LAMPIRAN A

Tabel A.1 Kapasitas Data pada QR *code*[2]

Version	No. of Modules/ side (A)	Function pattern modules (B)	Format and Version Information modules (C)	Data modules except (C) ($D=A^2-B-C$)	Data capacity [codewords][a] (E)	Remainder Bits
1	21	202	31	208	26	0
2	25	235	31	359	44	7
3	29	243	31	567	70	7
4	33	251	31	807	100	7
5	37	259	31	1 079	134	7
6	41	267	31	1 383	172	7
7	45	390	67	1 568	196	0
8	49	398	67	1 936	242	0
9	53	406	67	2 336	292	0
10	57	414	67	2 768	346	0
11	61	422	67	3 232	404	0
12	65	430	67	3 728	466	0
13	69	438	67	4 256	532	0
14	73	611	67	4 651	581	3
15	77	619	67	5 243	655	3
16	81	627	67	5 867	733	3
17	85	635	67	6 523	815	3
18	89	643	67	7 211	901	3
19	93	651	67	7 931	991	3
20	97	659	67	8 683	1 085	3
21	101	882	67	9 252	1 156	4
22	105	890	67	10 068	1 258	4
23	109	898	67	10 916	1 364	4
24	113	906	67	11 796	1 474	4
25	117	914	67	12 708	1 588	4
26	121	922	67	13 652	1 706	4
27	125	930	67	14 628	1 828	4
28	129	1 203	67	15 371	1 921	3
29	133	1 211	67	16 411	2 051	3
30	137	1 219	67	17 483	2 185	3
31	141	1 227	67	18 587	2 323	3
32	145	1 235	67	19 723	2 465	3
33	149	1 243	67	20 891	2 611	3
34	153	1 251	67	22 091	2 761	3
35	157	1 574	67	23 008	2 876	0
36	161	1 582	67	24 272	3 034	0
37	165	1 590	67	25 568	3 196	0
38	169	1 598	67	26 896	3 362	0
39	173	1 606	67	28 256	3 532	0
40	177	1 614	67	29 648	3 706	0

Tabel A.2 Jumlah Codeword dan Kapasitas Data Masukan Berdasarkan Tingkatan Error Correction[2]

Version	Error correction level	Number of data codewords[a]	Number of data bits[b]	Data capacity			
				Numeric	Alphanumeric	8-bit Byte	Kanji
1	L	19	152	41	25	17	10
	M	16	128	34	20	14	8
	Q	13	104	27	16	11	7
	H	9	72	17	10	7	4
2	L	34	272	77	47	32	20
	M	28	224	63	38	26	16
	Q	22	176	48	29	20	12
	H	16	128	34	20	14	8
3	L	55	440	127	77	53	32
	M	44	352	101	61	42	26
	Q	34	272	77	47	32	20
	H	26	208	58	35	24	15
4	L	80	640	187	114	78	48
	M	64	512	149	90	62	38
	Q	48	384	111	67	46	28
	H	36	288	82	50	34	21
5	L	108	864	255	154	106	65
	M	86	688	202	122	84	52
	Q	62	496	144	87	60	37
	H	46	368	106	64	44	27
6	L	136	1 088	322	195	134	82
	M	108	864	255	154	106	65
	Q	76	608	178	108	74	45
	H	60	480	139	84	58	36
7	L	156	1 248	370	224	154	95
	M	124	992	293	178	122	75
	Q	88	704	207	125	86	53
	H	66	528	154	93	64	39
8	L	194	1 552	461	279	192	118
	M	154	1 232	365	221	152	93
	Q	110	880	259	157	108	66
	H	86	688	202	122	84	52
9	L	232	1 856	552	335	230	141
	M	182	1 456	432	262	180	111
	Q	132	1 056	312	189	130	80
	H	100	800	235	143	98	60

10	L	274	2 192	652	395	271	167
	M	216	1 728	513	311	213	131
	Q	154	1 232	364	221	151	93
	H	122	976	288	174	119	74
11	L	324	2 592	772	468	321	198
	M	254	2 032	604	366	251	155
	Q	180	1 440	427	259	177	109
	H	140	1 120	331	200	137	85
12	L	370	2 960	883	535	367	226
	M	290	2 320	691	419	287	177
	Q	206	1 648	489	296	203	125
	H	158	1 264	374	227	155	96
13	L	428	3 424	1 022	619	425	262
	M	334	2 672	796	483	331	204
	Q	244	1 952	580	352	241	149
	H	180	1 440	427	259	177	109
14	L	461	3 688	1 101	667	458	282
	M	365	2 920	871	528	362	223
	Q	261	2 088	621	376	258	159
	H	197	1 576	468	283	194	120
15	L	523	4 184	1 250	758	520	320
	M	415	3 320	991	600	412	254
	Q	295	2 360	703	426	292	180
	H	223	1 784	530	321	220	136
16	L	589	4 712	1 408	854	586	361
	M	453	3 624	1 082	656	450	277
	Q	325	2 600	775	470	322	198
	H	253	2 024	602	365	250	154
17	L	647	5 176	1 548	938	644	397
	M	507	4 056	1 212	734	504	310
	Q	367	2 936	876	531	364	224
	H	283	2 264	674	408	280	173
18	L	721	5 768	1 725	1 046	718	442
	M	563	4 504	1 346	816	560	345
	Q	397	3 176	948	574	394	243
	H	313	2 504	746	452	310	191
19	L	795	6 360	1 903	1 153	792	488
	M	627	5 016	1 500	909	624	384
	Q	445	3 560	1 063	644	442	272
	H	341	2 728	813	493	338	208

