Hamminggetallen
Een complexiteitsvergelijking tussen verschillende algoritmen

Jeroen Claassens

26 januari 2009

Inhoudsopgave

Hoofdstuk 1

Inleiding

1.1 Hamminggetallen en Hammingrij

De verzameling Hamminggetallen [1] bestaat uit alle natuurlijke getallen die een product van een 2,3 en 5-macht zijn:

$$H ::= \{x, y, z \in \mathbb{N} \mid 2^x \cdot 3^y \cdot 5^z\} \tag{1.1}$$

De Hammingrij H_i is de gesorteerde rij van alle Hamminggetallen. Deze rij heeft de volgende eigenschappen:

De Hammingrij is oplopend.

$$\forall_{i \in \mathbb{N}}(H_i < H_{i+1}) \tag{1.2}$$

Alle getallen in de Hammingrij zijn Hamminggetallen.

$$\forall_{i \in \mathbb{N}}(H_i \in H) \tag{1.3}$$

Alle Hamminggetallen zijn onderdeel van de Hammingrij.

$$\forall_{x \in H}(\exists_{i \in \mathbb{N}}(x = H_i)) \tag{1.4}$$

De Hammingrij is interessant omdat het een simpele definitie is van een rij, waarvan echter een efficiënt algoritme om het te genereren niet voor de hand ligt. Dijkstra [2] schrijft dit probleem toe aan R.W. Hamming. Voorstanders van functionele programmeertalen gebruiken de Hammingrij als voorbeeld om te demonstreren dat dit soort definities eenvoudiger te programmeren zijn in een functionele taal dan in een procedurele taal. Het voordeel van functionele talen is dat het eenvoudig is om oneindige lijsten te definiëren waardoor dit soort problemen uitermate geschikt zijn om in functionele talen uit te drukken. Hierdoor wordt de code uitermate leesbaar en snel te implementeren. Het wil echter niet zeggen dat het algoritme ook efficiënt is in de uitvoering. Deze scriptie laat zien dat een grotere mate

van efficiëntie bereikt kan worden door simpelweg een ander algorithme te construeren.

Dijkstra [1] gebruikte de Hammingrij in een voorbeeld voor wat hij *"taking the relation outside (the repetitive construct)"* noemt. Als voorbeeld is de uitwerking van Dijkstra prima, maar omdat het gebaseerd is op een algemeen principe, is het niet de meest efficiënte methode voor het bepalen van Hamminggetallen.

De algorithmen in dit artikel zijn gebouwd voor hun efficiëntie. Implementaties van de algorithmen in procedurele en functionele talen zullen code opleveren die niet in een oogopslag duidelijk maken wat de achterliggende gedachte is.

1.2 Complexiteit van Hammingrijalgoritmes

Om de efficiëntie van verschillende Hammingrijalgoritmes met elkaar te kunnen vergelijken is het wenselijk om een complexiteitsanalyse op deze algorithmes te verrichten. Helaas bevat het door ons gepresenteerde algoritme een lijst die erg onregelmatig groeit. De complexiteit van het algoritme is direct afhankelijk van de grootte van deze lijst waardoor het bepalen van de complexiteit van dit algoritme bijzonder moeilijk is. Om toch een uitspraak te kunnen doen over de complexiteit van de verschillende algoritmen maken we gebruik van Java implementaties, zie hoofdstuk 5, waarvan we de meetgegevens vergelijken.

1.3 n-smooth getallen

In de wiskunde zijn de n-smooth getallen [6], met n een priemgetal, gedefiniëerd als de verzameling van natuurlijke getallen met priemfactoren die kleiner of gelijk zijn aan n. Hamminggetallen zijn dus 5-smooth getallen.

De door ons gebruikte implementaties genereren lijsten van getallen met priemfactoren uit $\{2, 3, 5\}$. Bij Dijkstra's algoritme zijn deze getallen te vervangen door elk willekeurig getal. Ons algoritme heeft de beperking dat de factoren uit de verzameling onderling relatief priem moeten zijn. Vanwege de unieke-priemfactorisatie is elke rij die te genereren is met Dijksta's algoritme, ook uit te drukken met een set factoren die onderling relatief priem zijn.

Hoofdstuk 2

Complexiteit

2.1 Tijd- en geheugencomplexiteit

De complexiteit van een algorithme kan grofweg in twee belangrijke categoriën ingedeeld worden, de tijdcomplexiteit en de geheugencomplexiteit. Veel algorithmen maken gebruik van het feit dat het soms mogelijk is om de tijdcomplexiteit te verlagen ten koste van de geheugencomplexiteit (zie 2.3). De tijdcomplexiteit van een algoritme is direct gerelateerd aan de snelheid waarmee een implementatie het algoritme uitvoert.

2.2 Meeteenheden van complexiteit

2.2.1 Tijdcomplexiteit

We gebruiken de bitcomplexiteit [8] om de tijdcomplexiteit van onze algoritmen uit te drukken. Alle benodigde berekeningen zijn uit te drukken als optellingen, aftrekkingen, vermenigvuldigingen en delingen. Voor elk van deze vier basisoperaties wordt bijgehouden hoeveel bitoperaties ze benodigd hebben. het aantal benodigde bitoperaties hangt af van de grootte in bits van de input.

Een kleinere bitcomplexiteit wil echter nog niet zeggen dat het ook een sneller uitvoering heeft. De snelheid van de uitvoering is ook afhankelijk van de machine waarop de code wordt uitgevoerd. Karatsuba [5] heeft aangetoond dat vermenigvuldiging in $O(n^{log(3)})$ operaties kan. Dit houdt in dat Karatsuba bij zeer grote getallen zeker sneller is dan de traditionele methode, maar vanwege een zekere mate van administratie die nodig is voor Karatsuba, is er een drempelwaarde waarboven Karatsuba pas voordeliger wordt. De hoogte van de drempelwaarde hangt ondermeer af van het wel of niet hardwarematig geïmplementeerd zijn van Karatsuba. Wij gebruiken twee bitcomplexiteiten: één die Karatsuba gebruikt, en één die dat niet doet.

Strassen [7] heeft een nog snellere methode dan Karatsuba, maar deze

wordt pas bij getallen groter dan $2^{2^{17}} \approx 16$ kb [4]. Vanwege een codering voor Hamminggetallen die wij gebruiken, hoofdstuk 3, houdt dit in dat het voordeel pas optreed bij Hamminggetallen groter dan $2^{2^{2^{17}}} = 2^{2^{131072}}$. Omdat 1 Terabyte $\approx 2^{2^{40}}$ bits zijn de Hamminggetallen waarop Strassen sneller vermenigvuldigd niet eens in binaire code op te slaan in al het huidig beschikbare geheugen ter wereld.

Operatie	Tijdcomplexiteit
Optellen en aftrekken	$5 \cdot max(n, m) - 3$
Vermenigvuldigen en delen	$5nm - 3 \cdot max(n, m)$
Vermenigvuldigen en delen (Karatsuba)	$max(n, m)^{log(3)}$
Vergelijken	$min(n, m)$

Figuur 2.1: Gebruikte bitcomplexiteiten bij de primaire operaties op een n- en m-bits getal.

Figuur 2.1 geeft aan welke berekeningen gebruikt worden voor de verschillende operaties.

2.2.2 Geheugencomplexiteit

Het gebruike geheugen wordt simpelweg bepaald door het aantal bits dat gebruikt wordt door onze datastructuren. We kijken niet naar het gebruikte geheugen door het complete programma, omdat de implementaties van de algoritmen nergens recursief zijn, en dus een constante grootte hebben.

2.3 Tijdscomplexiteitswinst door geheugengebruik.

Het kan nuttig zijn als gedurende de uitvoer van het algorithme meerdere malen een zekere functie uitgerekend wordt voor een zekere input, deze waarde éénmaal uit te rekenen en de gevonden waarde tijdelijk op te slaan in een tabel. Bij het gebruik van een AVL-tree groeit de bitcomplexiteit logarithmisch ten opzichte van de grootte van de AVL-tree. Als een algorithme gebruik maakt van een tabel, moet voor de complexiteitsbepaling niet alleen de complexiteit van het opzoeken meegenomen worden, maar ook de complexiteit van het creëren van de tabel.

2.4 De praktijk

Als men in de praktijk een programma sneller wil maken, dan zijn hiervoor twee mogelijkheden: het verlagen van de tijdcomplexiteit van het algoritme of het parallel uitvoeren van delen van het algoritme op verschillende processoren. Als een algoritme parallel uitgevoerd wordt, heeft het echter nog

dezelfde tijdscomplexiteit omdat het aantal bitoperaties dat nodig is hierdoor niet veranderd. Een algoritme kan alleen parallel uitgevoerd worden als de berekeningen die uitgevoerd moeten worden in hoge mate onafhankelijk zijn.

Hoofdstuk 3

Gebruikte technieken

3.1 Gödelcodering

De Gödelcodering levert een eenvoudige representatie van Hamminggetallen. Elk Hamminggetal is daarin een tuple van drie waardes waarvoor geldt:

$$[\![x, y, z]\!] = 2^x \cdot 3^y \cdot 5^z$$

De Gödelcodering van een Hamminggetal is dus opgebouwd uit drie waardes die we in dit artikel de componenten van de Gödelcodering noemen.

Vermenigvuldiging van twee Gödelgecodeerde Hamminggetallen kan door het optellen van de afzonderlijke waardes in de Gödelcodering:

$$[\![x, y, z]\!] \cdot [\![n, m, p]\!] = [\![x + n, y + m, z + p]\!]$$

Optelling kan niet uitgevoerd worden in uitsluitend Gödelcodering. Als twee Hamminggetallen opgeteld moeten worden, zullen deze eerst naar een andere notatie, bijvoorbeeld binair, geconverteerd worden. Daarmee is het dus ook niet mogelijk direct voor twee willekeurige Hamminggetallen in Gödelcodering te bepalen wat de ordening is.

