682040998	Church (1965)
8	56130437228687557907788	Wiedemann (1991)

Table 1.1: Known values of d_n

The table shows that the time between finding one and the next Dedekind number is about 10 to 20 years, which suggests that these years could be a good time for finding d_9.

1.1 Definitions

Before starting with the theory, some definitions need to be given. Most of the definitions are used as Wiedemann (1991) did. This way, consistency is preserved.

The power set of a set S, $\wp(S)$, is defined as the set of all subsets of S, including the empty set and the set S itself. We define

$$V(n) = \{0, 1, \ldots, n-1\} \tag{1.1}$$

and

$$Q(n) = \wp(V(n)) \tag{1.2}$$
$$\#Q(n) = 2^n \tag{1.3}$$

Note that the size of $Q(n)$ is 2^n. $Q(n)$ is used as a basis during this thesis.

$$\begin{aligned} S \text{ monotone in } Q(n) \equiv{} & S \subseteq Q(n) \wedge \\ & \forall t \in S, \forall u \in Q(n)(t \subseteq u \Rightarrow u \in S) \end{aligned} \tag{1.4}$$

So, a subset S of $Q(n)$ is monotone in $Q(n)$, if for every element u in $Q(n)$, if for any t in S, $t \subseteq u$ then $u \in S$.

For example, take $n = 3$ and take $S = \{01, 012\} \subseteq Q(3)$ as in figure 1.1. Note that, for convenience, sets like $\{0, 1\}$ and $\{0, 1, 2\}$ are written as 01 respectively 012. Now $S \subseteq Q(3)$ and also S is monotone in $Q(3)$, since there are no elements in $Q(3)$ that are above any elements of S but not in S. If, for example, 0 would be added to S, i.e. $S = \{0, 01, 012\}$, S would not be monotone in $Q(3)$, since $02 \in Q(n)$ and $0 \subseteq 02$. So, like $S = \{01, 012\}$, $S = \{0, 01, 02, 012\}$ is monotone in $Q(3)$.

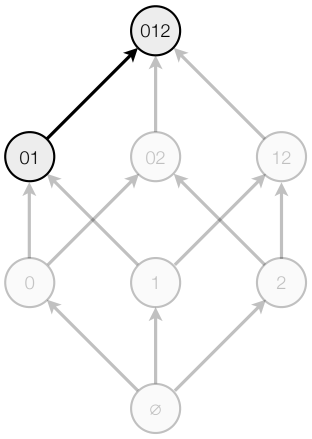

Figure 1.1: $\{01, 012\}$

Let D_n be defined as

$$D_n = \{S \subseteq Q(n) \mid S \text{ monotone in } Q(n)\} \tag{1.5}$$

$$d_n = \#D_n \tag{1.6}$$

So, D_n is the set of all monotone subsets of $Q(n)$ and d_n is the cardinality of D_n. d_n is the n^{th} number in the sequence of Dedekind Numbers. Known values of d_n are given in table 1.1.

A set $S \in Q(n)$ is called *equivalent* to a set $T \in Q(n)$ if a permutation φ of the set $V(n)$, exists such that $\varphi(S) = T$, we write $S \sim T$. Here $\varphi(S)$ means

$$\varphi(S) = \{\{\varphi(x) \mid x \in s\} \mid s \in S\} \tag{1.7}$$

Let R_n contain the least representative of each equivalence class in D_n, with respect to the lexicographical ordening.

For example, take $n = 3$, $S = \{0, 01, 02, 012\}$ and $T = \{2, 02, 12, 012\}$. Observe that, both S and T are monotone in $Q(3)$. Also take (in one-line notation) permutation $\varphi = \begin{pmatrix} 2 & 1 & 0 \end{pmatrix}$. Now $\varphi(S) = \{2, 02, 12, 012\} = T$.

1.2 Examples

To clarify these definitions, an example for $n = 3$ is given. For $n = 3$, $Q(n)$ is equal to

$$\{\varnothing, 0, 1, 01, 2, 02, 12, 012\}$$

Now, the elements of D_3 are given in table 1.2. Here ε means $S = \varnothing \in D_3$. Furthermore, the elements of R_3 are given in the first row of table 1.3. The

∅, 0, 1, 01, 2, 02, 12, 012	0, 01, 02, 12, 012	01, 02, 012
0, 1, 01, 2, 02, 12, 012	2, 02, 12, 012	12, 012
1, 01, 2, 02, 12, 012	01, 02, 12, 012	02, 012
0, 01, 2, 02, 12, 012	1, 01, 12, 012	01, 012
0, 1, 01, 02, 12, 012	0, 01, 02, 012	012
01, 2, 02, 12, 012	02, 12, 012	ε
1, 01, 02, 12, 012	01, 12, 012	

Table 1.2: Sets in D_3

other rows are the equivalents of the element in the first row. This example

∅, 0, 1, 01, 2, 02, 12, 012		
0, 1, 01, 2, 02, 12, 012		
0, 1, 01, 02, 12, 012	0, 01, 2, 02, 12, 012	1, 01, 2, 02, 12, 012
0, 01, 02, 12, 012	1, 01, 02, 12, 012	01, 2, 02, 12, 012
01, 02, 12, 012		
0, 01, 02, 012	1, 01, 12, 012	2, 02, 12, 012
01, 02, 012	01, 12, 012	02, 12, 012
01, 012	02, 012	12, 012
012		
ε		

Table 1.3: Equivalent sets in D_3

is used in the other chapters of this thesis, to improve the understanding of the theory.

Chapter 2

Theory on Monotone Subsets

In this chapter, the algorithm that Wiedemann (1991) used to compute d_8 is discussed, together with the algorithm to generate D_n, described by Fidytek et al. (2001).

2.1 Algorithms

While describing the algorithms, we use an operation to add or delete an element in all elements inside a set S. For this, define,

$$S \oplus n = \{t \cup \{n\} \mid t \in S\} \tag{2.1}$$

$$S \ominus n = \{t \mid t \cup \{n\} \in S\} \tag{2.2}$$

Observe that these operations preserve monoticity from $Q(n)$ to $Q(n+1)$ and vice-versa.

2.1.1 Generating $Q(n+1)$ from $Q(n)$

It is easy to see that when $Q(n)$ is available, $Q(n+1)$ can be produced by taking all elements of $Q(n)$ and also add n to all elements. Also note, $Q(n) \subseteq Q(n+1)$ and $\#Q(n+1) = 2^{n+1} = 2 \cdot 2^n = 2 \cdot \#Q(n)$. See algorithm 1.