20	L	861	6 888	2 061	1 249	858	528
	M	669	5 352	1 600	970	666	410
	Q	485	3 880	1 159	702	482	297
	H	385	3 080	919	557	382	235
21	L	932	7 456	2 232	1 352	929	572
	M	714	5 712	1 708	1 035	711	438
	Q	512	4 096	1 224	742	509	314
	H	406	3 248	969	587	403	248
22	L	1 006	8 048	2 409	1 460	1 003	618
	M	782	6 256	1 872	1 134	779	480
	Q	568	4 544	1 358	823	565	348
	H	442	3 536	1 056	640	439	270
23	L	1 094	8 752	2 620	1 588	1 091	672
	M	860	6 880	2 059	1 248	857	528
	Q	614	4 912	1 468	890	611	376
	H	464	3 712	1 108	672	461	284
24	L	1 174	9 392	2 812	1 704	1 171	721
	M	914	7 312	2 188	1 326	911	561
	Q	664	5 312	1 588	963	661	407
	H	514	4 112	1 228	744	511	315
25	L	1 276	10 208	3 057	1 853	1 273	784
	M	1 000	8 000	2 395	1 451	997	614
	Q	718	5 744	1 718	1 041	715	440
	H	538	4 304	1 286	779	535	330
26	L	1 370	10 960	3 283	1 990	1 367	842
	M	1 062	8 496	2 544	1 542	1 059	652
	Q	754	6 032	1 804	1 094	751	462
	H	596	4 768	1 425	864	593	365
27	L	1 468	11 744	3 517	2 132	1 465	902
	M	1 128	9 024	2 701	1 637	1 125	692
	Q	808	6 464	1 933	1 172	805	496
	H	628	5 024	1 501	910	625	385
28	L	1 531	12 248	3 669	2 223	1 528	940
	M	1 193	9 544	2 857	1 732	1 190	732
	Q	871	6 968	2 085	1 263	868	534
	H	661	5 288	1 581	958	658	405
29	L	1 631	13 048	3 909	2 369	1 628	1 002
	M	1 267	10 136	3 035	1 839	1 264	778
	Q	911	7 288	2 181	1 322	908	559
	H	701	5 608	1 677	1 016	698	430
30	L	1 735	13 880	4 158	2 520	1 732	1 066
	M	1 373	10 984	3 289	1 994	1 370	843
	Q	985	7 880	2 358	1 429	982	604
	H	745	5 960	1 782	1 080	742	457

31	L	1 843	14 744	4 417	2 677	1 840	1 132
	M	1 455	11 640	3 486	2 113	1 452	894
	Q	1 033	8 264	2 473	1 499	1 030	634
	H	793	6 344	1 897	1 150	790	486
32	L	1 955	15 640	4 686	2 840	1 952	1 201
	M	1 541	12 328	3 693	2 238	1 538	947
	Q	1 115	8 920	2 670	1 618	1 112	684
	H	845	6 760	2 022	1 226	842	518
33	L	2 071	16 568	4 965	3 009	2 068	1 273
	M	1 631	13 048	3 909	2 369	1 628	1 002
	Q	1 171	9 368	2 805	1 700	1 168	719
	H	901	7 208	2 157	1 307	898	553
34	L	2 191	17 528	5 253	3 183	2 188	1 347
	M	1 725	13 800	4 134	2 506	1 722	1 060
	Q	1 231	9 848	2 949	1 787	1 228	756
	H	961	7 688	2 301	1 394	958	590
35	L	2 306	18 448	5 529	3 351	2 303	1 417
	M	1 812	14 496	4 343	2 632	1 809	1 113
	Q	1 286	10 288	3 081	1 867	1 283	790
	H	986	7 888	2 361	1 431	983	605
36	L	2 434	19 472	5 836	3 537	2 431	1 496
	M	1 914	15 312	4 588	2 780	1 911	1 176
	Q	1 354	10 832	3 244	1 966	1 351	832
	H	1 054	8 432	2 524	1 530	1 051	647
37	L	2 566	20 528	6 153	3 729	2 563	1 577
	M	1 992	15 936	4 775	2 894	1 989	1 224
	Q	1 426	11 408	3 417	2 071	1 423	876
	H	1 096	8 768	2 625	1 591	1 093	673
38	L	2 702	21 616	6 479	3 927	2 699	1 661
	M	2 102	16 816	5 039	3 054	2 099	1 292
	Q	1 502	12 016	3 599	2 181	1 499	923
	H	1 142	9 136	2 735	1 658	1 139	701
39	L	2 812	22 496	6 743	4 087	2 809	1 729
	M	2 216	17 728	5 313	3 220	2 213	1 362
	Q	1 582	12 656	3 791	2 298	1 579	972
	H	1 222	9 776	2 927	1 774	1 219	750
40	L	2 956	23 648	7 089	4 296	2 953	1 817
	M	2 334	18 672	5 596	3 391	2 331	1 435
	Q	1 666	13 328	3 993	2 420	1 663	1 024
	H	1 276	10 208	3 057	1 852	1 273	784

[a] All codewords shall be 8 bits in length.

[b] The number of Data Bits includes bits for Mode Indicator and Character Count Indicator.

Tabel A.3 Jumlah *Error Correction Codeword*[2]

Version	Total number of codewords	Error correction level	Number of error correction codewords	Number of error correction blocks	Error correction code per block [a]
1	26	L	7	1	(26,19,2) [b]
		M	10	1	(26,16,4) [b]
		Q	13	1	(26,13,6) [b]
		H	17	1	(26,9,8) [b]
2	44	L	10	1	(44,34,4) [b]
		M	16	1	(44,28,8)
		Q	22	1	(44,22,11)
		H	28	1	(44,16,14)
3	70	L	15	1	(70,55,7) [b]
		M	26	1	(70,44,13)
		Q	36	2	(35,17,9)
		H	44	2	(35,13,11)
4	100	L	20	1	(100,80,10)
		M	36	2	(50,32,9)
		Q	52	2	(50,24,13)
		H	64	4	(25,9,8)
5	134	L	26	1	(134,108,13)
		M	48	2	(67,43,12)
		Q	72	2 2	(33,15,9) (34,16,9)
		H	88	2 2	(33,11,11) (34,12,11)
6	172	L	36	2	(86,68,9)
		M	64	4	(43,27,8)
		Q	96	4	(43,19,12)
		H	112	4	(43,15,14)
7	196	L	40	2	(98,78,10)
		M	72	4	(49,31,9)
		Q	108	2 4	(32,14,9) (33,15,9)
		H	130	4 1	(39,13,13) (40,14,13)

8	242	L	48	2	(121,97,12)
		M	88	2 2	(60,38,11) (61,39,11)
		Q	132	4 2	(40,18,11) (41,19,11)
		H	156	4 2	(40,14,13) (41,15,13)
9	292	L	60	2	(146,116,15)
		M	110	3 2	(58,36,11) (59,37,11)
		Q	160	4 4	(36,16,10) (37,17,10)
		H	192	4 4	(36,12,12) (37,13,12)
10	346	L	72	2 2	(86,68,9) (87,69,9)
		M	130	4 1	(69,43,13) (70,44,13)
		Q	192	6 2	(43,19,12) (44,20,12)
		H	224	6 2	(43,15,14) (44,16,14)
11	404	L	80	4	(101,81,10)
		M	150	1 4	(80,50,15) (81,51,15)
		Q	224	4 4	(50,22,14) (51,23,14)
		H	264	3 8	(36,12,12) (37,13,12)
12	466	L	96	2 2	(116,92,12) (117,93,12)
		M	176	6 2	(58,36,11) (59,37,11)
		Q	260	4 6	(46,20,13) (47,21,13)
		H	308	7 4	(42,14,14) (43,15,14)
13	532	L	104	4	(133,107,13)
		M	198	8 1	(59,37,11) (60,38,11)
		Q	288	8 4	(44,20,12) (45,21,12)
		H	352	12 4	(33,11,11) (34,12,11)