Een ander voordeel naast de snelheid van het vermenigvuldigen van Gödelcodering is dat dit een compactere notatie oplevert, wat geheugenruimte bespaart. Zo geldt:

$$9000 = [\![3, 2, 3]\!]$$

In binaire notatie levert dit:

$$10001100101000 = [\![11, 10, 11]\!]$$

Waardoor het getal 3*2=6 bits nodig heeft i.p.v. 14 bits.

Hoofdstuk 4

De algorithmen voor n-smooth getallen

4.1 Dijkstra's algoritme

Figuur 4.1 geeft een implementatie van Dijkstra's algoritme in Haskell.

```
scale n (x:xs) = (n * x) : (scale n xs)
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) = if x == y then
                          x : (merge xs ys)
                      else if x < y then
                          x : (merge xs (y:ys))
                      else
                          y : (merge (x:xs) ys)

seq = 1 : (merge (scale 2 seq)
                 (merge (scale 3 seq) (scale 5 seq)))
```

Figuur 4.1: Dijkstra's algoritme in Haskell.

De Haskell-implementatie is kort en duidelijk. De laatste regel definiëert de Hammingrij als het getal 1 gevolgd door dezelfde lijst vermenigvuldigd met 2, en gemerged met de lijst vermenigvuldigd met 3 en gemerged met de lijst vermenigvuldigd met 5. De implementaties van de scale en de merge zijn triviaal. Er zijn twee opmerkingen betreffende de efficiëntie van dit algoritme. Ten eerste moet om een i^e getal uit de rij te berekenen alle voorgaande getallen in het geheugen staan. Bij een grote i levert dit een enorme belasting op het geheugen. Ten tweede geeft de code `...ys) = if x == y then` ... aan dat er teveel berekend wordt.

Een alternatief voor het probleem van de meervoudige berekende waardes

```
scale n (x:xs) = (n * x) : (scale n xs)
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) = if x < y then
                          x : (merge xs (y:ys))
                      else
                          y : (merge (x:xs) ys)

seq2 = 1 : (merge (scale 2 seq2))
seq3 = merge seq2 (scale 3 seq3)
seq5 = merge seq3 (scale 5 seq5)
```

Figuur 4.2: Verbeterd Dijkstra's algoritme in Haskell.

geeft Kahn [3]. De code in figuur 4.2 laat dit zien. Hier worden er drie rijen gemaakt: seq2, seq3 en seq5 die respectievelijk de 2, 3 en 5-smooth getallen bevatten. Vanwege de unieke priemfactorisatie is gegarandeerd dat er geen waardes meerdere malen voorkomen in één van de lijsten.

4.2 Het alternatieve algoritme

Het uitgangspunt voor het alternatieve algoritme is de mogelijkheid een successorfunctie te bouwen die gegeven een willekeurig Hamminggetal, het eerstvolgende Hamminggetal retourneert. Gebruikmakend van een dergelijk functie, is het opsommen van de Hammingrij eenvoudig en weinig geheugenbelastend, mits de functie ook weinig geheugen gebruikt.

4.2.1 De successorfunctie

Hammingfactoren

Een Hammingfactor definiëren we als de vereenvoudigde deling van twee opeenvolgende Hamminggetallen. De successorfunctie zoekt, gegeven een input, de kleinste Hammingfactor die vermenigvuldigd met deze input een geheel getal oplevert. Dit getal is dan het eerstvolgende Hamminggetal.

Het aantal Hammingfactoren

In figuur 4.3 zien we dat er 22 Hammingfactoren voor de generatie van de eerste 1000 Hamminggetallen bestaan. Er zijn 51 Hammingfactoren voor de eerste miljoen Hamminggetallen. Hieruit volgt ons vermoeden dat het aantal Hammingfactoren klein is ten opzichte van het aantal Hamminggetallen. Om dit te bewijzen moeten we eerst een aantal eigenschappen van de Hammingfactoren bepalen.

Breuk	Gödel	Decimaal		Breuk	Gödel	Decimaal
$\frac{2}{1}$	[1, 0, 0]	2		$\frac{16875}{16384}$	[-14, 3, 4]	1.029968262
$\frac{3}{2}$	[-1, 1, 0]	1.5		$\frac{128}{125}$	[7, 0, -3]	1.024
$\frac{4}{3}$	[2, -1, 0]	1.333333333		$\frac{3125}{3072}$	[-10, -1, 5]	1.017252604
$\frac{5}{4}$	[-2, 0, 1]	1.25		$\frac{20000}{19683}$	[5, -9, 4]	1.016105269
$\frac{6}{5}$	[1, 1, -1]	1.2		$\frac{81}{80}$	[-4, 4, -1]	1.0125
$\frac{9}{8}$	[-3, 2, 0]	1.125		$\frac{2048}{2025}$	[11, -4, -2]	1.011358025
$\frac{10}{9}$	[1, -2, 1]	1.111111111		$\frac{78732}{78125}$	[2, 9, -7]	1.0077696
$\frac{27}{25}$	[0, 3, -2]	1.08		$\frac{393216}{390625}$	[17, 1, -8]	1.00663296
$\frac{16}{15}$	[4, -1, -1]	1.066666667		$\frac{15625}{15552}$	[-6, -5, 6]	1.00469393
$\frac{135}{128}$	[-7, 3, 1]	1.0546875		$\frac{1600000}{1594323}$	[9, -13, 5]	1.003560759
$\frac{25}{24}$	[-3, -1, 2]	1.041666667		$\frac{32805}{32768}$	[-15, 8, 1]	1.00112915

Figuur 4.3: Alle benodigde factoren voor de eerste 1000 Hamminggetallen ($H_{1000} = 51200000$).

Unieke Hammingfactor

Als de lijst van factoren beperkt is, moeten bepaalde factoren meermalen gebruikt worden. Daarom kunnen we spreken van een eerste keer dat een factor gebruikt wordt. Als gegeven $f = \frac{H_{i+1}}{H_i}$ en f is niet te vereenvoudigen, kunnen we concluderen dat f voor het eerst optreedt bij H_i en het volgende Hamminggetal H_{i+1} is. Verder weten we dan dat H_i en H_{i+1} relatief priem zijn.

Het aantal Hammingfactoren is dus even groot als het aantal opeenvolgende Hamminggetallen die onderling relatief priem zijn.

Type	H_i	H_{i+1}
1	$2^a \cdot 3^0 \cdot 5^0$	$2^0 \cdot 3^b \cdot 5^c$
2	$2^0 \cdot 3^b \cdot 5^c$	$2^a \cdot 3^0 \cdot 5^0$
3	$2^0 \cdot 3^b \cdot 5^0$	$2^a \cdot 3^0 \cdot 5^c$
4	$2^a \cdot 3^0 \cdot 5^c$	$2^0 \cdot 3^b \cdot 5^0$
5	$2^0 \cdot 3^0 \cdot 5^c$	$2^a \cdot 3^b \cdot 5^0$
6	$2^a \cdot 3^b \cdot 5^0$	$2^0 \cdot 3^0 \cdot 5^c$

Figuur 4.4: Mogelijke plekken waar een nieuwe factor kan ontstaan.

Het aantal opeenvolgende Hamminggetallen dat relatief priem is

In figuur 4.4 staan de zes mogelijke type paren die relatief priem zijn. Tussen twee Hamminggetallen h en $2 \cdot h$ is er maximaal één Hamminggetal van de vorm $2^a \cdot 3^0 \cdot 5^0$, één Hamminggetal van de vorm $2^0 \cdot 3^b \cdot 5^0$ en één Hamminggetal van de vorm $2^0 \cdot 3^0 \cdot 5^c$. Hierdoor kan er tussen h en $2 \cdot h$ maximaal één factor van elk type zijn. Het aantal factoren groeit dus linear bij een exponentiële groei van de Hamminggetallen. Dus groeit het aantal factoren logaritmisch ten opzichte van de Hamminggetallen. Omdat het aantal Hamminggetallen met een derde macht van het logarithme groeit, zie appendix A.2, is de conclusie dat het aantal factoren *maximaal* groeit met een derdemachtswortel ten opzichte van het aantal Hamminggetallen.

4.2.2 Het genereren van de lijst met factoren

Tot nu toe bepalen we de Hammingfactor aan de hand van twee opeenvolgende Hamminggetallen. Maar als we de successorfunctie willen gebruiken om deze getallen te genereren, kunnen we niet de factoren bepalen met behulp van Hamminggetallen. Om een alternatief te vinden moeten we eerst meer kenmerken van deze Hammingfactoren bepalen.

De initiële lijst

Voor het creëren van nieuwe Hammingfactoren hebben we een kleine initële lijst van Hammingfactoren nodig. Laat er een lijst F_5 zijn waarin alle Hammingfactoren benodigd om alle Hamminggetallen tot en met 5 te genereren. De lijst kan bijvoorbeeld gegenereerd worden door Dijkstra's algoritme te gebruiken.[1] De lijst bestaat dan uit:

$$F_5 ::= \left\{ \frac{2}{1}, \frac{3}{2}, \frac{4}{3}, \frac{5}{4} \right\} \tag{4.1}$$

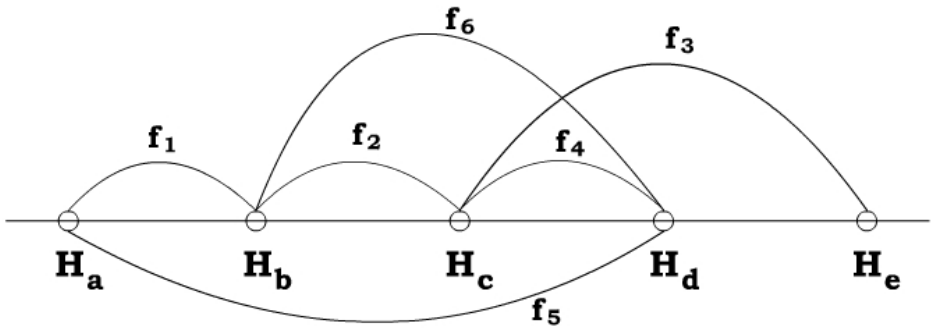
We weten dat elk Hamminggetal is te schrijven als een product van 2, 3 en 5. Omdat geldt:

$$2 = \frac{2}{1}, \quad 3 = \frac{2}{1} \cdot \frac{3}{2}, \quad 5 = \frac{2}{1} \cdot \frac{3}{2} \cdot \frac{4}{3} \cdot \frac{5}{4} \tag{4.2}$$

kunnen we dus, door substitutie, elk Hamminggetal schijven als een product van Hammingfactoren uit F_5:

$$\forall_{h \in H} (h = \prod_{i=0}^{q} a_i \text{Waarbij voor alle } a_i \in F_5) \tag{4.3}$$

[1] We zullen later zien dat er ook een andere manier is om een goede initiële lijst te krijgen.