Algorithm 1: Generate $Q(n+1)$ from $Q(n)$

```
Input: $Q(n)$, powerset on $V(n)$
Output: $Q(n+1)$, powerset on $V(n+1)$
$V \leftarrow Q(n)$;
foreach $S \in Q(n)$ do
    $V \leftarrow V \cup (S \oplus n)$;
return $V$;
```

2.1.2 Generating D_{n+1} from D_n

Before starting with the algorithm Wiedemann used to calculate d_8, first the algorithm described to generate D_{n+1} from D_n is shown. This is done because when computing d_8, D_6 is used as a basis. This algorithm makes also use of the way the power set $Q(n)$ is built up.

First, observe that an element S in D_{n+1} can be split in two parts, $S \cap Q(n)$ and $S \setminus Q(n)$. The elements in $S \cap Q(n)$ are the elements in S that do not contain n, while the elements in $S \setminus Q(n)$ are the elements of S that do contain n.

$$S = (S \cap Q(n)) \cup (S \setminus Q(n)) \tag{2.3}$$

$$(S \cap Q(n)) \cap (S \setminus Q(n)) = \varnothing \tag{2.4}$$

We observe that $S \cap Q(n)$ is monotone in $Q(n)$, and also $(S \setminus Q(n)) \ominus n$ is monotone in $Q(n)$. As a consequence, the elements of D_{n+1} can be obtained by taking all possible combinations for $S \cap Q(n)$ and $(S \setminus Q(n)) \ominus n$, and combining them to one element. Furthermore, note that $S \cap Q(n) \subseteq (S \setminus Q(n)) \ominus n$, since otherwise S would not have been monotone in $Q(n+1)$. This is the only constraint when picking two elements. Take two elements from D_n, S and T. Both S and T are monotone in $Q(n)$. If $S \subseteq T$, all constraints are satisfied. So $S \cup (T \oplus n)$ is monotone in $Q(n+1)$, and thus in D_{n+1}. For the complete algorithm, see algorithm 2.

For a formal proof, see Fidytek et al. (2001) or Yusun (2011).

Algorithm 2: Generate D_{n+1} from D_n

```
Input: $D_n$, containing all monotone subsets in $Q(n)$
Output: $D_{n+1}$, containing all monotone subsets in $Q(n+1)$.
$V \leftarrow \varnothing$;
foreach $S \in D_n$ do
    foreach $T \in D_n$ do
        if $S \subseteq T$ then
            $V \leftarrow V \cup (S \cup (T \oplus n))$;
return $V$;
```

2.1.3 Computing d_{n+2} from D_n

To be able to count d_n for n up to 8, the method described in Wiedemann (1991) can be used. This method only computes the number of elements and will not give the set itself. It will start from D_n, to compute d_{n+2}.

Analogous to splitting up elements in two parts of $Q(n+1)$ for finding D_{n+1}, to find d_{n+2}, elements in $Q(n+2)$ are split into four parts. For every $S \subseteq Q(n+2)$, these parts are defined as follows.

$$\cdot_{00}, \cdot_{01}, \cdot_{10}, \cdot_{11} : D_{n+2} \to D_n$$

$$S_{00} = Q(n) \cap S \tag{2.5}$$

$$S_{10} = Q(n) \cap (S \ominus n) \tag{2.6}$$

$$S_{01} = Q(n) \cap (S \ominus (n+1))\} \tag{2.7}$$

$$S_{11} = Q(n) \cap ((S \ominus n) \ominus (n+1))\} \tag{2.8}$$

So these are the parts containing n and/or $n+1$, or neither of them. Since S is monotone, $S_{00} \subseteq S_{01}$, $S_{00} \subseteq S_{10}$, $S_{01} \subseteq S_{11}$, $S_{10} \subseteq S_{11}$ and also S_{00}, S_{01}, S_{10} and S_{11} are monotone. Furthermore, S can be obtained by re-adding the omitted elements to each set inside S_{ij}.

Now D_{n+2} could be obtained, by taking all possibilities for S_{00}, S_{01}, S_{10} and S_{11}, and constructing S as follows

$$S = S_{00} \cup (S_{10} \oplus n) \cup (S_{01} \oplus (n+1)) \cup ((S_{11} \oplus n) \oplus (n+1)) \tag{2.9}$$

This will work, but since we are only interested in the cardinality of D_{n+2} there is a more efficient way. To compute the value for d_{n+2}, all possibilities for S_{01} and S_{10} are walked through. Then for every combination of S_{01} and S_{10}, the number of possibilities for S_{00} and S_{11} are computed and multiplied. This clearly will give all possibilities for S, for given S_{01} and S_{10}. Thus, to compute d_{n+2} we loop over D_n and perform this calculation for all possible combinations of S_{01} and S_{10}.

The next thing to discuss, is the way on how to compute the number of possible choices for S_{00} and S_{11}, given S_{01} and S_{10}. To do this, some extra operations are introduced.

First, let the dual of a set $S \subseteq Q(n)$ be S^* defined as

$$\begin{aligned} \cdot^* : \wp\left(Q(n)\right) &\to \wp\left(Q(n)\right) \\ S^* &= \{t^c : t \in S\}^c \\ &= \{V(n) - t \mid t \in S\}^c \\ &= Q(n) - \{V(n) - t \mid t \in S\} \end{aligned} \tag{2.10}$$

Where $\cdot^c$ is the complement of a set. Observe that for the dual S^* of a set $S \subseteq Q(n)$, the following properties hold

$$S \in D_n \Leftrightarrow S^* \in D_n \tag{2.11}$$

$$(S \cup T)^* = S^* \cap T^* \tag{2.12}$$

$$S \subseteq T \Leftrightarrow T^* \subseteq S^* \tag{2.13}$$

Also let the η-value of a set T be defined as

$$\begin{aligned} \eta : D_n &\to \mathbb{N} \\ \eta(T) &= \#\{S \in D_n \mid S \subseteq T\} \\ &= \#\left(D_n \cap \{S \mid S \subseteq T\}\right) \\ &= \#\left(D_n \cap \wp(T)\right) \end{aligned} \tag{2.14}$$

This η-value of a set T is the number of monotone subsets from D_n contained in T. So this η-value, can directly be used to compute the number of possibilities for S_{00} for a given S_{01} and S_{10}. Computing $\eta(S_{01} \cap S_{10})$ will give the number of possibilities for S_{00}.

To compute the number of possibilities for S_{11}, we use the properties of the dual. The number we are looking for is the number of monotone subsets from D_n containing in $S_{01} \cup S_{10}$.

$$\begin{aligned}
&\#\{S \in D_n \mid (S_{01} \cup S_{10}) \subseteq S\} \\
=\ &\text{by (2.12)} \\
&\#\{S^* \in D_n \mid S^* \subseteq (S_{01} \cup S_{10})^*\} \\
=\ &\text{by (2.11) and (2.13)} \\
&\#\{S \in D_n \mid S \subseteq (S^*_{01} \cap S^*_{10})\} \\
=\ &\text{by (2.14)} \\
&\eta(S^*_{01} \cap S^*_{10}) \qquad (2.15)
\end{aligned}$$

An important thing to notice in order to understand this consequence is that only the number of possibilities for S_{11} is relevant, and not the actual possibilities itself.