14	581	L	120	3 1	(145,115,15) (146,116,15)
		M	216	4 5	(64,40,12) (65,41,12)
		Q	320	11 5	(36,16,10) (37,17,10)
		H	384	11 5	(36,12,12) (37,13,12)
15	655	L	132	5 1	(109,87,11) (110,88,11)
		M	240	5 5	(65,41,12) (66,42,12)
		Q	360	5 7	(54,24,15) (55,25,15)
		H	432	11 7	(36,12,12) (37,13,12)
16	733	L	144	5 1	(122,98,12) (123,99,12)
		M	280	7 3	(73,45,14) (74,46,14)
		Q	408	15 2	(43,19,12) (44,20,12)
		H	480	3 13	(45,15,15) (46,16,15)
17	815	L	168	1 5	(135,107,14) (136,108,14)
		M	308	10 1	(74,46,14) (75,47,14)
		Q	448	1 15	(50,22,14) (51,23,14)
		H	532	2 17	(42,14,14) (43,15,14)
18	901	L	180	5 1	(150,120,15) (151,121,15)
		M	338	9 4	(69,43,13) (70,44,13)
		Q	504	17 1	(50,22,14) (51,23,14)
		H	588	2 19	(42,14,14) (43,15,14)

19	991	L	196	3 4	(141,113,14) (142,114,14)
		M	364	3 11	(70,44,13) (71,45,13)
		Q	546	17 4	(47,21,13) (48,22,13)
		H	650	9 16	(39,13,13) (40,14,13)
20	1 085	L	224	3 5	(135,107,14) (136,108,14)
		M	416	3 13	(67,41,13) (68,42,13)
		Q	600	15 5	(54,24,15) (55,25,15)
		H	700	15 10	(43,15,14) (44,16,14)
21	1 156	L	224	4 4	(144,116,14) (145,117,14)
		M	442	17	(68,42,13)
		Q	644	17 6	(50,22,14) (51,23,14)
		H	750	19 6	(46,16,15) (47,17,15)
22	1 258	L	252	2 7	(139,111,14) (140,112,14)
		M	476	17	(74,46,14)
		Q	690	7 16	(54,24,15) (55,25,15)
		H	816	34	(37,13,12)
23	1 364	L	270	4 5	(151,121,15) (152,122,15)
		M	504	4 14	(75,47,14) (76,48,14)
		Q	750	11 14	(54,24,15) (55,25,15)
		H	900	16 14	(45,15,15) (46,16,15)

24	1 474	L	300	6 4	(147,117,15) (148,118,15)
		M	560	6 14	(73,45,14) (74,46,14)
		Q	810	11 16	(54,24,15) (55,25,15)
		H	960	30 2	(46,16,15) (47,17,15)
25	1 588	L	312	8 4	(132,106,13) (133,107,13)
		M	588	8 13	(75,47,14) (76,48,14)
		Q	870	7 22	(54,24,15) (55,25,15)
		H	1050	22 13	(45,15,15) (46,16,15)
26	1 706	L	336	10 2	(142,114,14) (143,115,14)
		M	644	19 4	(74,46,14) (75,47,14)
		Q	952	28 6	(50,22,14) (51,23,14)
		H	1110	33 4	(46,16,15) (47,17,15)
27	1 828	L	360	8 4	(152,122,15) (153,123,15)
		M	700	22 3	(73,45,14) (74,46,14)
		Q	1 020	8 26	(53,23,15) (54,24,15)
		H	1 200	12 28	(45,15,15) (46,16,15)
28	1 921	L	390	3 10	(147,117,15) (148,118,15)
		M	728	3 23	(73,45,14) (74,46,14)
		Q	1 050	4 31	(54,24,15) (55,25,15)
		H	1 260	11 31	(45,15,15) (46,16,15)

29	2 051	L	420	7 7	(146,116,15) (147,117,15)
		M	784	21 7	(73,45,14) (74,46,14)
		Q	1 140	1 37	(53,23,15) (54,24,15)
		H	1 350	19 26	(45,15,15) (46,16,15)
30	2 185	L	450	5 10	(145,115,15) (146,116,15)
		M	812	19 10	(75,47,14) (76,48,14)
		Q	1 200	15 25	(54,24,15) (55,25,15)
		H	1 440	23 25	(45,15,15) (46,16,15)
31	2 323	L	480	13 3	(145,115,15) (146,116,15)
		M	868	2 29	(74,46,14) (75,47,14)
		Q	1 290	42 1	(54,24,15) (55,25,15)
		H	1 530	23 28	(45,15,15) (46,16,15)
32	2 465	L	510	17	(145,115,15)
		M	924	10 23	(74,46,14) (75,47,14)
		Q	1 350	10 35	(54,24,15) (55,25,15)
		H	1 620	19 35	(45,15,15) (46,16,15)
33	2 611	L	540	17 1	(145,115,15) (146,116,15)
		M	980	14 21	(74,46,14) (75,47,14)
		Q	1 440	29 19	(54,24,15) (55,25,15)
		H	1 710	11 46	(45,15,15) (46,16,15)

34	2 761	L	570	13 6	(145,115,15) (146,116,15)
		M	1 036	14 23	(74,46,14) (75,47,14)
		Q	1 530	44 7	(54,24,15) (55,25,15)
		H	1 800	59 1	(46,16,15) (47,17,15)
35	2 876	L	570	12 7	(151,121,15) (152,122,15)
		M	1 064	12 26	(75,47,14) (76,48,14)
		Q	1 590	39 14	(54,24,15) (55,25,15)
		H	1 890	22 41	(45,15,15) (46,16,15)
36	3 034	L	600	6 14	(151,121,15) (152,122,15)
		M	1 120	6 34	(75,47,14) (76,48,14)
		Q	1 680	46 10	(54,24,15) (55,25,15)
		H	1 980	2 64	(45,15,15) (46,16,15)
37	3 196	L	630	17 4	(152,122,15) (153,123,15)
		M	1 204	29 14	(74,46,14) (75,47,14)
		Q	1 770	49 10	(54,24,15) (55,25,15)
		H	2 100	24 46	(45,15,15) (46,16,15)
38	3 362	L	660	4 18	(152,122,15) (153,123,15)
		M	1 260	13 32	(74,46,14) (75,47,14)
		Q	1 860	48 14	(54,24,15) (55,25,15)
		H	2 220	42 32	(45,15,15) (46,16,15)