Figuur 4.5: Het bepalen van nieuwe factoren.

Stel we hebben een lijst F van alle Hammingfactoren gebruikt voor het genereren van alle Hamminggetallen tot H_c, zie figuur 4.5.

$$F = \{f_1, f_2, f_3, f_5, \ldots\} \tag{4.4}$$

Als f_3 de kleinste factor uit F is die toegepast kan worden op H_c en er is een Hamminggetal H_d is waarvoor geldt: $H_c < H_d < H_e$, dan weten we dat f_4 een nieuwe Hammingfactor is. Omdat H_d te schrijven is als een product van Hammingfactoren uit F is er ook een Hamminggetal dat vermenigvuldigd met één factor uit F, H_d oplevert. In het figuur is dat H_a. Hierbij hebben we H_a zo gekozen dat dit het grootste getal is waarvoor er één Hammingfactor is die onderling vermenigvuldigd H_d oplevert.

We kunnen nu $f_6 = \frac{f_5}{f_1}$ berekenen. f_6 kan geen Hammingfactor zijn, omdat H_a het grootste getal was dat met één vermenigvuldiging met een Hammingfactor uit F H_d opleverde. Als we de verkregen factor nogmaals delen door een Hammingfactor uit F krijgen we: $f_4 = \frac{f_6}{f_2}$. Omdat er tussen h_d en H_a maar een eindig aantal Hamminggetallen ligt, is er een eindig aantal delingen nodig om f_4 te vinden.

Daarmee geldt dus dat gegeven een lijst van Hammingfactoren gebruikt tot een bepaalde Hamminggetal, elk nieuwe Hammingfactor te bepalen is door een eindig aantal delingen van deze factoren.

Zinnige factoren

Omdat elk Hamminggetal vermenigvuldigd kan worden met $\frac{2}{1}$ en omdat een factor altijd een groter Hamminggetal oplevert weten we dat voor alle Hammingfactoren geldt: $1 < f_i \leq 2$. Factoren die niet op het interval $(1, 2]$ liggen zijn daarmee niet zinnig.

Uitgezonderd van de beginset van factoren, zijn alle factoren een deling van twee andere factoren. Maar zijn alle delingen van factoren ook nieuwe Hammingfactoren? Dit blijkt niet het geval te zijn. $\frac{2}{1}$ en $\frac{9}{8}$ zijn beide

factoren, maar $\frac{16}{9}$ is dat niet. Dat komt omdat er ook een factor is $\frac{10}{9}$, die toepasbaar is in alle gevallen dat $\frac{16}{9}$ dat ook is, en altijd een kleiner Hamminggetal oplevert. Hierbij introduceren we een nieuw begrip genaamd beter:

$$f_1 \text{is beter dan} f_2 ::= 1 < f_1 < f_2 \wedge \forall_{h \in H_{\mathbb{N}}}(f_2 \cdot h \in \mathbb{N} \Rightarrow f_1 \cdot h \in \mathbb{N}) \quad (4.5)$$

Het kan zijn dat een nieuwe factor beter is dan een al bestaande factor. Voor alle Hammingfactoren geldt dat er geen andere factor is die beter is.

Het algoritme

We hebben een lijst met Hammingfactoren F en slaan alle onderlinge delingen die zinnige factoren opleveren, op in een lijst P van potentiële Hammingfactoren. We nemen uit P de factor met de kleinste noemer, omdat alle andere factoren pas voor het eerst gebruikt kunnen worden bij een groter Hamminggetal. We voegen daarna alle zinnige delingen van deze factor met alle factoren uit F toe aan P. Als alle nieuwe factoren een grotere noemer hebben dan de door ons geselecteerde, is er blijkbaar geen betere factor gevonden met deze delingen, en weten we dat elke volgende deling ook niet tot een betere factor zal leiden. Tot slot voegen we de nieuwe factor toe aan F. waardoor F dan alle Hammingfactoren bevat benodigd voor het creëren van alle Hamminggetallen tot de waarde van de deler van de nieuwe factor.

Als we F en P als volgt initiëren: $F = \{\frac{2}{1}\}$ en $P = \{\frac{3}{2}, \frac{5}{3}\}$, dan blijkt dat dit voldoende is om alle Hammingfactoren te kunnen genereren.

Als F oplopend gesorteerd is op grootte, en er een Hamminggetal h gegeven wordt, dan is de eerste factor f uit F waarvoor geldt $f \cdot h \in \mathbb{N}$ de factor die het eerstvolgende Hamminggetal na h oplevert. Als echter geldt dat h groter is dan de grootste deler van alle factoren uit F, moet er eerst nieuwe Hammingfactoren berekend worden.

4.2.3 De ordening van de lijst met Gödelgecodeerde factoren.

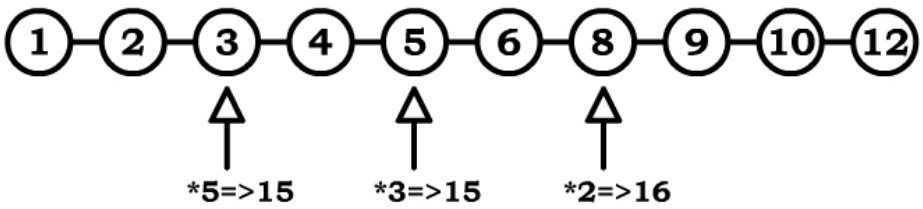
Als we gebruik maken van Gödelcodering om het geheugengebruik te minimaliseren, hebben we het probleem dat de lijst met factoren een geordende lijst moet zijn. Bovendien moeten we in het bovenbeschreven proces de factor isoleren met de kleinste noemer. Om deze reden wordt bij elke factor naast de Gödelcodering ook de decimale waarden van de teller en noemer opgeslagen.

Hoofdstuk 5

Java implementaties

De voornoemde algoritmen zijn geïmplementeerd in Java. Teneinde de tijd- en geheugencomplexiteit van de algoritmen te bepalen worden alle berekeningen op de Hamminggetallen door een aparte class geregistreerd. Dit is nodig omdat een meting van de benodigde tijd beïnvloed kan worden als een processor aan multitasking doet en door de tijd benodigd door Java's virtuele machine (VM). Omdat Java een automatische garbage collector heeft zou dit ook de meting kunnen beïnvloeden.

5.1 Dijkstra's algoritme



Figuur 5.1: Dijkstra's algoritme als een gelinkte lijst.

Het voorbeeld van Dijkstra's algoritme gegeven in figuur 4.1 is geschreven in Haskell. Om een gelijkwaardige implementatie te krijgen in Java, dat wil zeggen gelijkwaardig in tijd- en geheugengebruik, is gekozen om de lijst te genereren in de vorm van een gelinkte lijst. Langs deze lijst lopen drie pointers die de waarde die ze op dat moment aanwijzen vermenigvuldigen met respectievelijk 2, 3 of 5, zie figuur 5.1. De pointers met de laagste waarde levert het volgende Hamminggetal op. Dit getal wordt achteraan de

lijst toegevoegd en deze pointers worden één positite opgeschoven.

5.2 Kahn's algoritme

Het verbeterde algoritme uit figuur 4.2 is identiek aan de voorgaande implementatie. Het enige verschil is dat elk item uit de gelinkte lijst ook bevat of dit item is ontstaan uit een vermenigvuldiging met 2, 3 of 5. Bij het opschuiven van de respectievelijke pointers, stoppen deze bij het eerstvolgende element dat gegenereerd was door deze pointer, of een pointer met een kleinere factor.

5.2.1 Het geheugengebruik

De implementatie van de berekening het geheugengebruik kan op twee manieren geïmplementeerd worden. Naief kan er altijd een pointer naar het eerste item in de lijst wijzen. Bij het opvragen van het gebruikte geheugen itereer je over alle elementen in de gelinkte lijst en telt het geheugengebruik van de individuele getallen bij elkaar op.

Om het geheugen minder te belasten is gekozen om alle items uit de gelinkte lijst die vóór de pointer die met 5 vermenigvuldigd staan, te laten garbagecollecten. Deze elementen worden namelijk niet gebruikt voor de verdere berekening van de Hamminggetallen. Voordat de 5-pointer één positie opschuift wordt er een memorycounter opgehoogd met het geheugengebruik van de waarde waar de pointer naar wijst. Hierdoor kan op elk moment het geheugengebruik opgevraagd worden benodigd om alle tot dan toe gegenereerde Hammingetallen in op te slaan.