Taking this together, this results in the following summation.

$$d_{n+2} = \sum_{S_{01} \in D_n} \sum_{S_{10} \in D_n} \eta(S_{01} \cap S_{10}) \cdot \eta(S^*_{01} \cap S^*_{10}) \qquad (2.16)$$

This is put together in algorithm 3.

Algorithm 3: Compute d_{n+2} from D_n

```
Input: $D_n$ containing all monotone subsets in $Q(n)$
Output: $d_{n+2}$, cardinality of $D_{n+2}$
$result \leftarrow 0$;
foreach $S \in D_n$ do
    foreach $T \in D_n$ do
        $result := result + \eta(S \cap T) \cdot \eta(S^* \cap T^*)$;
return $result$;
```

2.1.4 Computing d_{n+2} from D_n and R_n

The algorithms described so far are not very efficient when computing d_n. For this, Wiedemann (1991) noticed that there are a lot of symmetries in D_n. These symmetries are used when constructing R_n. R_n contains the least representative of all equivalent classes in D_n, with respect to the lexicographical ordening. Let $\alpha_T(K) = T$ for any $K \in D_n$. So α_T is a permutation that permutes K to T. Then, define $p(k)$ as follows,

$$\begin{aligned} p(K) = \{\alpha_T \mid T \in K_{/\sim} \wedge \\ \alpha_T \text{ is lexicographically smallest s.t. } \alpha_T(K) = T\} \end{aligned} \tag{2.17}$$

Also, let $\textsc{perms}_n$ be the set of all permutations on $V(n)$.

Now, start with equation (2.16).

$$\begin{aligned}
d_{n+2} &= \sum_{S_{01}\in D_n} \sum_{S_{10}\in D_n} \eta(S_{01} \cap S_{10}) \cdot \eta(S_{01}^* \cap S_{10}^*) \\
&= (*\text{ Definition of } R_n *) \\
&\quad \sum_{K\in R_n} \sum_{S_{01}\sim K} \sum_{S_{10}\in D_n} \eta(S_{01} \cap S_{10}) \cdot \eta(S_{01}^* \cap S_{10}^*) \\
&= (*\text{ Definition (2.17) } *) \\
&\quad \sum_{K\in R_n} \sum_{\alpha\in p(K)} \sum_{S_{10}\in D_n} \eta(\alpha(K) \cap S_{10}) \cdot \eta(\alpha(K)^* \cap S_{10}^*) \\
&= (* \ \forall \alpha \in \textsc{perms}_n, D_n = \{\alpha(S) \mid S \in D_n\} = \alpha[D_n] *) \\
&\quad \sum_{K\in R_n} \sum_{\alpha\in p(K)} \sum_{S_{10}\in D_n} \eta(\alpha(K) \cap \alpha(S_{10})) \cdot \eta(\alpha(K)^* \cap \alpha(S_{10})^*) \\
&= (* \ \alpha(X) \cap \alpha(Y) = \alpha(X \cap Y) *) \\
&\quad \sum_{K\in R_n} \sum_{\alpha\in p(K)} \sum_{S_{10}\in D_n} \eta(\alpha(K \cap S_{10})) \cdot \eta(\alpha(K^* \cap S_{10})^*) \\
&= (*\eta(T) = \eta(\alpha(T)), \gamma(K) = \#p(K) *) \\
&\quad \sum_{K\in R_n} \sum_{S_{10}\in D_n} \gamma(K) \cdot \eta(K \cap S_{10}) \cdot \eta(K^* \cap S_{10}^*)
\end{aligned}$$

So now, instead of looping over D_n, the outer loop of algorithm 3 can be replaced by a loop over R_n. This means the number of iterations is significantly decreased. This is put together in algorithm 4.

2.2 Representation

Before starting on the implementation of the algorithms, an alternative way of representing sets is described. This is useful, because otherwise sets will

Algorithm 4: Compute d_{n+2} from D_n and R_n

```
Input: $D_n$ containing all monotone subsets in $Q(n)$ and $R_n$
       containing the least representative of each equivalence class
       in $D_n$, with respect to the lexicographical ordening
Output: $d_{n+2}$, cardinality of $D_{n+2}$
$result \leftarrow 0$;
foreach $K \in R_n$ do
    foreach $T \in D_n$ do
        $result := result + \gamma(K) \cdot \eta(K \cap T) \cdot \eta(K^* \cap T^*)$
return $result$;
```

get complicated and large, which is not preferable. For example, when $n = 6$, subsets of $Q(n)$ can have sizes equal to $2^6 = 64$. Since for calculating d_8, the monotone subsets of $Q(6)$ are needed, this will result in big sets and will take lots of memory.

2.2.1 Power set $Q(n)$

Since the sets in $Q(n)$ contain elements of the set $V(n) = \{0, 1, \dots n - 1\}$, they can be described as an array of bits. The bit on position i indicates whether or not a i is in the set. This way, the elements of $Q(n)$ can be represented very efficiently, since they have a constant size of n bits. Furthermore, there is a natural order (e.g. the lexicographical order), which is useful when representing the monotone subsets.

For example, take $n = 3$, so $Q(n) = \{012, 12, 02, 2, 01, 1, 0, \varnothing\}$. Each element can only contain 0, 1 and 2. So if we represent $Q(n)$ using bit arrays

$$Q(n) = \{111, 011, 101, 001, 110, 010, 100, 000\} \tag{2.18}$$

Using this representation, it is really efficient to represent and also really easy construct $Q(n)$, since this is the same as constructing $[0, 2^n)$ in binary.

2.2.2 Monotone Subsets

Any subset $S \subseteq Q(n)$ can be described as an array of bits of length 2^n. Each bit indicates whether an element $x \in Q(n)$, is in S or not. For the monotone subsets to be represented this way, $Q(n)$ has to be sorted.

For example, take $n = 3$. Now take $S = \{01, 02, 012\}$. The binary representation of S is shown in table 2.1. So in binary representation $S = 10101000$. Observe that $Q(n)$ is sorted in lexicographical order.

$Q(n)$	012	12	02	2	01	1	0	$\varnothing$
S	1	0	1	0	1	0	0	0

Table 2.1: Binary representation of $\{01, 02, 012\}$

This representation for sets is used in the implementation of the program.

Chapter 3

Implementation

The program is implemented with the algorithms described in section 2.1 as a basis. This way, it is made sure the theory is correct and the focus can be on improving the code. Also the representation of sets described in section 2.2 is followed to store sets in an efficient way.