39	3 532	L	720	20 4	(147,117,15) (148,118,15)
		M	1 316	40 7	(75,47,14) (76,48,14)
		Q	1 950	43 22	(54,24,15) (55,25,15)
		H	2 310	10 67	(45,15,15) (46,16,15)
40	3 706	L	750	19 6	(148,118,15) (149,119,15)
		M	1 372	18 31	(75,47,14) (76,48,14)
		Q	2 040	34 34	(54,24,15) (55,25,15)
		H	2 430	20 61	(45,15,15) (46,16,15)

[a] (c, k, r): c = total number of codewords
k = number of data codewords
r = number of error correction capacity

Tabel A.4 *Generator Polynomial*

Number of error correction codewords	Generator polynomials
7	$x^7+\alpha^{87}x^6+\alpha^{229}x^5+\alpha^{146}x^4+\alpha^{149}x^3+\alpha^{238}x^2+\alpha^{102}x+\alpha^{21}$
10	$x^{10}+\alpha^{251}x^9+\alpha^{67}x^8+\alpha^{46}x^7+\alpha^{61}x^6+\alpha^{118}x^5+\alpha^{70}x^4+\alpha^{64}x^3+\alpha^{94}x^2+\alpha^{32}x+\alpha^{45}$
13	$x^{13}+\alpha^{74}x^{12}+\alpha^{152}x^{11}+\alpha^{176}x^{10}+\alpha^{100}x^9+\alpha^{86}x^8+\alpha^{100}x^7+\alpha^{106}x^6+\alpha^{104}x^5+\alpha^{130}x^4+\alpha^{218}x^3+\alpha^{206}x^2+\alpha^{140}x+\alpha^{78}$
15	$x^{15}+\alpha^{8}x^{14}+\alpha^{183}x^{13}+\alpha^{61}x^{12}+\alpha^{91}x^{11}+\alpha^{202}x^{10}+\alpha^{37}x^9+\alpha^{51}x^8+\alpha^{58}x^7+\alpha^{58}x^6+\alpha^{237}x^5+\alpha^{140}x^4+\alpha^{124}x^3+\alpha^{5}x^2+\alpha^{99}x+\alpha^{105}$
16	$x^{16}+\alpha^{120}x^{15}+\alpha^{104}x^{14}+\alpha^{107}x^{13}+\alpha^{109}x^{12}+\alpha^{102}x^{11}+\alpha^{161}x^{10}+\alpha^{76}x^9+\alpha^{3}x^8+\alpha^{91}x^7+\alpha^{191}x^6+\alpha^{147}x^5+\alpha^{169}x^4\alpha^{182}x^3+\alpha^{194}x^2+\alpha^{225}x+\alpha^{120}$
17	$x^{17}+\alpha^{43}x^{16}+\alpha^{139}x^{15}+\alpha^{206}x^{14}+\alpha^{78}x^{13}+\alpha^{43}x^{12}+\alpha^{239}x^{11}+\alpha^{123}x^{10}+\alpha^{206}x^9+\alpha^{214}x^8+\alpha^{147}x^7+\alpha^{24}x^6+\alpha^{99}x^5+\alpha^{150}x^4+\alpha^{39}x^3+\alpha^{243}x^2+\alpha^{163}x+\alpha^{136}$
18	$x^{18}+\alpha^{215}x^{17}+\alpha^{234}x^{16}+\alpha^{158}x^{15}+\alpha^{94}x^{14}+\alpha^{184}x^{13}+\alpha^{97}x^{12}+\alpha^{118}x^{11}+\alpha^{170}x^{10}+\alpha^{79}x^9+\alpha^{187}x^8+\alpha^{152}x^7+\alpha^{148}x^6+\alpha^{252}x^5+\alpha^{179}x^4+\alpha^{5}x^3+\alpha^{98}x^2+\alpha^{96}x+\alpha^{153}$
20	$x^{20}+\alpha^{17}x^{19}+\alpha^{60}x^{18}+\alpha^{79}x^{17}+\alpha^{50}x^{16}+\alpha^{61}x^{15}+\alpha^{163}x^{14}+\alpha^{26}x^{13}+\alpha^{187}x^{12}+\alpha^{202}x^{11}+\alpha^{180}x^{10}+\alpha^{221}x^9+\alpha^{225}x^8+\alpha^{83}x^7+\alpha^{239}x^6+\alpha^{156}x^5+\alpha^{164}x^4+\alpha^{212}x^3+\alpha^{212}x^2+\alpha^{188}x+\alpha^{190}$
22	$x^{22}+\alpha^{210}x^{21}+\alpha^{171}x^{20}+\alpha^{247}x^{19}+\alpha^{242}x^{18}+\alpha^{93}x^{17}+\alpha^{230}x^{16}+\alpha^{14}x^{15}+\alpha^{109}x^{14}+\alpha^{221}x^{13}+\alpha^{53}x^{12}+\alpha^{200}x^{11}+\alpha^{74}x^{10}+\alpha^{8}x^9+\alpha^{172}x^8+\alpha^{98}x^7+\alpha^{80}x^6+\alpha^{219}x^5+\alpha^{134}x^4+\alpha^{160}x^3+\alpha^{105}x^2+\alpha^{165}x+\alpha^{231}$
24	$x^{24}+\alpha^{229}x^{23}+\alpha^{121}x^{22}+\alpha^{135}x^{21}+\alpha^{48}x^{20}+\alpha^{211}x^{19}+\alpha^{117}x^{18}\alpha^{251}x^{17}+\alpha^{126}x^{16}+\alpha^{159}x^{15}+\alpha^{180}x^{14}+\alpha^{169}x^{13}+\alpha^{152}x^{12}+\alpha^{192}x^{11}+\alpha^{226}x^{10}+\alpha^{228}x^9+\alpha^{218}x^8+\alpha^{111}x^7+x^6+\alpha^{117}x^5+\alpha^{232}x^4+\alpha^{87}x^3+\alpha^{96}x^2+\alpha^{227}x+\alpha^{21}$
26	$x^{26}+\alpha^{173}x^{25}+\alpha^{125}x^{24}+\alpha^{158}x^{23}+\alpha^{2}x^{22}+\alpha^{103}x^{21}+\alpha^{182}x^{20}+\alpha^{118}x^{19}+\alpha^{17}x^{18}+\alpha^{145}x^{17}+\alpha^{201}x^{16}+\alpha^{111}x^{15}+\alpha^{28}x^{14}+\alpha^{165}x^{13}+\alpha^{53}x^{12}+\alpha^{161}x^{11}+\alpha^{21}x^{10}+\alpha^{245}x^9+\alpha^{142}x^8+\alpha^{13}x^7+\alpha^{102}x^6+\alpha^{48}x^5+\alpha^{227}x^4+\alpha^{153}x^3+\alpha^{145}x^2+\alpha^{218}x+\alpha^{70}$