5.3 De gebruikte getalcoderingen

Alle in Java geschreven algoritmen kunnen worden aangeroepen met een parameter die bepaald wat voor een type getalcodering gebruikt word. De twee gebruikte typen zijn een extensie op de Javaclass "BigInteger"genaamd "Number", en een Goedelcodering. Het geheugengebruik van "Number"wordt gegeven door de Java method "BitCount()". De Goedelcodering gebruikt voor de drie exponenten ook "Number" en het geheugengebruik is de som van de bitcounts van die drie exponenten. Als voor een Gödelgecodeerd getal ook de BigInteger wordt uitgerekend, wordt deze bij het getal opgeslagen en de bitcount toegevoegd aan het totaal benodigd geheugen voor dat getal.

Bij elke operatie op "Number": optellen, aftrekken, delen, vermenigvuldigen en vergelijken, wordt in een statische class opgeslagen hoeveel bitoperaties er voor nodig zijn.

5.4 Echt processor- en geheugengebruik

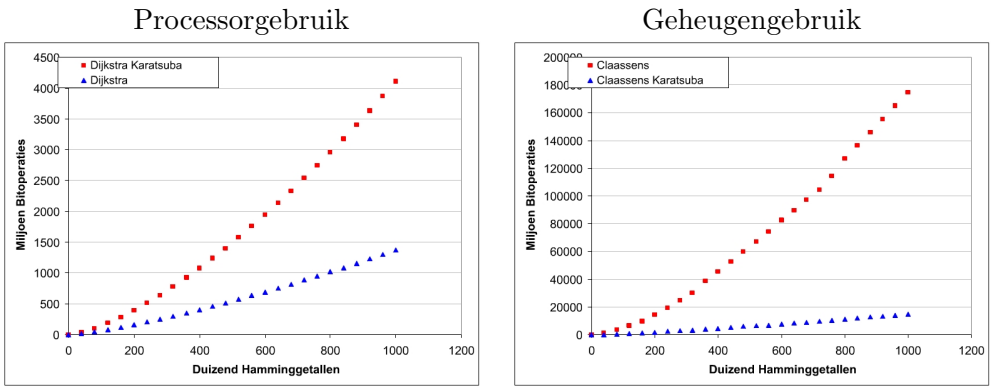
Om een idee te krijgen of hoe snel de implementaties zijn zonder overhead van complexiteitsbepaling, en met native Java datastructuren hebben we ook een implementatie van het verbeterde Dijkstra algoritme en ons eigen algoritme met Gödelcodering met Longs getest op tijd- en geheugengebruik.

Hoofdstuk 6

De resultaten van de tests

Als eerste hebben we gecontroleerd of alle geïmplementeerde algoritmen ook daadwerkelijk de correcte Hammingrij genereerden. Voor de eerste miljoen Hamminggetallen gaven de algoritmen de correcte Hamminggetallen.

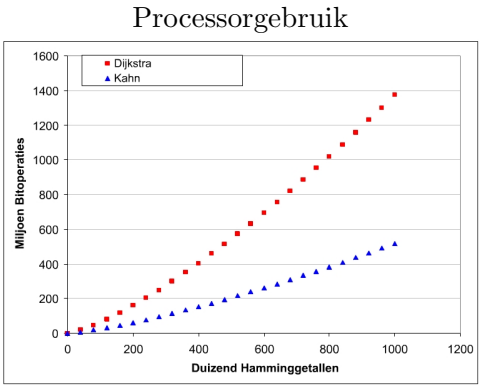
6.1 Karatsuba bij de twee algoritmen



Figuur 6.1: Beide algoritmen met en zonder Karatsuba.

In 6.1 zien we dat er een slechtere tijdcomplexiteit is bij Dijkstra's algoritme als Karatsuba gebruikt wordt. Dit komt omdat Karatsuba voordeel heeft bij vermenigvuldiging van twee getallen van ongeveer gelijke grootte. Dijkstra vermenigvuldigd ook grote Hamminggetallen slechts met 2, 3 en 5.

Bij ons algoritme wordt een gegeven getal vermenigvuldigd met een factor, een teller en een noemer. Deze getallen zijn veel groter dan 5, waardoor Karatsuba hier wel voordeel heeft.



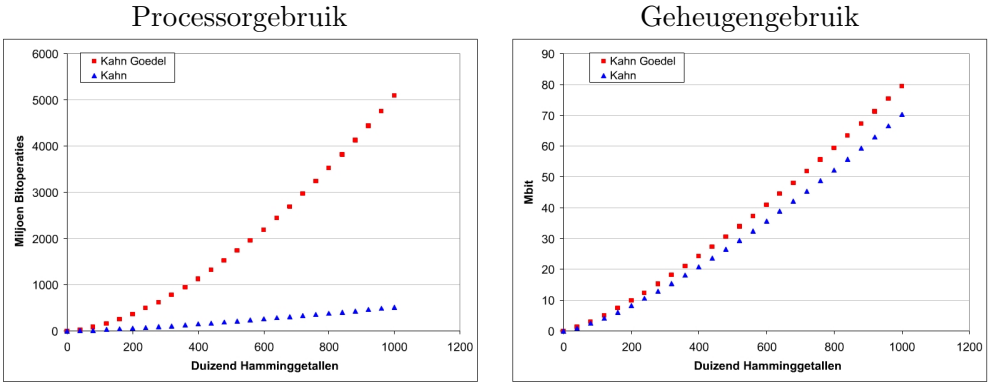
Figuur 6.2: Dijkstra's algoritme versus Kahn's algoritme.

6.2 Dijkstra's algoritme

In 4.1 gaven we al de mogelijkheid aan om het traditionele algoritme van Dijkstra te verbeteren. De beide implementaties laten een groot verschil in processorgebruik zien, zie figuur 6.2. Het viel te verwachten dat er een verbetering van ongeveer een factor 3 zou zijn, omdat bijna elk Hamminggetal drie maal berekend zou worden. Naarmate de getallen groter worden komt het verschil duidelijker in de buurt van een factor 3.

Het geheugengebruik tussen beide implementatie laat geen verschil zien, wat ook precies volgens onze verwachting was.

6.3 Gödelcodering bij Dijkstra's algoritme

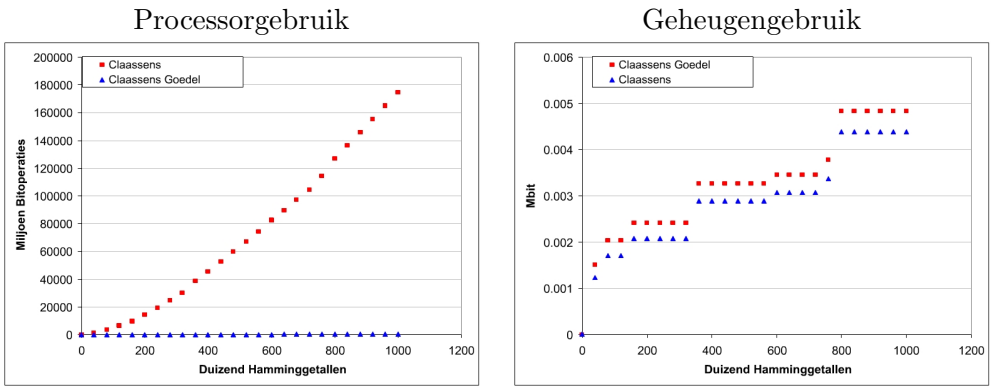


Figuur 6.3: Dijkstra's algoritme met en zonder Gödelcodering.

Figuur 6.3 laat duidelijk zien dat het gebruiken van Gödelcodering bij Dijkstra's algoritme zowel voor de snelheid als ook het geheugengebruik geen enkel positief effect heeft. Dat komt omdat het algoritme voor elk nieuw

Hamminggetal het kleinste kiest uit drie opties. Omdat het vergelijken van Gödelgecodeerde getallen niet kan, moet elk getal omgezet worden naar een binaire notatie. Daarmee gaat al het voordeel verloren. Het geheugen van een goedelgecodeerd getal bevat dan zowel de codering als de berekende waarde, en is daarmee dus iets minder efficiënt dan eenvoudigweg de waarde opslaan.

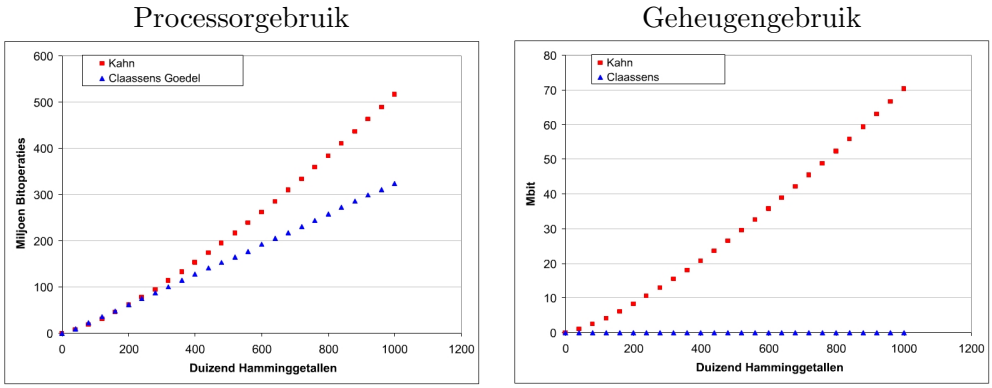
6.4 Gödelcodering bij ons algoritme



Figuur 6.4: Ons algoritme met en zonder Gödelcodering.

Figuur 6.4 laat zien dat het geheugengebruik met Gödelcodering iets slechter is dan zonder. Hiervoor geldt dezelfde redenering als bij Dijkstra's algoritme. Ook de lijst van factoren, wat alles is dat in het geheugen staat, is een geordende lijst. Het is overduidelijk dat de Gödelcodering een verbetering is als we de naar het aantal benodigde operaties kijken. Dit komt doordat alle Hamminggetallen als Gödelcodering worden opgelevert, wat slechts optellingen van kleine getallen vergt in plaats van vermenigvuldiging van grote getallen.