This project's source is also available on `github.com/arjenzijlstra`.

3.1 Representation

For most of the data C++ containers provided by the *Standard Template Library* are used. A container is an object that stores a collection of other objects (its elements). This way accessors, iterators and generic algorithms are provided by the STL. Accessors are used to access the stored information. Iterators represent the begin and the end of the stored data and are used to iterate over the elements. Generic algorithms can be used for all sorts of operations. Two algorithms that are useful for us are `copy` and `find`, to copy containers and to search for elements in containers respectively.

3.1.1 Monotone Subsets

At first, monotone subsets were represented as `std::set<std::set<size_t>>`, i.e. sets of sets of numbers. This choice was made, because no doubles are allowed in the sets. This resulted in one big set containing all of these monotone subsets, taking a lot of memory. Also, the sets were sorted every time an element was inserted or modified, which is also very slow.

To solve the issue with the sorting of the elements on every insertion, instead of `std::set`, `std::vector` can be used in many cases. This way, the elements are not sorted. This does not give any problems, since most of the algorithms used, do not require sorting of the elements, nor will it result in double elements in the set. Though for some sets, still `std::set <size_t>` is used, since sorting might be required to prevent doubles.

To decrease the memory usage of the sets, the alternative representation of monotone subsets can be used. Since these can be described as an array of bits, `std::bitsets` are used. This choice is made, so they can be manipulated by standard logic operators and they can be converted to integers.

This results in a final representation of monotone subsets as `std::bitset<size>` where `size` is equal to 2^n. Now D_n is of the form `std::vector<std::bitset<size>>`.

3.1.2 Dedekind Numbers

Since d_8 is bigger than 2^{64}, it is not possible to store it in a standard datatype provided by C++ . For this, an unsigned integer containing two 64-bits numbers is implemented. This is done by creating a class `UInt128` containing two `uint_fast64_t`'s. `uint_fast64_t` is chosen, since this is the fastest datatype with at least 64-bits.

Also, since just the **`operator`**+ is needed to compute the number, only that operator is implemented. This implementation is done by simply checking whether a carry is needed or not. If so, increase the high-part by one. Note, **`operator`**+ makes use of **`operator`**+=, which is implemented as follows.

Listing 3.1: `uint128/operatoraddassign1.cpp`

```

#include "uint128.ih"

namespace Dedekind
{
  UInt128 &UInt128::operator+=(UInt128 const &other)
  {
    if (d_lo > std::numeric_limits<unsigned long>::max() - other.d_lo)
    {
      ++d_hi;
    }

    d_lo += other.d_lo;
    d_hi += other.d_hi;
    return *this;
  }
}
```

Lastly, an **operator**<< would come in handy to print the result. This was first implemented by just multiplying the high-part by `pow(2, 64)` and than add the low-part. This resulted in a small error, which was caused by the precision of a double. The (*incorrect*) answer obtained using this method is:

$$56\ 130\ 437\ 228\ 687\ 561\ 588\ 736$$

To solve this problem, the **operator**<< is implemented differently. First an array of numbers is created, the all digits are computed by looping over the bits, increase by one if a bit is set, multiply by 2 every next bit, and at the same time keeping track of the carries that occur (see the code for more details). This approach results in the right answers for big numbers ($> 2^{64}$, thus also for d_8).

Listing 3.2: `uint128/operatorinsert.cpp`

```

#include "uint128.ih"

// http://stackoverflow.com/questions/4361441/c-print-a-biginteger-in-base-10

namespace Dedekind
{
  std::ostream &operator<<(std::ostream &out, UInt128 const &uint128)
  {
    size_t d[39] = {0}; // a 128 bit number has at most 39 digits

    // starting at the highest, for each bit
    for (int iter = 63; iter != -1; --iter)
    {
      // increase the lowest digit if this bit is set
      if ((uint128.d_hi >> iter) & 1)
      {
        d[0]++;
      }

      // multiply by 2, since bits represent powers of 2
      for (size_t idx = 0; idx < 39; ++idx)
      {
        d[idx] *= 2;
      }

      // handle carries/overflow
      for (size_t idx = 0; idx < 38; ++idx)
      {
        d[idx + 1] += d[idx] / 10;
        d[idx] %= 10;
      }
    }

    for (int iter = 63; iter > -1; --iter)
    {
      // increase the lowest digit if this bit is set
      if ((uint128.d_lo >> iter) & 1)
      {
        d[0]++;
      }

      // only multiply if more bits will follow
      if (iter > 0)
      {
        for (size_t idx = 0; idx < 39; ++idx)
        {
          d[idx] *= 2;
        }
      }

      // handle carries/overflow
      for (size_t idx = 0; idx < 38; ++idx)
      {
        d[idx + 1] += d[idx] / 10;
        d[idx] %= 10;
      }
    }

    // find highest digit to be inserted in outputstream
    int idx;
    for (idx = 38; idx > 0; --idx)
    {
      if (d[idx] > 0)
      {
        break;
      }
    }

    // insert from here
    for (; idx > -1; --idx)
    {
      out << d[idx];
    }

    return out;
  }
}
```

3.2 Algorithms

The algorithms described in section 2.1 are implemented in such a way that it is not only fast, but also readable and easy to recognise.

3.2.1 Generating D_{n+1} from D_n

In listing 3.3 the implementation for generating D_{n+1} is given. Iterators are used to loop over the set D_n. Also, the `bitsets` contained in the sets returned by this function are twice the size of the `bitsets` received. To use this function, the **`operator`**`<=` is needed, and also `concatenate` is needed. Both are described in section 3.3.

Listing 3.3: Generate D_n

```
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));
      }
    }
  }

  return dn1;
}
```

3.2.2 Computing d_{n+2} from D_n and R_n *(in parallel)*

In listing 3.4 the implementation for computing D_{n+2} is given. Iterators are used to loop over the set D_n, while for iterating over R_n the index operator is used. This is because of the parallelisation using `MPI`, which is described later. To implement this algorithm, `dual` and `eta` are needed, to return the dual of a `bitset` respectively the η-value. Also `BitSetLess` is needed to sort `bitsets` on integer value. All three are described in section 3.3.

Furthermore, some choices are made within the implementation of this function.

First of all, preprocessing is done for all elements of D_n. This means that the duals and η-values are calculated for each element. These values are saved in `maps`. This is because a `map` has a very fast lookup, which is relevant later in this algorithm[1]. Since the elements within each class in R_n

[1]A `map` is usually implemented as a red-black tree. See the *Containers* library on `cppreference.com`

all have the same η-value, these can all be calculated at once, and then be added to the map, per element. The duals can be calculated and added to a separate map at the same time.