28	$x^{28}+\alpha^{168}x^{27}+\alpha^{223}x^{26}+\alpha^{200}x^{25}+\alpha^{104}x^{24}+\alpha^{224}x^{23}+\alpha^{234}x^{22}$ $+\alpha^{108}x^{21}+\alpha^{180}x^{20}+\alpha^{110}x^{19}+\alpha^{190}x^{18}+\alpha^{195}x^{17}+\alpha^{147}x^{16}$ $+\alpha^{205}x^{15}+\alpha^{27}x^{14}+\alpha^{232}x^{13}+\alpha^{201}x^{12}+\alpha^{21}x^{11}+\alpha^{43}x^{10}$ $+\alpha^{245}x^{9}+\alpha^{87}x^{8}+\alpha^{42}x^{7}+\alpha^{195}x^{6}+\alpha^{212}x^{5}+\alpha^{119}x^{4}+\alpha^{242}x^{3}$ $+\alpha^{37}x^{2}+\alpha^{9}x+\alpha^{123}$
30	$x^{30}+\alpha^{41}x^{29}+\alpha^{173}x^{28}+\alpha^{145}x^{27}+\alpha^{152}x^{26}+\alpha^{216}x^{25}+\alpha^{31}x^{24}$ $+\alpha^{179}x^{23}+\alpha^{182}x^{22}+\alpha^{50}x^{21}+\alpha^{48}x^{20}+\alpha^{110}x^{19}+\alpha^{86}x^{18}$ $+\alpha^{239}x^{17}+\alpha^{96}x^{16}+\alpha^{222}x^{15}+\alpha^{125}x^{14}+\alpha^{42}x^{13}+\alpha^{173}x^{12}$ $+\alpha^{226}x^{11}+\alpha^{193}x^{10}+\alpha^{224}x^{9}+\alpha^{130}x^{8}+\alpha^{156}x^{7}+\alpha^{37}x^{6}$ $+\alpha^{251}x^{5}+\alpha^{216}x^{4}+\alpha^{238}x^{3}+\alpha^{40}x^{2}+\alpha^{192}x+\alpha^{180}$
32	$x^{32}\alpha^{10}x^{31}+\alpha^{6}x^{30}+\alpha^{106}x^{29}+\alpha^{190}x^{28}+\alpha^{249}x^{27}+\alpha^{167}x^{26}$ $+\alpha^{4}x^{25}+\alpha^{67}x^{24}+\alpha^{209}x^{23}+\alpha^{138}x^{22}+\alpha^{138}x^{21}+\alpha^{32}x^{20}$ $+\alpha^{242}x^{19}+\alpha^{123}x^{18}+\alpha^{89}x^{17}+\alpha^{27}x^{16}+\alpha^{120}x^{15}+\alpha^{185}x^{14}$ $+\alpha^{80}x^{13}+\alpha^{156}x^{12}+\alpha^{38}x^{11}+\alpha^{69}x^{10}+\alpha^{171}x^{9}+\alpha^{60}x^{8}+\alpha^{28}x^{7}$ $+\alpha^{222}x^{6}+\alpha^{80}x^{5}+\alpha^{52}x^{4}+\alpha^{254}x^{3}+\alpha^{185}x^{2}+\alpha^{220}x+\alpha^{241}$
34	$x^{34}+\alpha^{111}x^{33}+\alpha^{77}x^{32}+\alpha^{146}x^{31}+\alpha^{94}x^{30}+\alpha^{26}x^{29}+\alpha^{21}x^{28}$ $+\alpha^{108}x^{27}+\alpha^{19}x^{26}+\alpha^{105}x^{25}+\alpha^{94}x^{24}+\alpha^{113}x^{23}+\alpha^{193}x^{22}$ $+\alpha^{86}x^{21}+\alpha^{140}x^{20}+\alpha^{163}x^{19}+\alpha^{125}x^{18}+\alpha^{58}x^{17}+\alpha^{158}x^{16}$ $+\alpha^{229}x^{15}+\alpha^{239}x^{14}+\alpha^{218}x^{13}+\alpha^{103}x^{12}+\alpha^{56}x^{11}+\alpha^{70}x^{10}$ $+\alpha^{114}x^{9}+\alpha^{61}x^{8}+\alpha^{183}x^{7}+\alpha^{129}x^{6}+\alpha^{167}x^{5}+\alpha^{13}x^{4}+\alpha^{98}x^{3}$ $+\alpha^{62}x^{2}+\alpha^{129}x+\alpha^{51}$