Figuur 6.5 laat zien dat ons algoritme op zowel het aantal operaties als ook het geheugengebruik beter presteert dan Dijkstra's. Het betere geheugengebruik is volledig te danken aan het feit dat we niet alle Hamminggetallen in het geheugen hebben, maar slechts alle benodigde factoren. De verbetering in het processorgebruikt is te danken aan het gebruik van Gödelcodering, waardoor kostbare vermenigvuldigingen omgezet zijn naar optellingen. Het voordeel komt pas echt tot uiting bij zeer grote getallen. Het lijkt dat bij ons algoritme de tijdcomplexiteit lineair is ten opzicht van het aantal Hamminggetallen, maar dit is niet het geval. Ook de exponenten in de Gödelcodering groeien, waardoor het optellen van grote Hamminggetallen méér kost dan van kleine getallen, waardoor er geen lineair verband kan zijn.

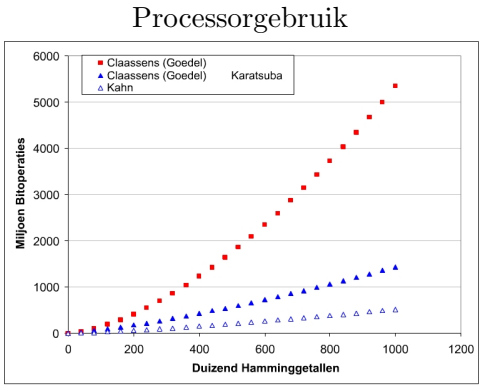


Figuur 6.5: Ons algoritme vergeleken met Dijkstra's.

6.4.1 Sprongen in de geheugencomplexiteit

Figuur 6.4 laat duidelijke sprongen zien. Deze zijn vooral duidelijke bij 400 en 800 (duizend). Dit komt omdat op die plekken respectievelijk de factoren $[-178, 146, -23]$ en $[2, 176, -121]$ toegevoegd worden. Een nadere inspectie van de tijdcomplexiteit op deze punten, laat ook een lichte extra toename in de complexiteit zien.

6.5 Kanttekening bij de verbetering in processorgebruik

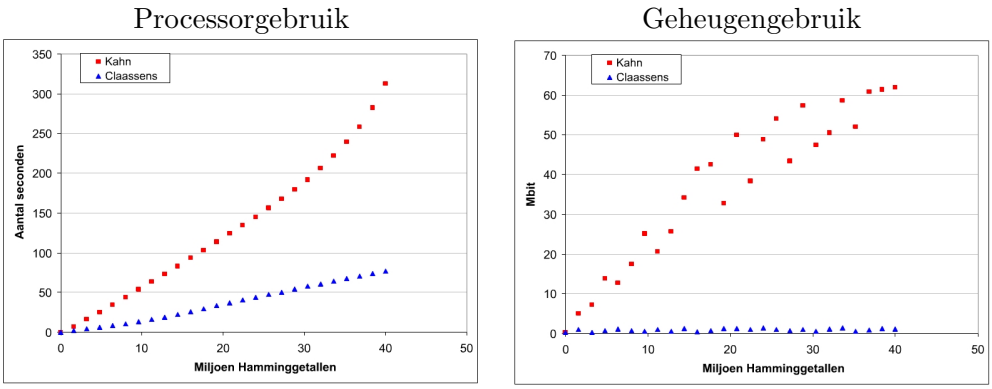


Figuur 6.6: Ons algoritme met decimale output vergeleken met Dijkstra's.

De twee algoritmes leveren wel twee verschillende outputs. Dijkstra's geeft een binaire/decimale lijst van Hamminggetallen, waar ons algoritme een Gödelgecodeerde lijst is. Als we ook een binaire lijst van Hamminggetallen willen opleveren moet elk getal ook 'echt' berekend worden. In figuur

6.6 is te zien dat Kahn's algoritme een veel betere tijdcomplexiteit heeft dan ons algoritme. Zelfs het gebruikt van Karatsuba levert te weinig voordeel op ten opzicht van Kahn.

6.6 Implementatie met native datatypes



Figuur 6.7: Ons algoritme met decimale output vergeleken met Dijkstra's.

We hebben ook de implementaties met native datatypen met elkaar vergeleken om te zien of de berekende complexiteiten overeenkomen in de gemeten snelheid en het geheugengebruik. Figuur 6.7 zien we dat de snelheid van onze implementatie beduidend beter is dan Kahn's algoritme. Boven de 30 miljoen Hamminggetallen neemt de snelheid van Dijkstra snel af. Dit is te verklaren dat dan het beschikbare geheugen bijna volledig gebruikt wordt, waardoor Java's VM veel tijd kwijt is om het geheugenbeheer te doen.

Als we naar het geheugengebruik kijken is het ook duidelijk dat ons algoritme hier beter presteert. Wel valt op dat het geheugengebruik sterk fluktureert en dat het dus geen directe indicatie is van de grootte van de gebruikte datastructuren.

Hoofdstuk 7

Conclusies

Bepaalde talen zijn misschien wel handig in het snel schrijven van een implementatie, maar leveren zeker niet altijd efficiënte code op. Een goed doordacht algoritme kan tot significante verbetering in efficiëntie leiden. Hierbij blijft er ook nog een afweging betreffende de leesbaarheid van de code. We hebben aangetoont dat met slechts kleine verandering aan een bestaand algoritme duidelijke verbetering optreedt, zonder dat de code onduidelijk wordt.

Het algoritme dat wij geïntroduceerd hebben heeft een verbetering in de efficiëntie maar dit gaat duidelijk ten koste van de leesbaarheid van het algoritme. In onze ogen zou het wenselijk zijn om de mogelijkheid te hebben om twee verschillende implementaties geautomatiseerd te kunnen vergelijken. Dan zou Dijkstra's originele algoritme gebruikt kunnen worden als een werkende specificatie, en ons algoritme gechecked kunnen of het voldoet aan de specificatie.

Bibliografie

[1] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall inc., 1976.

[2] W.H.J. Feijen Edsger W. Dijkstra. *Ik ben nog op zoek naar een betere verwijzing, maar nog niet gevonden.A Method of Programming.* Addison-Wesley, 1988.

[3] Kahn G. and MacQueen D.B. Coroutines and networks of parallel processes. In *Proc. IFIP Congress 77*, pages 993–998, 1977.

[4] Luis Carlos Coronado Garca. Can schönhage multiplication speed up the rsa encryption or decryption? Presentation, 2005. Department of Computer Science University of Technology, Darmstadt.

[5] A. Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. *Proceedings of the USSR Academy of Sciences*, 145(145):293–294, 1962.

[6] Carl Pomerance Richard Crandall. *Prime numbers : a computational perspective.* Springer, 2001.

[7] A. Schönhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(7):281–292, 1971.

[8] D. van Leijenhorst. *T2-diktaat.* Radboud Universiteit Nijmegen, 2003.

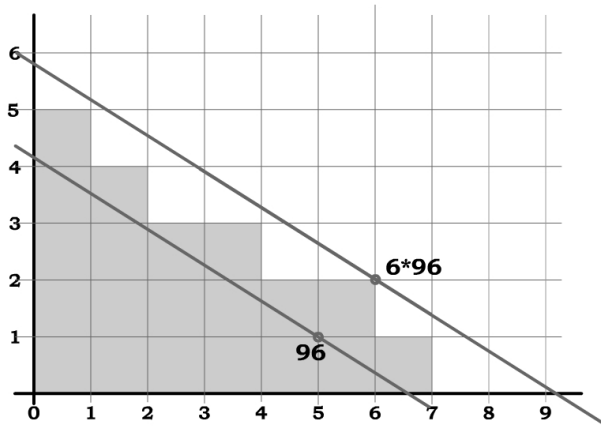
Bijlage A

Het bepalen van een schattingsfunctie voor 3-smooth en Hamminggetallen

A.1 3-smooth getallen

We weten dat $n = 2^{\,{}^2\log(n)} 3^0 = 2^0 3^{\,{}^3\log(n)}$. Als we in een coördinatenstelsel aan een punt de volgende betekenis geven: $(x, y) = 2^x 3^y$, dan is de rechte door de punten $({}^2\log(n), 0)$ en $(0, {}^3\log(n))$ de verzameling van alle coördinaten met een gelijke waarde.

Als we snel willen afschatten wat de index i is van een gegeven 3-smooth getal n, komt dat overeen met het aantal punten onder en op de driehoek van de lijn n en de x- en y-as in het coördinatenstelsel.



Figuur A.1: Coördinatenstelsel met de lijn voor 96 en $6 \cdot 96$.

In figuur A.1 is een dergelijk coördinatenstelsel gegeven met daar in de lijn voor $96 = 2^5 3^1 = 2^{\,^2\log(96)} 3^0 = 2^0 3^{\,^3\log(96)}$. Dus de lijn die door de punten $(\,^2\log(96), 0)$, $(5, 1)$ en $(0,\,^3\log(96))$ gaat is de lijn die alle punten bevat die corresponderen met 96.

De index van het 3-smooth-getal 96 is gelijk aan het aantal punten dat onder of op de lijn liggen, inclusief de punten op de beide assen. In figuur A.1 is voor al die punten het vlak rechtsboven dat punt grijs gekleurd. Voor bijvoorbeeld $(3, 2)$ is het vierkant tussen de punten $(3, 2), (4, 2), (3, 3), (4, 3)$ grijs gekleurd. Het totale oppervlak van deze grijze vierkanten is gelijk aan de gezochte index.

We weten nu dat het oppervlak van de driehoek gevormd door de lijn en beide assen $\frac{^2\log(96)\ ^3\log(96)}{2}$ een onderafschatting is van het oppervlak van de grijze vierkanten.

Omdat 96 zelf de grootste waarde is die een grijs vierkant bevat weten we dat de lijn door $6 \cdot 96 = 576$ zeker niet door een vierkant gaat dat grijs gekleurd is. Daarmee is het oppervlak onder die driehoek een bovenafschatting is van het totale oppervlak van alle grijze vierkanten.