When the preprocessing is complete, the implementation is continued as in algorithm 4. The first element of each class in R_n is used as S. Now the bitwise AND is equal to the intersection of two sets. Also the cardinality of each set in R_n is equal to the gamma-value.

The function returns a `UInt128`, which is an unsigned integer implemented containing 2 64bits numbers. This way there is enough space for d_8. For more details, see section 3.1.2.

The implementation is parallelised using `MPI`. The choice is made to keep it simple, and to only parallelise the big nested for-loop. This is done using quite a simple strategy: every process starts at the element at index equal to their id and each iteration it increments the iterator by the total number of processes. This way, R_n is divided in quite a balanced way. Since there is not a really big difference in load balancing this way, no other distributed memory strategies are tried, since the expectations are that it will not achieve a significant higher speedup.

Listing 3.4: Compute d_n

```cpp
template <size_t size>
UInt128 compute(std::vector<std::bitset<size>> const &dn,
    std::vector<std::vector<std::bitset<size>>> const &rn,
    size_t rank = 0, size_t nprocs = 1)
{
  std::map<std::bitset<size>, std::bitset<size>, BitSetLess> duals;
  std::map<std::bitset<size>, size_t, BitSetLess> etas;

  // Preprocess duals and eta's of all elements
  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);
    }
  }
  // Preprocessing complete


  UInt128 result;
  for (size_t idx = rank; idx < rn.size(); idx += nprocs)
  {
    auto iter(rn[idx].begin());
    for (auto iter2 = dn.begin(); iter2 != dn.end(); ++iter2)
    {
      auto first = *iter & *iter2;
      auto second = duals[*iter] & duals[*iter2];

      result += rn[idx].size() * etas[first] * etas[second];
    }
  }

  return result;
}
```

3.2.3 Generating R_n

Besides the helper functions used in listing 3.3 and listing 3.4, only a function to generate R_n is needed. Since this involves mostly performing permutations, more helper functions are needed for this. First all permutations are generated, using `Internal::permutations`.

After that, for each element in D_n, the whole equivalent class is generated, using alle permutations, and put in a vector. Then the vector is added to R_n as a whole, since the equivalent elements can be used when calculating the η-values. To make sure that no classes are added twice, a `set` keeps record of all processed monotone subsets. At first, this was done using a `vector`, to have a fast insertion. But since the lookup is far more important, a `set` is faster in the end.

Also another problem occured because of some C++ problem. There are two simple ways of finding an element in a `set`,
`std::find(processed.begin(), processed.end(), *iter)`
and,
`processed.find(*iter)`
will both do the job. Although they look very similar, there is one small difference that makes a huge difference. The one that makes use of `std::find` provided by the `algorithms` library, simply checks every element to find the one it is looking for. Now the one that uses the `find` member of the `set`, uses the underlying structure[2] to find the one it is looking for. This way, checking for an element is a lot quicker.

Listing 3.5: Generate R_n

```
template <size_t Number, size_t Power>
std::vector<std::vector<std::bitset<Power>>> generateRn(
    std::vector<std::bitset<Power>> const &dn)
{
  auto permutations = Internal::permutations<Number, Power>();
```

[2]A `set` is usually implemented as a red-black tree. See the *Containers* library on `cppreference.com`

```

  std::vector<std::vector<std::bitset<Power>>> rn;
  std::set<std::bitset<Power>, BitSetLess> processed;
  for (auto iter = dn.begin(); iter != dn.end(); ++iter)
  {
    if (processed.find(*iter) == processed.end())
    {
      auto equivs = Internal::equivalences(*iter, permutations);

      std::vector<std::bitset<Power>> permuted;
      copy(equivs.begin(), equivs.end(), std::back_inserter(permuted));
      for (auto perm = equivs.begin(); perm != equivs.end(); ++perm)
      {
        processed.insert(*perm);
      }

      rn.push_back(permuted);
    }
  }

  return rn;
}
```

3.3 Helper Functions

To implement the algorithms from section 2.1, extra functionality is needed. For this, an `Internal` namespace is used, containing this functionality.

3.3.1 Operator<=

The **`operator`**`<=` indicates whether the monotone subset on the left-hand side is contained in the monotone subset on the right-hand side. This is translated into bit arrays by checking if a bit is true in the set at the left that is not true in the set on the right.

Listing 3.6: Operator<=

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

3.3.2 Concatenate

Concatenating two `bitsets` is done by creating a new `bitset` with a size twice as big as the input `bitsets`. Now both `bitsets` are translated to a string and concatenated using the `std::string`, **`operator`**`+`. The result of this is used as initialiser for the resulting `bitset`.

Listing 3.7: Concatenate

```cpp
template <size_t size>
std::bitset<(size << 1)> concatenate(std::bitset<size> const &lhs,
    std::bitset<size> const &rhs)
{
  std::string lhs_str = lhs.to_string();
  std::string rhs_str = rhs.to_string();

  return std::bitset<(size << 1)>(lhs_str + rhs_str);
}
```

3.3.3 Dual

The dual operation is defined in section 2.1 equation 2.10. It is defined as $S^* = \{t^c : t \in S\}^c$, which is exactly the same as writing the `bitset` in reverse, and also flipping all bits. For that, `reverse` is implemented and used to implement `dual`.

Listing 3.8: Dual

```cpp
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();
}
```

3.3.4 Eta

The η-value of some monotone subset T is defined in 2.1 equation 2.14 as the number of members of D_n contained in T. This is calculated by looping over the elements of D_n, and counting the number of elements that are contained in T.

Listing 3.9: Eta

```cpp
template <size_t size>
size_t eta(std::bitset<size> const &bset,
        std::vector<std::bitset<size>> const &dn)
{
  size_t result = 0;
  for (size_t idx = 0; idx < dn.size(); ++idx)
  {
    if (dn[idx] <= bset)
    {
      ++result;
    }
  }
  return result;
}
```

3.3.5 Permutations

Permutations can be calculated using `std::next_permutation`. Every iteration, it will permute the elements in `permutation` in a structured way. Using this on an array containing n elements, this results in a vector, containing all permutations on n elements. Also the power set $Q(n)$ is generated, because this is needed to generate the permutations on subsets of $Q(n)$, which is done by `subsetPermutation`. `subsetPermutation` is also implemented in the namespace `Internal`.

Listing 3.10: Permutations on n elements

```
template <size_t Number, size_t Power>
std::vector<std::array<size_t, Power>> permutations()
{
  std::vector<std::bitset<Number>> powerset =
      Internal::PowerSet<Number>::powerSetBin();

  std::array<size_t, Number> permutation;
  for (size_t idx = 0; idx != Number; ++idx)
  {
    permutation[idx] = idx;
  }

  std::vector<std::array<size_t, Power>> result;
  do
  {
    result.push_back(
        Internal::subsetPermutation<Number, Power>(permutation, powerset));
  }
  while (std::next_permutation(permutation.begin(), permutation.end()));

  return result;
}
```

3.3.6 Power Set

To generate the power set $Q(n)$, the algorithm described in section 2.1 is used. This is done, using template-meta programming in C++ . This way, parts are already known compile-time. So it does not slow down the computation. $Q(n)$ is represented using `bitsets`, just like subsets of $Q(n)$ are represented.