36	$x^{36}+\alpha^{200}x^{35}+\alpha^{183}x^{34}+\alpha^{98}x^{33}+\alpha^{16}x^{32}+\alpha^{172}x^{31}+\alpha^{31}x^{30}$ $+\alpha^{246}x^{29}+\alpha^{234}x^{28}+\alpha^{60}x^{27}+\alpha^{152}x^{26}+\alpha^{115}x^{25}+x^{24}$ $+\alpha^{167}x^{23}+\alpha^{152}x^{22}+\alpha^{113}x^{21}+\alpha^{248}x^{20}+\alpha^{238}x^{19}+\alpha^{107}x^{18}$ $+\alpha^{18}x^{17}+\alpha^{63}x^{16}+\alpha^{218}x^{15}+\alpha^{37}x^{14}+\alpha^{87}x^{13}+\alpha^{210}x^{12}$ $+\alpha^{105}x^{11}+\alpha^{177}x^{10}+\alpha^{120}x^{9}+\alpha^{74}x^{8}+\alpha^{121}x^{7}+\alpha^{196}x^{6}$ $+\alpha^{117}x^{5}+\alpha^{251}x^{4}+\alpha^{113}x^{3}+\alpha^{233}x^{2}+\alpha^{30}x+\alpha^{120}$
40	$x^{40}+\alpha^{59}x^{39}+\alpha^{116}x^{38}+\alpha^{79}x^{37}+\alpha^{161}x^{36}+\alpha^{252}x^{35}+\alpha^{98}x^{34}$ $+\alpha^{128}x^{33}+\alpha^{205}x^{32}+\alpha^{128}x^{31}+\alpha^{161}x^{30}+\alpha^{247}x^{29}+\alpha^{57}x^{28}$ $+\alpha^{163}x^{27}+\alpha^{56}x^{26}+\alpha^{235}x^{25}+\alpha^{106}x^{24}+\alpha^{53}x^{23}+\alpha^{26}x^{22}$ $+\alpha^{187}x^{21}+\alpha^{174}x^{20}+\alpha^{226}x^{19}+\alpha^{104}x^{18}+\alpha^{170}x^{17}+\alpha^{7}x^{16}$ $+\alpha^{175}x^{15}+\alpha^{35}x^{14}+\alpha^{181}x^{13}+\alpha^{114}x^{12}+\alpha^{88}x^{11}+\alpha^{41}x^{10}$ $+\alpha^{47}x^{9}+\alpha^{163}x^{8}+\alpha^{125}x^{7}+\alpha^{134}x^{6}+\alpha^{72}x^{5}+\alpha^{20}x^{4}+\alpha^{232}x^{3}$ $+\alpha^{53}x^{2}+\alpha^{35}x+\alpha^{15}$
42	$x^{42}+\alpha^{250}x^{41}+\alpha^{103}x^{40}+\alpha^{221}x^{39}+\alpha^{230}x^{38}+\alpha^{25}x^{37}+\alpha^{18}x^{36}$ $+\alpha^{137}x^{35}+\alpha^{231}x^{34}+x^{33}+\alpha^{3}x^{32}+\alpha^{58}x^{31}+\alpha^{242}x^{30}+\alpha^{221}x^{29}$ $+\alpha^{191}x^{28}+\alpha^{110}x^{27}+\alpha^{84}x^{26}+\alpha^{230}x^{25}+\alpha^{8}x^{24}+\alpha^{188}x^{23}$ $+\alpha^{106}x^{22}+\alpha^{96}x^{21}+\alpha^{147}x^{20}+\alpha^{15}x^{19}+\alpha^{131}x^{18}+\alpha^{139}x^{17}$ $+\alpha^{34}x^{16}+\alpha^{101}x^{15}+\alpha^{223}x^{14}+\alpha^{39}x^{13}+\alpha^{101}x^{12}+\alpha^{213}x^{11}$ $+\alpha^{199}x^{10}+\alpha^{237}x^{9}+\alpha^{254}x^{8}+\alpha^{201}x^{7}+\alpha^{123}x^{6}+\alpha^{171}x^{5}$ $+\alpha^{162}x^{4}+\alpha^{194}x^{3}+\alpha^{117}x^{2}+\alpha^{50}x+\alpha^{96}$