Als we de lijn nemen die midden tussen deze twee lijnen ligt, dat is de lijn van $96\sqrt{6}$, is dit een redelijke afschatting van de index van het gegeven 3-smooth-getal.

$$I_{(n)} = \frac{^2\log(n\sqrt{6})\ ^3\log(n\sqrt{6})}{2} = \frac{(\ln(n\sqrt{6}))^2}{2\ln(2)\ln(3)} \tag{A.1}$$

Als we deze functie herschrijven kunnen we ook een afschatting geven van het 3-smooth-getal gegeven een index.

$$H_i = \frac{e^{\sqrt{2i\ln(2)\ln(3)}}}{\sqrt{6}} \tag{A.2}$$

A.2 Hamminggetallen

Voor de Hamminggetallen hebben we te maken met een vlak waarvoor de waarde gelijk blijft, in plaats van een lijn. Het vlak tussen de onderafschatting en de bovenafschatting ligt $\sqrt{30}$ van het gegeven getal. Daarmee wordt de functie:

$$I_{(n)} = \frac{^2\log(n\sqrt{30})\ ^3\log(n\sqrt{30})\ ^5\log(n\sqrt{30})}{6} = \frac{(\ln(n\sqrt{30}))^3}{2\ln(2)\ln(3)\ln(5)}$$

Waardoor de afschatting van de Hamminggetallen gelijk wordt aan:

$$H_i = \frac{e^{\sqrt[3]{6i\ln(2)\ln(3)\ln(5)}}}{\sqrt{30}} \tag{A.3}$$

Bijlage B

De sourcecode van de gebruikte implementaties

B.1 IComparable.java

```
package numbers;

public interface IComparable<T>
{
  public int compareTo(T _value);
}
```

B.2 IGenerator.java

```
package numbers.hamminggenerators;
import numbers.INumber;

public interface IGenerator<TNumber extends INumber>
{
  public TNumber next() throws Exception;
  public long getMemory();
}
```

B.3 INumber.java

```
package numbers;
import java.lang.Comparable;
import java.math.BigInteger;

public interface INumber<TNumber>
{
  public INumber value();
```

```
  public TNumber add(TNumber _value);
  public TNumber subtract(TNumber _value);
  public TNumber multiply(TNumber _value);
  public TNumber divide(TNumber _value);
  public TNumber gcd(TNumber _value);
  public TNumber mod(TNumber _value);
  public boolean dividable(TNumber _value);
  public int compare(TNumber _value);
  public boolean equal(TNumber _value);
  public BigInteger getValue();
  public long memoryUse();
}
```

B.4 INumberGenerator.java

```
package numbers.hamminggenerators.numbergenerators;
import numbers.INumber;

public interface INumberGenerator<TNumber extends INumber<TNumber>>
{
  public TNumber create(String _input);
}
```

B.5 BitOperatorCounters.java

```
package numbers.efficiency;

public class BitOperatorCounters
{
  public long min = 0;
  public long max = 0;
  public long best = 0;
  public long counter = 0;
}
```

B.6 Claassens.java

```
package numbers.hamminggenerators.claassens;
import numbers.hamminggenerators.numbergenerators.implementation.NumberGenerator;
import numbers.implementation.Number;
import numbers.hamminggenerators.numbergenerators.INumberGenerator;
import numbers.INumber;
import test.Test;
import numbers.hamminggenerators.IGenerator;
import java.util.TreeSet;
import java.util.ArrayList;

public class Claassens<TNumber extends INumber<TNumber>>
  implements
```

```
    IGenerator<TNumber>
{
  private TreeSet<Factor<TNumber>> factors;
  private ArrayList<Factor<TNumber>> potentials;
  private TNumber getNextPotential;
  private TNumber current;
  private boolean first;

  public Claassens(INumberGenerator<TNumber> _generator) {
    factors = new TreeSet<Factor<TNumber>>();
    potentials = new ArrayList<Factor<TNumber>>();
    getNextPotential = _generator.create("1");

    factors.add(new Factor<TNumber>(_generator.create("2"), _generator.create("1")));
    potentials.add(new Factor<TNumber>(_generator.create("3"), _generator.create("2")));
    potentials.add(new Factor<TNumber>(_generator.create("5"), _generator.create("3")));

    current = _generator.create("1");
    first=true;
  }

  public TNumber next() throws Exception {
    if (first) {
      first = false;
      return current;
    }
    current = next(current);
    return current;
  }
  public TNumber next(TNumber _value) throws Exception {
    while (_value.equal(getNextPotential)) {
      nextPotential();
    }

    for(Factor<TNumber> factor:factors) {
      if (factor.isApplicable(_value)) {
        return _value.divide(factor.denominator).multiply(factor.numerator);
      }
    }

    return null;
  }

  public void nextPotential() {
    while (true) {
      Factor<TNumber> bestFactor = potentials.get(0);
      for(Factor<TNumber> factor:potentials) {
        int comp = factor.denominator.compare(bestFactor.denominator);
        if (comp<0 || (comp==0 && factor.numerator.compare(bestFactor.numerator)<0)) {
          bestFactor = factor;
        }
      }

      potentials.remove(bestFactor);
```

```
    for(Factor<TNumber> factor:factors) {
      if (!bestFactor.equalTo(factor)) {
        Factor<TNumber> newFactor = factor.divide(bestFactor);
        if (!newFactor.isBiggerThan1()) {
          newFactor = newFactor.inverse();
        }
        addPotential(newFactor);
      }
    }


    if (addFactor(bestFactor)) {
      getNextPotential = bestFactor.denominator;
      return;
    }
  }
}

public boolean addPotential(Factor<TNumber> _newFactor) {
  if (_newFactor.denominator.compare(_newFactor.numerator)==0) {
    return false;
  }
  for(Factor<TNumber> factor:factors) {
    if (factor.isBetterThan(_newFactor) || factor.equalTo(_newFactor)) {
      return false;
    }
  }
  for(Factor<TNumber> factor:potentials) {
    if (factor.isBetterThan(_newFactor) || factor.equalTo(_newFactor)) {
      return false;
    }
  }
  potentials.add(_newFactor);
  return true;
}
public boolean addFactor(Factor<TNumber> _newFactor) {
  for(Factor<TNumber> factor:factors) {
    if (factor.isBetterThan(_newFactor) || factor.equalTo(_newFactor)) {
      return false;
    }
  }
  factors.add(_newFactor);
  return true;
}
 public long getMemory() {
  return getMemory(factors) + getMemory(potentials);
}
 public long getMemory(Iterable<Factor<TNumber>> _iteratable) {
   long result =0;

   for(Factor<TNumber> factor :_iteratable) {
     result += factor.memoryUse();
   }
```

```
    return result;
  }
}
```

B.7 DijkstraImproved.java

```
package numbers.hamminggenerators.dijkstra;
import numbers.hamminggenerators.numbergenerators.implementation.NumberGenerator;
import numbers.implementation.Number;
import numbers.hamminggenerators.numbergenerators.INumberGenerator;
import numbers.INumber;
import test.Test;
import numbers.hamminggenerators.IGenerator;
import java.util.concurrent.LinkedBlockingQueue;

public class DijkstraImproved<TNumber extends INumber<TNumber>>
  implements
    IGenerator<TNumber>
{
  private LinkedListItemImproved<TNumber> endOfList;
  private LinkedListItemPointerImproved<TNumber> p2;
  private LinkedListItemPointerImproved<TNumber> p3;
  private LinkedListItemPointerImproved<TNumber> p5;
  private long memoryCounter = 0;

  public DijkstraImproved(INumberGenerator<TNumber> _generator) {
    endOfList = new LinkedListItemImproved<TNumber>(_generator.create("1"),2);
    p2 = new LinkedListItemPointerImproved<TNumber>(_generator.create("2"), endOfList,2);
    p3 = new LinkedListItemPointerImproved<TNumber>(_generator.create("3"), endOfList,3);
    p5 = new LinkedListItemPointerImproved<TNumber>(_generator.create("5"), endOfList,5);
  }

  public TNumber next() {
    TNumber result = endOfList.value;
    if (p2.nextValue.compare(p3.nextValue)<0) {
      if (p2.nextValue.compare(p5.nextValue)<0) {
        endOfList.next = new LinkedListItemImproved<TNumber>(p2.nextValue,2);
        endOfList = endOfList.next;
        p2.moveNext();
      } else {
        endOfList.next = new LinkedListItemImproved<TNumber>(p5.nextValue,5);
        endOfList = endOfList.next;
        memoryCounter += p5.item.value.memoryUse();
        p5.moveNext();
      }
    } else {
      if (p3.nextValue.compare(p5.nextValue)<0) {
        endOfList.next = new LinkedListItemImproved<TNumber>(p3.nextValue,3);
        endOfList = endOfList.next;
        p3.moveNext();
```

```
      } else {
        endOfList.next = new LinkedListItemImproved<TNumber>(p5.nextValue,5);
        endOfList = endOfList.next;
        memoryCounter += p5.item.value.memoryUse();
        p5.moveNext();
      }
    }

    return result;
  }
  public long getMemory() {
    long result = 0;
    LinkedListItemImproved<TNumber> current = p5.item;

    while (current!=null) {
      result += current.value.memoryUse();
      current = current.next;
    }
    return result + memoryCounter;
  }
}
```