Listing 3.11: Power Set $Q(n)$

```
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>() });
}
```

3.3.7 Subset Permutation

To generate the permutation on a subset of $Q(n)$, from a permutation on the elements within the sets in $Q(n)$, the permutation is applied to the numbers in every set in $Q(n)$. Then for every set S, the index of the obtained element is set as destination for set S. This way, a permutation on $Q(n)$ is generated from a permutation on $V(n)$.

Listing 3.12: Subset Permutation

```
template <size_t Number, size_t Power>
std::array<size_t, Power> subsetPermutation(
    std::array<size_t, Number> const &permutation,
    std::vector<std::bitset<Number>> const &pset)
{
  std::array<size_t, Power> result;
  size_t idx = 0;
  for (auto iter = pset.begin(); iter != pset.end(); ++iter)
  {
    std::bitset<Number> tmp = permute(permutation, *iter);
    result[idx++] = find(pset.begin(), pset.end(), tmp) - pset.begin();
  }
  return result;
}
```

3.3.8 Permutation

When permuting a set, the bits of the `bitset` are swapped, according to the permutation. This is equal for $Q(n)$ and $S \subseteq Q(n)$, since both are represented as arrays of bits.

Listing 3.13: Perform Permutation

```
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;
}
```

3.3.9 Equivalence class

To generate all equivalences of a certain element, all permutations should be performed on the element. In this case, a `set` is needed as a container, since no double elements are allowed.

Listing 3.14: Equivalences of S

```
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;
}
```

3.3.10 Sorting `bitsets`

Bitsets are sorted on integer value from high to low. This way, the sets will always output the sets containing most elements first. This way, they will be printed from high to low, which follows the structure of the lattices.

Listing 3.15: Bitset Compare

```cpp
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();
    }
};
```

Chapter 4

Findings and Future Work

This thesis, followed the strategy Wiedemann (1991) used in 1991 to compute d_8. In 1991, this took 200 hours to compute on a Cray-2. Today, taking the same approach, on 144 cores of the millipede cluster of te University of Groningen, it took less than 30 minutes (1737.56 seconds to be exact). On 12 cores it took about 12338.2 seconds ($\approx 3,5$ hours). Which gives a speedup of about 7.1

Using this same approach to compute d_9, is not possible yet. There are some factors to take into account when taking this conclusion. First of all, D_7 is needed for this. The storage needed for any element in D_7 is equal to 128 bits. Since $d_7 = 2414682040998$, this will take at least $2414682040998 \cdot 128$ bits $\approx 38,63$ TB of storage. This amount of fast memory is not available yet. Furthermore, the time needed to computed d_9 would be really long. The time needed to compute d_6 is about 0.001 seconds, for d_7, this is about 0.1 seconds and for d_8, it takes about 3 hours (all on 12 cores), which is over 10 000 seconds. So, if the time to compute d_9, would increase as much from d_8 as it increased from d_7 to d_8 (which is not realistic at all, even better than a best-case scenario), this would take about 1 000 000 000 seconds on 12 cores. Imagine that around 1000 cores are available, it would still take around 10 000 000 seconds, which is equal to 115 days. Since this is estimated very optimistic, it can be concluded that computing d_9, using this approach is not possible yet.

While computing d_9 using Wiedemann's approach is not achievable, Fidytek et al. (2001) describe a way of computing d_{n+4} from D_n. Since

d_5 is much smaller than d_7, it might be worthwhile looking at this approach to try to compute d_9. Also Bakoev (2012) use a similar approach with matrices. He is even optimistic about computing d_9 in a reasonable amount of time, given that an efficient solution is found for counting the elements of one of the cases described.

Furthermore, for improving the approach taken in this thesis, a lot of built-in C++ functionality, could be implemented in plain C using less templates, to make the program more cross-platform and flexible. Also, this could make it a little more efficient, since the functionality can be made more specific. This could result in a little bit faster program, and might help to find higher values of d_n.

Finally, some other parallelisation strategies could be tried. At the moment, R_n is statically divided into separate blocks, since the focus was more on improving the algorithm itself. Using this strategy, the difference between the first and the last finished process was between 1 and 5 %. Using a strategy which uses a dynamic division of the blocks. The difference would probably be less than when using a static division, but since more communication is needed, this could give a lot overhead, which could result in an overal slower program. Also, some more parts of the program could be parallelised. At the moment, just the nested for-loop, that costs the most time is parallelised. For example, parts as generating D_6 and R_6 are computed sequential, since these parts are negligable to computing d_8. Also the preprocessing is done sequential, to minimise communication.

References

Bakoev, V. (2012). One more way for counting monotone boolean functions. In *Proc. of the XIII Intern. Workshop on Algebraic and Combinatorial Coding Theory (ACCT)*, pages 15–21.

Berman, J. and Köhler, P. (1976). Cardinalities of finite distributive lattices. *Mitt. Math. Sem. Giessen*, 121:103–124.

Church, R. (1940). Numerical analysis of certain free distributive structures. *Duke Math. J*, 6(3):732–734.

Church, R. (1965). Enumeration by rank of the elements of the free distributive lattice with seven generators. *Notices Amer. Math. Soc*, 12:724.

Dedekind, R. (1897). *Ueber Zerlegungen von Zahlen durch ihre grössten gemeinsamen Theiler*. F. Vieweg & Sohn.

Fidytek, R., Mostowski, A. W., Somla, R., and Szepietowski, A. (2001). Algorithms counting monotone boolean functions. *Information Processing Letters*, 79:203–209.

Stephen, T. and Yusun, T. (2012). Counting inequivalent monotone boolean functions. *arXiv preprint arXiv:1209.4623*.

Ward, M. (1946). Note on the order of free distributive lattices. *Bull. Amer. Math. Soc*, 52(5):423.

Wiedemann, D. (1991). A computation of the eighth dedekind number. *Order*, 8:5–6.

Yusun, T. J. (2011). Dedekind numbers and related sequences. Master's thesis, Simon Fraser University.

Appendix A

Source Code

This source is also available on `github.com/arjenzijlstra`.