44	$x^{44}+\alpha^{190}x^{43}+\alpha^{7}x^{42}+\alpha^{61}x^{41}+\alpha^{121}x^{40}+\alpha^{71}x^{39}+\alpha^{246}x^{38}$ $+\alpha^{69}x^{37}+\alpha^{55}x^{36}+\alpha^{168}x^{35}+\alpha^{188}x^{34}+\alpha^{89}x^{33}+\alpha^{243}x^{32}$ $+\alpha^{191}x^{31}+\alpha^{25}x^{30}+\alpha^{72}x^{29}+\alpha^{123}x^{28}+\alpha^{9}x^{27}+\alpha^{145}x^{26}$ $+\alpha^{14}x^{25}+\alpha^{247}x^{24}+\alpha x^{23}+\alpha^{238}x^{22}+\alpha^{44}x^{21}+\alpha^{78}x^{20}$ $+\alpha^{143}x^{19}+\alpha^{62}x^{18}+\alpha^{224}x^{17}+\alpha^{126}x^{16}+\alpha^{118}x^{15}+\alpha^{114}x^{14}$ $+\alpha^{68}x^{13}+\alpha^{163}x^{12}+\alpha^{52}x^{11}+\alpha^{194}x^{10}+\alpha^{217}x^{9}+\alpha^{147}x^{8}$ $+\alpha^{204}x^{7}+\alpha^{169}x^{6}+\alpha^{37}x^{5}+\alpha^{130}x^{4}+\alpha^{113}x^{3}+\alpha^{102}x^{2}+\alpha^{73}x+\alpha^{181}$
46	$x^{46}+\alpha^{112}x^{45}+\alpha^{94}x^{44}+\alpha^{88}x^{43}+\alpha^{112}x^{42}+\alpha^{253}x^{41}+\alpha^{224}x^{40}$ $+\alpha^{202}x^{39}+\alpha^{115}x^{38}+\alpha^{187}x^{37}+\alpha^{99}x^{36}+\alpha^{89}x^{35}+\alpha^{5}x^{34}$ $+\alpha^{54}x^{33}+\alpha^{113}x^{32}+\alpha^{129}x^{31}+\alpha^{44}x^{30}+\alpha^{58}x^{29}+\alpha^{16}x^{28}$ $+\alpha^{135}x^{27}+\alpha^{216}x^{26}+\alpha^{169}x^{25}+\alpha^{211}x^{24}+\alpha^{36}x^{23}+\alpha x^{22}$ $+\alpha^{4}x^{21}+\alpha^{96}x^{20}+\alpha^{60}x^{19}+\alpha^{241}x^{18}+\alpha^{73}x^{17}+\alpha^{104}x^{16}$ $+\alpha^{234}x^{15}+\alpha^{8}x^{14}+\alpha^{249}x^{13}+\alpha^{245}x^{12}+\alpha^{119}x^{11}+\alpha^{174}x^{10}$ $+\alpha^{52}x^{9}+\alpha^{25}x^{8}+\alpha^{157}x^{7}+\alpha^{224}x^{6}+\alpha^{43}x^{5}+\alpha^{202}x^{4}+\alpha^{223}x^{3}$ $+\alpha^{19}x^{2}+\alpha^{82}x+\alpha^{15}$
48	$x^{48}+\alpha^{228}x^{47}+\alpha^{25}x^{46}+\alpha^{196}x^{45}+\alpha^{130}x^{44}+\alpha^{211}x^{43}+\alpha^{146}x^{42}$ $+\alpha^{60}x^{41}+\alpha^{24}x^{40}+\alpha^{251}x^{39}+\alpha^{90}x^{38}+\alpha^{39}x^{37}+\alpha^{102}x^{36}$ $+\alpha^{240}x^{35}+\alpha^{61}x^{34}+\alpha^{178}x^{33}+\alpha^{63}x^{32}+\alpha^{46}x^{31}+\alpha^{123}x^{30}$ $+\alpha^{115}x^{29}+\alpha^{18}x^{28}+\alpha^{221}x^{27}+\alpha^{111}x^{26}+\alpha^{135}x^{25}+\alpha^{160}x^{24}$ $+\alpha^{182}x^{23}+\alpha^{205}x^{22}+\alpha^{107}x^{21}+\alpha^{206}x^{20}+\alpha^{95}x^{19}+\alpha^{150}x^{18}$ $+\alpha^{120}x^{17}+\alpha^{184}x^{16}+\alpha^{91}x^{15}+\alpha^{21}x^{14}+\alpha^{247}x^{13}+\alpha^{156}x^{12}$ $+\alpha^{140}x^{11}+\alpha^{238}x^{10}+\alpha^{191}x^{9}+\alpha^{11}x^{8}+\alpha^{94}x^{7}+\alpha^{227}x^{6}$ $+\alpha^{84}x^{5}+\alpha^{50}x^{4}+\alpha^{163}x^{3}+\alpha^{39}x^{2}+\alpha^{34}x+\alpha^{108}$
50	$x^{50}+\alpha^{232}x^{49}+\alpha^{125}x^{48}+\alpha^{157}x^{47}+\alpha^{161}x^{46}+\alpha^{164}x^{45}+\alpha^{9}x^{44}$ $+\alpha^{118}x^{43}+\alpha^{46}x^{42}+\alpha^{209}x^{41}+\alpha^{99}x^{40}+\alpha^{203}x^{39}+\alpha^{193}x^{38}$ $+\alpha^{35}x^{37}+\alpha^{3}x^{36}+\alpha^{209}x^{35}+\alpha^{111}x^{34}+\alpha^{195}x^{33}+\alpha^{242}x^{32}$ $+\alpha^{203}x^{31}+\alpha^{225}x^{30}+\alpha^{46}x^{29}+\alpha^{13}x^{28}+\alpha^{32}x^{27}+\alpha^{160}x^{26}$ $+\alpha^{126}x^{25}+\alpha^{209}x^{24}+\alpha^{130}x^{23}+\alpha^{160}x^{22}+\alpha^{242}x^{21}+\alpha^{215}x^{20}$ $+\alpha^{242}x^{19}+\alpha^{75}x^{18}+\alpha^{77}x^{17}+\alpha^{42}x^{16}+\alpha^{189}x^{15}+\alpha^{32}x^{14}$ $+\alpha^{113}x^{13}+\alpha^{65}x^{12}+\alpha^{124}x^{11}+\alpha^{69}x^{10}+\alpha^{228}x^{9}+\alpha^{114}x^{8}$ $+\alpha^{235}x^{7}+\alpha^{175}x^{6}+\alpha^{124}x^{5}+\alpha^{170}x^{4}+\alpha^{215}x^{3}+\alpha^{232}x^{2}$ $+\alpha^{133}x+\alpha^{205}$
52	$x^{52}+\alpha^{116}x^{51}+\alpha^{50}x^{50}+\alpha^{86}x^{49}+\alpha^{186}x^{48}+\alpha^{50}x^{47}+\alpha^{220}x^{46}$ $+\alpha^{251}x^{45}+\alpha^{89}x^{44}+\alpha^{192}x^{43}+\alpha^{46}x^{42}+\alpha^{86}x^{41}+\alpha^{127}x^{40}$ $+\alpha^{124}x^{39}+\alpha^{19}x^{38}+\alpha^{184}x^{37}+\alpha^{233}x^{36}+\alpha^{151}x^{35}+\alpha^{215}x^{34}$ $+\alpha^{22}x^{33}+\alpha^{14}x^{32}+\alpha^{59}x^{31}+\alpha^{145}x^{30}+\alpha^{37}x^{29}+\alpha^{242}x^{28}$ $+\alpha^{203}x^{27}+\alpha^{134}x^{26}+\alpha^{254}x^{25}+\alpha^{89}x^{24}+\alpha^{190}x^{23}+\alpha^{94}x^{22}$ $+\alpha^{59}x^{21}+\alpha^{65}x^{20}+\alpha^{124}x^{19}+\alpha^{113}x^{18}+\alpha^{100}x^{17}+\alpha^{233}x^{16}$ $+\alpha^{235}x^{15}+\alpha^{121}x^{14}+\alpha^{22}x^{13}+\alpha^{76}x^{12}+\alpha^{86}x^{11}+\alpha^{97}x^{10}$ $+\alpha^{39}x^{9}+\alpha^{242}x^{8}+\alpha^{200}x^{7}+\alpha^{220}x^{6}+\alpha^{101}x^{5}+\alpha^{33}x^{4}+\alpha^{239}x^{3}$ $+\alpha^{254}x^{2}+\alpha^{116}x+\alpha^{51}$