B.8 DijkstraLinkedList.java

```
package numbers.hamminggenerators.dijkstra;
import numbers.hamminggenerators.numbergenerators.implementation.NumberGenerator;
import numbers.implementation.Number;
import numbers.hamminggenerators.numbergenerators.INumberGenerator;
import numbers.INumber;
import test.Test;
import numbers.hamminggenerators.IGenerator;

public class DijkstraLinkedList<TNumber extends INumber<TNumber>>
  implements
    IGenerator<TNumber>
{
  protected LinkedListItem<TNumber> endOfList;
  private LinkedListItemPointer<TNumber> p2;
  private LinkedListItemPointer<TNumber> p3;
  private LinkedListItemPointer<TNumber> p5;
  private long memoryCounter = 0;

  public DijkstraLinkedList(INumberGenerator<TNumber> _generator) {
    endOfList = new LinkedListItem<TNumber>(_generator.create("1"));
    p2 = new LinkedListItemPointer<TNumber>(_generator.create("2"), endOfList);
    p3 = new LinkedListItemPointer<TNumber>(_generator.create("3"), endOfList);
    p5 = new LinkedListItemPointer<TNumber>(_generator.create("5"), endOfList);
  }

  public TNumber next() {
    TNumber result = endOfList.value;
```

```
    int comp23 = p2.nextValue.compare(p3.nextValue);

    if (comp23<0) {
      int comp25 = p2.nextValue.compare(p5.nextValue);
      if (comp25<0) {
        endOfList.next = new LinkedListItem<TNumber>(p2.nextValue);
        endOfList = endOfList.next;
        p2.moveNext();
      } else {
        endOfList.next = new LinkedListItem<TNumber>(p5.nextValue);
        endOfList = endOfList.next;
        if (comp25==0) {
          p2.moveNext();
        }
        memoryCounter += p5.item.value.memoryUse();
        p5.moveNext();
      }
    } else {
      int comp35 = p3.nextValue.compare(p5.nextValue);
      if (comp35<0) {
        endOfList.next = new LinkedListItem<TNumber>(p3.nextValue);
        endOfList = endOfList.next;
        if (comp23==0) {
          p2.moveNext();
        }
        p3.moveNext();
      } else {
        endOfList.next = new LinkedListItem<TNumber>(p5.nextValue);
        endOfList = endOfList.next;
        if (p2.nextValue.compare(p5.nextValue)==0) {
          p2.moveNext();
        }
        if (comp35==0) {
          p3.moveNext();
        }
        memoryCounter += p5.item.value.memoryUse();
        p5.moveNext();
      }
    }

    return result;
  }
  public long getMemory() {
    long result = 0;
    LinkedListItem current = p5.item;
    while (current!=null) {
      result += current.value.memoryUse();
      current = current.next;
    }

    return result;// + memoryCounter;
  }
}
```

B.9 DijkstraLinkedListFull.java

```
package numbers.hamminggenerators.dijkstra;
import numbers.hamminggenerators.numbergenerators.implementation.NumberGenerator;
import numbers.implementation.Number;
import numbers.hamminggenerators.numbergenerators.INumberGenerator;
import numbers.hamminggenerators.dijkstra.LinkedListItem;
import numbers.INumber;
import test.Test;
import numbers.hamminggenerators.IGenerator;

public class DijkstraLinkedListFull<TNumber extends INumber<TNumber>>
  extends
    DijkstraLinkedList<TNumber>
{
  private LinkedListItem<TNumber> root;

  public DijkstraLinkedListFull(INumberGenerator<TNumber> _generator) {
    super(_generator);
    root = endOfList;
  }

  public long getMemory() {
    long result = 0;
    LinkedListItem current = root;
    while (current!=null) {
      result += current.value.memoryUse();
      current = current.next;
    }

    return result;
  }
}
```

B.10 EfficiencyRegistrator.java

```
package numbers.efficiency;
import java.math.*;

public class EfficiencyRegistrator
{
  private BitOperatorCounters addition = new BitOperatorCounters();
  private BitOperatorCounters multiplication = new BitOperatorCounters();
  private BitOperatorCounters comparing = new BitOperatorCounters();

  public void add(long _bitSize1, long _bitSize2) {
    addition.min += addition(Math.max(_bitSize1, _bitSize2));
    addition.max += addition(Math.max(_bitSize1, _bitSize2));
```

```
    addition.best += addition(Math.max(_bitSize1, _bitSize2));
    addition.counter++;
  }
  public void multiply(long _bitSize1, long _bitSize2) {
    multiplication.min  += _bitSize1* _bitSize2;
    multiplication.max  += 5 * Math.min(_bitSize1, _bitSize2) - 3*Math.max(_bitSize1, _bitSize2);
    multiplication.best += bestMultiplication(Math.max(_bitSize1, _bitSize2));
    multiplication.counter++;
  }
  public void compare(long _bitSize1, long _bitSize2) {
    comparing.min += addition(Math.max(_bitSize1, _bitSize2));
    comparing.max += addition(Math.max(_bitSize1, _bitSize2));
    comparing.best += addition(Math.max(_bitSize1, _bitSize2));
    comparing.counter++;
  }


  private long addition(long _bitSize) {
    return 5*_bitSize -3;
  }

  private long multiplication(long _bitSize) {
    return _bitSize * _bitSize;
  }
  private long bestMultiplication(long _bitSize) {
    return new Double(Math.pow(_bitSize,1.58496250072)).longValue();
  }

  public String toString() {
    String result =  "" + (addition.min + multiplication.min + comparing.min) + " " +
                     (addition.best + multiplication.best + comparing.best)+ " " +
                     (addition.max + multiplication.max + comparing.max) ;
    return result;
  }
}
```

B.11 Factor.java

```
package numbers.hamminggenerators.claassens;
import numbers.INumber;
import numbers.implementation.Number;

import java.lang.Comparable;

public class Factor<TNumber extends INumber<TNumber>>
  implements
    Comparable<Factor<TNumber>>
{
  public TNumber numerator;
  public TNumber denominator;

  protected void init(TNumber _numerator, TNumber _denominator) {
```

```
  TNumber gcd = _numerator.gcd(_denominator);
  numerator = _numerator.divide(gcd);
  denominator = _denominator.divide(gcd);
}
public Factor(TNumber _numerator, TNumber _denominator) {
  init( _numerator,  _denominator);
}

public TNumber multiply(TNumber _value) {
  if (_value.dividable(denominator)) {
    return numerator.multiply(_value).divide(denominator);
  } else {
    return null;
  }
}

public boolean isApplicable(TNumber _value) {
  return _value.dividable(denominator);
}

public Factor<TNumber> divide(Factor<TNumber> _factor) {
  return new Factor<TNumber>(numerator.multiply(_factor.denominator),
                             denominator.multiply(_factor.numerator));
}

public boolean isBiggerThan(Factor<TNumber> _factor) {
  return divide(_factor).isBiggerThan1();
}

public boolean isBetterThan(Factor<TNumber> _factor) {
  return _factor.denominator.dividable(denominator) && _factor.isBiggerThan(this);
}

public String toString() {
  String result = numerator + "/" + denominator;
  return result;
}

public boolean equalTo(Factor<TNumber> _factor) {
  return numerator.compare(_factor.numerator)==0 && denominator.compare(_factor.denominator)==0;
}

public boolean isBiggerThan1() {
  return numerator.compare(denominator)>0;
}

public Factor<TNumber> inverse() {
  return new Factor<TNumber>(denominator, numerator);
}

public int compareTo(Factor<TNumber> _factor) {
  Factor<TNumber> result = divide(_factor);
  return result.numerator.compare(result.denominator);
}
```

```
  public long memoryUse() {
    return numerator.memoryUse() + denominator.memoryUse();
  }
}
```

B.12 Goedel.java

```
package numbers.implementation;
import numbers.implementation.Number;
import numbers.INumber;
import numbers.implementation.PowerTable;
import numbers.IComparable;
import java.math.BigInteger;

public class Goedel
  implements
    INumber<Goedel>
{
  private static final long serialVersionUID = 20081020;

  public static final Number n0 = new Number("0");
  public static final Number n1 = new Number("1");
  public static final Number n2 = new Number("2");
  public static final Number n3 = new Number("3");
  public static final Number n5 = new Number("5");

  public static final GoedelToBigIntegerConvertor convertor = new GoedelToBigIntegerConvertor();

  public Number pow2;
  public Number pow3;
  public Number pow5;
  public Number value;

  public Goedel(Number _pow2, Number _pow3, Number _pow5) {
    pow2 = _pow2;
    pow3 = _pow3;
    pow5 = _pow5;
  }

  private void init(Number _value) {
    pow2=n0;
    pow3=n0;
    pow5=n0;
    while (_value.mod(n2).compareTo(n0)==0) {
      _value = _value.divide(n2);
      pow2 = pow2.add(n1);
    }
    while (_value.mod(n3).compareTo(n0)==0) {
      _value = _value.divide(n3);
      pow3 = pow3.add(n1);
    }
```

```
  while (_value.mod(n5).compareTo(n0)==0) {
    _value = _value.divide(n5);
    pow5 = pow5.add(n1);
  }
}

public Goedel(String _value) {
  init(new Number(_value));
}

public Goedel(BigInteger _value) {
  init(new Number(_value));
}

public BigInteger downcast() {
  return value().downcast();
}
public Number value() {
  if (value==null) {
    value = new Number(""+convertor.convert(this));
  }
  return value;
}

public Goedel multiply(Goedel _value) {
  return new Goedel(_value.pow2.add(pow2),_value.pow3.add(pow3),_value.pow5.add(pow5));
}
public Goedel divide(Goedel _value) {
  return new Goedel(pow2.subtract(_value.pow2),
                    pow3.subtract(_value.pow3),
                    pow5.subtract(_value.pow5));
}

public Goedel subtract(Goedel _value) {
  assert(false);
  return null;
}

public Goedel add(Goedel _value) {
  assert(false);
  return null;
}

public int compare(Goedel _value) {
  return value().compareTo(_value.value());
}

public boolean dividable(Goedel _value) {
  return divide(_value).notAFraction();
}

public Goedel gcd(Goedel _value) {
  Number p2 = pow2.compareTo(_value.pow2) < 0?pow2:_value.pow2;
  Number p3 = pow3.compareTo(_value.pow3) < 0?pow3:_value.pow3;
```

```
    Number p5 = pow5.compareTo(_value.pow5) < 0?pow5:_value.pow5;
    return new Goedel(p2, p3, p5);
  }

  public Goedel mod(Goedel _value) {
    return subtract(_value.multiply(divide(_value)));
  }

  public long memoryUse() {
    return pow2.bitCount() + pow3.bitCount() + pow5.bitCount() +(value==null?0:value.bitCount());
  }

  public void printStatistics() {
    pow2.printStatistics();
  }
  public String toString() {
    return "[" + pow2 + ", " + pow3 + ", " + pow5 + "]";
  }
  public boolean notAFraction() {
    return pow2.compareTo(n0)>=0 && pow3.compareTo(n0)>=0 && pow5.compareTo(n0)>=0;
  }
  public boolean equal(Goedel _value) {
    return pow2.equals(_value.pow2) && pow3.equals(_value.pow3) && pow5.equals(_value.pow5);
  }
  public BigInteger getValue() {
    return downcast();
  }

}
```