A.1 Algorithms on Monotone Subsets

Listing A.1: `dedekind/dedekind.h`

```cpp

#ifndef DEDEKIND_H_
#define DEDEKIND_H_

#include <algorithm>
#include <bitset>
#include <iostream>
#include <map>
#include <set>
#include <vector>

#include "../uint128/uint128.h"

#include "bitsetless.h"
#include "bitsetoperleq.h"
#include "operwiedemann.h"
#include "permutations.h"
#include "powerof2.h"
#include "powersetbin.h"
#include "vectoroperinsert.h"

namespace Dedekind
{
  enum
  {
    BIGINTTAG
  };

  template <size_t Number, size_t Power>
  std::vector<std::vector<std::bitset<Power>>> generateRn(
      std::vector<std::bitset<Power>> const &dn)
  {
    auto permutations = Internal::permutations<Number, Power>();

    std::vector<std::vector<std::bitset<Power>>> rn;
    std::set<std::bitset<Power>, BitSetLess> processed;
    for (auto iter = dn.begin(); iter != dn.end(); ++iter)
    {
      if (processed.find(*iter) == processed.end())
      {
        auto equivs = Internal::equivalences(*iter, permutations);

        std::vector<std::bitset<Power>> permuted;
        copy(equivs.begin(), equivs.end(), std::back_inserter(permuted));
        for (auto perm = equivs.begin(); perm != equivs.end(); ++perm)
        {
          processed.insert(*perm);
        }

        rn.push_back(permuted);
      }
    }

    return rn;
  }

  template <size_t size>
  UInt128 compute(std::vector<std::bitset<size>> const &dn,
      std::vector<std::vector<std::bitset<size>>> const &rn,
      size_t rank = 0, size_t nprocs = 1)
  {
    std::map<std::bitset<size>, std::bitset<size>, BitSetLess> duals;
    std::map<std::bitset<size>, size_t, BitSetLess> etas;

    // Preprocess duals and eta's of all elements
    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);
      }
    }
    // Preprocessing complete


    UInt128 result;
    for (size_t idx = rank; idx < rn.size(); idx += nprocs)
    {
      auto iter(rn[idx].begin());
      for (auto iter2 = dn.begin(); iter2 != dn.end(); ++iter2)
      {
        auto first = *iter & *iter2;
        auto second = duals[*iter] & duals[*iter2];

        result += rn[idx].size() * etas[first] * etas[second];
      }
    }

    return result;
  }

  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));
        }
      }
    }

    return dn1;
  }

  namespace Internal
  {
    template <size_t Number>
    struct MonotoneSubsets
    {
      static std::vector<std::bitset<PowerOf2<Number>::value>> result;
    };

    template <size_t Number>
    std::vector<std::bitset<PowerOf2<Number>::value>>
        MonotoneSubsets<Number>::result(generate(
            MonotoneSubsets<(Number - 1)>::result));

    template <>
    struct MonotoneSubsets<0>
    {
      static std::vector<std::bitset<1>> result;
    };

    std::vector<std::bitset<1>> MonotoneSubsets<0>::result({std::bitset<1>(0),
        std::bitset<1>(1)});
  }

  template <size_t Number>
  UInt128 monotoneSubsets(size_t rank = 0, size_t size = 1)
  {
    auto dn = Internal::MonotoneSubsets<Number - 2>::result;

    std::cerr << "Rank " << rank << " is done generating D"
        << Number - 2 << ": " << dn.size() <<  '\n';

    auto rn = generateRn<Number - 2>(dn);

    std::cerr << "Rank " << rank << " is done generating R"
        << Number - 2 << ": " << rn.size() << '\n';

      UInt128 result = compute(dn, rn, rank, size);

      return result;
  }
}


#endif // end of guard DEDEKIND_H_
```

Listing A.2: `dedekind/dedekind.h`

```

#ifndef BITSETLESS_H_
#define BITSETLESS_H_

#include <bitset>
#include <cstddef>


namespace Dedekind
{

```

```

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();
    }
};

template <size_t size>
bool bitsetLess(std::bitset<size> const &lhs, std::bitset<size> const &rhs)
{
  return lhs.to_ulong() > rhs.to_ulong();
}

} // namespace Dedekind


#endif
```

Listing A.3: `dedekind/dedekind.h`

```

#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 A.4: `dedekind/dedekind.h`

```

#ifndef OPERWIEDEMANN_H_
#define OPERWIEDEMANN_H_

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

#include "bitsetoperleq.h"


namespace Dedekind
{

namespace Internal
{
```

```cpp


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,
        std::vector<std::bitset<size>> const &dn)
{
  size_t result = 0;
  for (size_t idx = 0; idx < dn.size(); ++idx)
  {
    if (dn[idx] <= bset)
    {
      ++result;
    }
  }
  return result;
}

template <size_t size>
std::bitset<(size << 1)> concatenate(std::bitset<size> const &lhs,
    std::bitset<size> const &rhs)
{
  std::string lhs_str = lhs.to_string();
  std::string rhs_str = rhs.to_string();

  return std::bitset<(size << 1)>(lhs_str + rhs_str);
}


} // namespace Internal

} // namespace Dedekind


#endif
```

Listing A.5: `dedekind/dedekind.h`

```cpp

#ifndef PERMUTATIONS_H_
#define PERMUTATIONS_H_

#include <algorithm>
#include <array>
#include <bitset>
#include <set>
#include <vector>

#include "powersetbin.h"


namespace Dedekind
{
```

```

namespace Internal
{


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;
}

template <size_t Number, size_t Power>
std::array<size_t, Power> subsetPermutation(
    std::array<size_t, Number> const &permutation,
    std::vector<std::bitset<Number>> const &pset)
{
  std::array<size_t, Power> result;
  size_t idx = 0;
  for (auto iter = pset.begin(); iter != pset.end(); ++iter)
  {
    std::bitset<Number> tmp = permute(permutation, *iter);
    result[idx++] = find(pset.begin(), pset.end(), tmp) - pset.begin();
  }
  return result;
}

template <size_t Number, size_t Power>
std::vector<std::array<size_t, Power>> permutations()
{
  std::vector<std::bitset<Number>> powerset =
      Internal::PowerSet<Number>::powerSetBin();

  std::array<size_t, Number> permutation;
  for (size_t idx = 0; idx != Number; ++idx)
  {
    permutation[idx] = idx;
  }

  std::vector<std::array<size_t, Power>> result;
  do
  {
    result.push_back(subsetPermutation<Number, Power>(permutation, powerset));
  }
  while (std::next_permutation(permutation.begin(), permutation.end()));

  return result;
}

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;
}


} // namespace Internal

} // namespace Dedekind


#endif
```

Listing A.6: `dedekind/dedekind.h`

```

#ifndef POWEROF2_H_
#define POWEROF2_H_

#include <cstddef>

namespace Dedekind
{

namespace Internal
{


template<size_t Number>
struct PowerOf2
{
  static size_t const value;
};

template<size_t Number>
size_t const PowerOf2<Number>::value((PowerOf2<Number - 1>::value << 1));


template<>
struct PowerOf2<0>
{
  static size_t const value;
};

size_t const PowerOf2<0>::value(1);



template<size_t Number>
struct LogOf2
{
  static size_t const value;
};

template<size_t Number>
size_t const LogOf2<Number>::value(LogOf2<(Number >> 1)>::value + 1);


template<>
struct LogOf2<1>
{
  static size_t const value;
};

size_t const LogOf2<1>::value(0);


} // namespace Internal

} // namespace Dedekind


#endif
```

Listing A.7: `dedekind/dedekind.h`

```

#ifndef POWERSETBIN_H_
#define POWERSETBIN_H_

#include <vector>
#include <bitset>

namespace Dedekind
{

namespace Internal
{


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>() });
}


} // namespace Internal

} // namespace Dedekind


#endif
```

A.2 128bits unsigned integer

A.2.1 Headers

Listing A.8: `uint128/uint128.h`

```

#ifndef DEDEKIND_UINT_
#define DEDEKIND_UINT_

#include <cstdint>
#include <iosfwd>

namespace Dedekind
{
  class UInt128
  {
    uint_fast64_t d_hi;
    uint_fast64_t d_lo;

    public:
      friend std::ostream &operator<<(std::ostream &out,
          UInt128 const &uint128);

      UInt128(UInt128 const &other) = default;
      UInt128(uint_fast64_t lo = 0, uint_fast64_t hi = 0);

      UInt128 &operator+=(uint_fast64_t other);
      UInt128 &operator+=(UInt128 const &other);

      uint_fast64_t hi() const;
      uint_fast64_t lo() const;
  };


  inline UInt128 operator+(UInt128 const &lhs, UInt128 const &rhs)
  {
    UInt128 tmp(lhs);
    return tmp += rhs;
  }

  inline uint_fast64_t UInt128::hi() const
  {
    return d_hi;
  }

  inline uint_fast64_t UInt128::lo() const
  {
    return d_lo;
  }
}

#endif
```

Listing A.9: `uint128/uint128.ih`

```

#include "uint128.h"

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

using namespace std;
```

A.2.2 Sources

Listing A.10: `uint128/uint128.cpp`

```

#include "uint128.ih"

namespace Dedekind
{
  UInt128::UInt128(uint_fast64_t lo, uint_fast64_t hi)
  :
    d_hi(hi),
    d_lo(lo)
  {
  }
}
```

Listing A.11: `uint128/operatorinsert.cpp`

```

#include "uint128.ih"

// http://stackoverflow.com/questions/4361441/c-print-a-biginteger-in-base-10

namespace Dedekind
{
  std::ostream &operator<<(std::ostream &out, UInt128 const &uint128)
  {
    size_t d[39] = {0}; // a 128 bit number has at most 39 digits

    // starting at the highest, for each bit
    for (int iter = 63; iter != -1; --iter)
    {
      // increase the lowest digit if this bit is set
      if ((uint128.d_hi >> iter) & 1)
      {
        d[0]++;
      }

      // multiply by 2, since bits represent powers of 2
      for (size_t idx = 0; idx < 39; ++idx)
      {
        d[idx] *= 2;
      }

      // handle carries/overflow
      for (size_t idx = 0; idx < 38; ++idx)
      {
        d[idx + 1] += d[idx] / 10;
        d[idx] %= 10;
      }
    }

    for (int iter = 63; iter > -1; --iter)
    {
      // increase the lowest digit if this bit is set
      if ((uint128.d_lo >> iter) & 1)
      {
        d[0]++;
      }

      // only multiply if more bits will follow
      if (iter > 0)
      {
        for (size_t idx = 0; idx < 39; ++idx)
        {
          d[idx] *= 2;
        }
      }

      // handle carries/overflow
      for (size_t idx = 0; idx < 38; ++idx)
      {
        d[idx + 1] += d[idx] / 10;
        d[idx] %= 10;
      }
    }

    // find highest digit to be inserted in outputstream
    int idx;
    for (idx = 38; idx > 0; --idx)
    {
      if (d[idx] > 0)
      {
        break;
      }
    }

    // insert from here
    for (; idx > -1; --idx)
    {
      out << d[idx];
    }

    return out;
  }
}
```

Listing A.12: `uint128/operatoraddassign1.cpp`

```

#include "uint128.ih"

namespace Dedekind
{
  UInt128 &UInt128::operator+=(UInt128 const &other)
  {
    if (d_lo > std::numeric_limits<unsigned long>::max() - other.d_lo)
    {
      ++d_hi;
    }

    d_lo += other.d_lo;
    d_hi += other.d_hi;
    return *this;
  }
}
```

Listing A.13: `uint128/operatoraddassign2.cpp`

```

#include "uint128.ih"

namespace Dedekind
{
  UInt128 &UInt128::operator+=(uint_fast64_t other)
  {
    if (d_lo > std::numeric_limits<unsigned long>::max() - other)
    {
      ++d_hi;
    }

    d_lo += other;
    return *this;
  }
}
```

A.3 Main

Listing A.14: `main.ih`

```

#include <iostream>
#include <sstream>
#include <mpi.h>
#include <sys/time.h>

#include "dedekind/dedekind.h"
#include "uint128/uint128.h"

using namespace std;


struct timeval tim2;


typedef Dedekind::UInt128 (*fptr)(size_t, size_t);

template <size_t a = 8>
fptr findFunction(size_t b)
{
  if (a == b)
  {
    return Dedekind::monotoneSubsets<a>;
  }
  else
  {
    return findFunction<a - 1>(b);
  }
}

template <>
fptr findFunction<6>(size_t b)
{
  return Dedekind::monotoneSubsets<6>;
}
```

Listing A.15: `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<N>(rank, size);
    Dedekind::UInt128 result = findFunction(n)(rank, size);

    double end = MPI::Wtime();
    cerr << "Rank " << rank << " done! Result: " << result << " in "
        << end - start << "s\n";

    // reduce over all processes
    if (rank == 0)
    {
      size_t toReceive = size;
      while (--toReceive)
      {
        // send the high and the low part of the result
        uint_fast64_t lohi[2];

        MPI::Status status;
        MPI::COMM_WORLD.Recv(lohi, 2, MPI::UNSIGNED_LONG,
            MPI::ANY_SOURCE, Dedekind::BIGINTTAG, status);

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

      double final = MPI::Wtime();
      cout << "d" << n << " = " << result
        << " (in " << final - start << "s)\n";
    }
    else
    {
      // send the high and the low part of the result
      uint_fast64_t lohi[2];
      lohi[0] = result.lo();
      lohi[1] = result.hi();
      MPI::COMM_WORLD.Send(&lohi, 2, 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();
}
```