B.13 GoedelGenerator.java

```
package numbers.hamminggenerators.numbergenerators.implementation;
import numbers.implementation.Goedel;
import numbers.hamminggenerators.numbergenerators.INumberGenerator;

public class GoedelGenerator
  implements
    INumberGenerator<Goedel>
{
  public Goedel create(String _input) {
    return new Goedel(_input);
  }
}
```

B.14 GoedelToBigIntegerConvertor.java

```
package numbers.implementation;
import java.math.BigInteger;
```

```
public class GoedelToBigIntegerConvertor
{
  public static final PowerTable p2 = new PowerTable(new Number("2"));
  public static final PowerTable p3 = new PowerTable(new Number("3"));
  public static final PowerTable p5 = new PowerTable(new Number("5"));

  public static BigInteger convert(Goedel _number) {
    Number result = new Number("1");
    result = result.multiply(p2.find(""+_number.pow2));
    result = result.multiply(p3.find(""+_number.pow3));
    result = result.multiply(p5.find(""+_number.pow5));

    return result;
  }

  public static long getMemoryUse() {
    return p2.getMemoryUse() + p3.getMemoryUse() + p5.getMemoryUse();
  }
}
```

B.15 LinkedListItem.java

```
package numbers.hamminggenerators.dijkstra;
import numbers.INumber;

public class LinkedListItem<TNumber extends INumber<TNumber>>
{
  public TNumber value;
  public LinkedListItem<TNumber> next;

  public LinkedListItem(TNumber _value) {
    value = _value;
  }
}
```

B.16 LinkedListItemImproved.java

```
package numbers.hamminggenerators.dijkstra;
import numbers.INumber;

public class LinkedListItemImproved<TNumber extends INumber<TNumber>>
{
  public int stopAt;
  public TNumber value;
  public LinkedListItemImproved<TNumber> next;

  public LinkedListItemImproved(TNumber _value, int _stopAt) {
    value = _value;
```

```
    stopAt = _stopAt;
  }
}
```

B.17 LinkedListItemPointer.java

```
package numbers.hamminggenerators.dijkstra;
import numbers.INumber;

public class LinkedListItemPointer<TNumber extends INumber<TNumber>>
{
  public LinkedListItem<TNumber> item;
  public TNumber base;
  public TNumber nextValue;

  public LinkedListItemPointer(TNumber _base, LinkedListItem<TNumber> _item) {
    base = _base;
    item = _item;
    nextValue = base.multiply(item.value);
  }

  public void moveNext() {
    item = item.next;
    nextValue = base.multiply(item.value);
  }
}
```

B.18 LinkedListItemPointerImproved.java

```
package numbers.hamminggenerators.dijkstra;
import numbers.INumber;

public class LinkedListItemPointerImproved<TNumber extends INumber<TNumber>>
{
  private int jumpTo;
  public LinkedListItemImproved<TNumber> item;
  public TNumber base;
  public TNumber nextValue;


  public LinkedListItemPointerImproved(TNumber _base, LinkedListItemImproved<TNumber> _item,
                                       int _jumpTo) {
    base = _base;
    item = _item;
    nextValue = base.multiply(item.value);
    jumpTo = _jumpTo;
  }
```

```
  public void moveNext() {
    while (true) {
      item = item.next;
      if (jumpTo>=item.stopAt) {
        nextValue = base.multiply(item.value);
        return;
      }
    }
  }
}
```

B.19 LookupTable.java

```
package numbers.implementation;
import java.lang.Comparable;
import java.util.TreeSet;

public class LookupTable<IDType extends Comparable<IDType>, ResultType>
  extends
    TreeSet<LookupTableAtom<IDType, ResultType>>
{
  private static final long serialVersionUID = 20081020;

  public void add(IDType _id, ResultType _result) {
    add(new LookupTableAtom<IDType, ResultType>(_id, _result));
  }

  public ResultType find(IDType _id) {
    LookupTableAtom<IDType, ResultType> atom = floor(new LookupTableAtom<IDType,
                                                        ResultType>(_id,null));
    if (atom !=null && atom.id.compareTo(_id)==0) {
      return atom.result;
    } else {
      return null;
    }
  }
}



class LookupTableAtom<IDType extends Comparable<IDType>, ResultType>
  implements
    Comparable<LookupTableAtom<IDType, ResultType>>
{
  public IDType id;
  public ResultType result;

  public LookupTableAtom(IDType _id, ResultType _result) {
    id = _id;
    result = _result;
  }
```

```
  public int compareTo(LookupTableAtom<IDType, ResultType> _value) {
    return id.compareTo(_value.id);
  }
}
```

B.20 Number.java

```
package numbers.implementation;
import numbers.efficiency.EfficiencyRegistrator;
import numbers.INumber;
import numbers.IComparable;
import java.math.BigInteger;

public class Number
  extends
    BigInteger
  implements
    INumber<Number>
{
  private static final long serialVersionUID = 20081020;

  public static EfficiencyRegistrator registrator = new EfficiencyRegistrator();

  public Number(String _value) {
    super(_value);
  }

  public Number(BigInteger _value) {
    super(_value.toString());
  }

  public BigInteger downcast() {
    return this;
  }

  public Number multiply(Number _value) {
    registrator.multiply(bitCount(),_value.bitCount());
    return new Number(multiply(_value.downcast()));
  }
  public Number divide(Number _value) {
    registrator.multiply(bitCount(),_value.bitCount());
    return new Number(divide(_value.downcast()));
  }

  public Number subtract(Number _value) {
    registrator.add(bitCount(),_value.bitCount());
    return new Number(subtract(_value.downcast()));
  }

  public Number add(Number _value) {
```

```
    registrator.add(bitCount(),_value.bitCount());
    return new Number(add(_value.downcast()));
  }

  public boolean dividable(Number _value) {
    return mod(_value).signum()==0;
  }

  public int compare(Number _value) {
    registrator.compare(bitCount(),_value.bitCount());
    return compareTo(_value.downcast());
  }

  public Number gcd(Number _value) {
    int r = compare(_value);
    if (r==0) {
      return this;
    }
    if (r==-1) {
      Number t = _value.mod(this);
      if (t.equals(BigInteger.ZERO)) {
        return this;
      }
      return gcd(t);
    }
    Number t = mod(_value);
    if (t.equals(BigInteger.ZERO)) {
      return _value;
    }
    return t.gcd(_value);
  }

  public Number mod(Number _value) {
    return subtract(_value.multiply(divide(_value)));
  }

  public long memoryUse() {
    return bitCount();
  }

  public void printStatistics() {
    System.out.println("Not yet implemented");
  }
  public INumber value() {
    return this;
  }
  public boolean equal(Number _value) {
    return equals(_value);
  }
  public BigInteger getValue() {
    return this;
  }
}
```

B.21 NumberGenerator.java

```
package numbers.hamminggenerators.numbergenerators.implementation;
import numbers.implementation.Number;
import numbers.hamminggenerators.numbergenerators.INumberGenerator;

public class NumberGenerator
  implements
    INumberGenerator<Number>
{
  public Number create(String _input) {
    return new Number(_input);
  }
}
```

B.22 PowerTable.java

```
package numbers.implementation;
import java.math.BigInteger;

public class PowerTable
  extends
    LookupTable<String,Number>
{
  private static final long serialVersionUID = 20081020;

  private Number base;

  protected void init(Number _base) {
    base = _base;
    add("1",_base);
    add("0",new Number("1"));
  }
  public PowerTable(Number _base) {
    init( _base);
  }

  public Number find(String _n) {
    Number result = super.find(_n);

    if (result==null) {
      int i = Integer.parseInt(_n);

      if (i%2==0) {
        result= find(""+(i/2));
        result = result.multiply(result);
        add(_n,result);
      } else {
        result= find(""+(i-1));
        result = result.multiply(base);
        add(_n,result);
      }
      add(_n,result);
```

```
    }

    return result;
  }

  public long getMemoryUse() {
    long result = 0;

    for(LookupTableAtom<String,Number> atom:this) {
      result += atom.result.memoryUse();
    }

    return result;
  }
}
```

B.23 Test.java

```
package test;
import numbers.hamminggenerators.IGenerator;
import numbers.implementation.Number;

public class Test
{
  public static void first(IGenerator _generator, int _max) throws Exception {
    int step = 20000;
    for(int i = 0;i<_max;i++) {
      _generator.next();
      if (i%step==0) {
        System.out.println(i + " " + Number.registrator + " " + _generator.getMemory());
      }
    }
  }

  public static void test(IGenerator _generator) throws Exception {
    long t = System.currentTimeMillis();
    firstValue(_generator, 1000001);
    System.out.println((0.0 + (System.currentTimeMillis()-t))/1000 + " seconds.");
  }
}
```