54	$x^{54}+\alpha^{183}x^{53}+\alpha^{26}x^{52}+\alpha^{201}x^{51}+\alpha^{87}x^{50}+\alpha^{210}x^{49}+\alpha^{221}x^{48}$ $+\alpha^{113}x^{47}+\alpha^{21}x^{46}+\alpha^{46}x^{45}+\alpha^{65}x^{44}+\alpha^{45}x^{43}+\alpha^{50}x^{42}$ $+\alpha^{238}x^{41}+\alpha^{184}x^{40}+\alpha^{249}x^{39}+\alpha^{225}x^{38}+\alpha^{102}x^{37}+\alpha^{58}x^{36}$ $+\alpha^{209}x^{35}+\alpha^{218}x^{34}+\alpha^{109}x^{33}+\alpha^{165}x^{32}+\alpha^{26}x^{31}+\alpha^{95}x^{30}$ $+\alpha^{184}x^{29}+\alpha^{192}x^{28}+\alpha^{52}x^{27}+\alpha^{245}x^{26}+\alpha^{35}x^{25}+\alpha^{254}x^{24}$ $+\alpha^{238}x^{23}+\alpha^{175}x^{22}+\alpha^{172}x^{21}+\alpha^{79}x^{20}+\alpha^{123}x^{19}+\alpha^{25}x^{18}$ $+\alpha^{122}x^{17}+\alpha^{43}x^{16}+\alpha^{120}x^{15}+\alpha^{108}x^{14}+\alpha^{215}x^{13}+\alpha^{80}x^{12}$ $+\alpha^{128}x^{11}+\alpha^{201}x^{10}+\alpha^{235}x^{9}+\alpha^{8}x^{8}+\alpha^{153}x^{7}+\alpha^{59}x^{6}+\alpha^{101}x^{5}$ $+\alpha^{31}x^{4}+\alpha^{198}x^{3}+\alpha^{76}x^{2}+\alpha^{31}x+\alpha^{156}$
56	$x^{56}+\alpha^{106}x^{55}+\alpha^{120}x^{54}+\alpha^{107}x^{53}+\alpha^{157}x^{52}+\alpha^{164}x^{51}+\alpha^{216}x^{50}$ $+\alpha^{112}x^{49}+\alpha^{116}x^{48}+\alpha^{2}x^{47}+\alpha^{91}x^{46}+\alpha^{248}x^{45}+\alpha^{163}x^{44}$ $+\alpha^{36}x^{43}+\alpha^{201}x^{42}+\alpha^{202}x^{41}+\alpha^{229}x^{40}+\alpha^{6}x^{39}+\alpha^{144}x^{38}$ $+\alpha^{254}x^{37}+\alpha^{155}x^{36}+\alpha^{135}x^{35}+\alpha^{208}x^{34}+\alpha^{170}x^{33}+\alpha^{209}x^{32}$ $+\alpha^{12}x^{31}+\alpha^{139}x^{30}+\alpha^{127}x^{29}+\alpha^{142}x^{28}+\alpha^{182}x^{27}+\alpha^{249}x^{26}$ $+\alpha^{177}x^{25}+\alpha^{174}x^{24}+\alpha^{190}x^{23}+\alpha^{28}x^{22}+\alpha^{10}x^{21}+\alpha^{85}x^{20}$ $+\alpha^{239}x^{19}+\alpha^{184}x^{18}+\alpha^{101}x^{17}+\alpha^{124}x^{16}+\alpha^{152}x^{15}+\alpha^{206}x^{14}$ $+\alpha^{96}x^{13}+\alpha^{23}x^{12}+\alpha^{163}x^{11}+\alpha^{61}x^{10}+\alpha^{27}x^{9}+\alpha^{196}x^{8}$ $+\alpha^{247}x^{7}+\alpha^{151}x^{6}+\alpha^{154}x^{5}+\alpha^{202}x^{4}+\alpha^{207}x^{3}+\alpha^{20}x^{2}+\alpha^{61}x+\alpha^{10}$
58	$x^{58}+\alpha^{82}x^{57}+\alpha^{116}x^{56}+\alpha^{26}x^{55}+\alpha^{247}x^{54}+\alpha^{66}x^{53}+\alpha^{27}x^{52}$ $+\alpha^{62}x^{51}+\alpha^{107}x^{50}+\alpha^{252}x^{49}+\alpha^{182}x^{48}+\alpha^{200}x^{47}+\alpha^{185}x^{46}$ $+\alpha^{235}x^{45}+\alpha^{55}x^{44}+\alpha^{251}x^{43}+\alpha^{242}x^{42}+\alpha^{210}x^{41}+\alpha^{144}x^{40}$ $+\alpha^{154}x^{39}+\alpha^{237}x^{38}+\alpha^{176}x^{37}+\alpha^{141}x^{36}+\alpha^{192}x^{35}+\alpha^{248}x^{34}$ $+\alpha^{152}x^{33}+\alpha^{249}x^{32}+\alpha^{206}x^{31}+\alpha^{85}x^{30}+\alpha^{253}x^{29}+\alpha^{142}x^{28}$ $+\alpha^{65}x^{27}+\alpha^{165}x^{26}+\alpha^{125}x^{25}+\alpha^{23}x^{24}+\alpha^{24}x^{23}+\alpha^{30}x^{22}$ $+\alpha^{122}x^{21}+\alpha^{240}x^{20}+\alpha^{214}x^{19}+\alpha^{6}x^{18}+\alpha^{129}x^{17}+\alpha^{218}x^{16}$ $+\alpha^{29}x^{15}+\alpha^{145}x^{14}+\alpha^{127}x^{13}+\alpha^{134}x^{12}+\alpha^{206}x^{11}+\alpha^{245}x^{10}$ $+\alpha^{117}x^{9}+\alpha^{29}x^{8}+\alpha^{41}x^{7}+\alpha^{63}x^{6}+\alpha^{159}x^{5}+\alpha^{142}x^{4}+\alpha^{233}x^{3}$ $+\alpha^{125}x^{2}+\alpha^{148}x+\alpha^{123}$
60	$x^{60}+\alpha^{107}x^{59}+\alpha^{140}x^{58}+\alpha^{26}x^{57}+\alpha^{12}x^{56}+\alpha^{9}x^{55}+\alpha^{141}x^{54}$ $+\alpha^{243}x^{53}+\alpha^{197}x^{52}+\alpha^{226}x^{51}+\alpha^{197}x^{50}+\alpha^{219}x^{49}+\alpha^{45}x^{48}$ $+\alpha^{211}x^{47}+\alpha^{101}x^{46}+\alpha^{219}x^{45}+\alpha^{120}x^{44}+\alpha^{28}x^{43}+\alpha^{181}x^{42}$ $+\alpha^{127}x^{41}+\alpha^{6}x^{40}+\alpha^{100}x^{39}+\alpha^{247}x^{38}+\alpha^{2}x^{37}+\alpha^{205}x^{36}$ $+\alpha^{198}x^{35}+\alpha^{57}x^{34}+\alpha^{115}x^{33}+\alpha^{219}x^{32}+\alpha^{101}x^{31}+\alpha^{109}x^{30}$ $+\alpha^{160}x^{29}+\alpha^{82}x^{28}+\alpha^{37}x^{27}+\alpha^{38}x^{26}+\alpha^{238}x^{25}+\alpha^{49}x^{24}$ $+\alpha^{160}x^{23}+\alpha^{209}x^{22}+\alpha^{121}x^{21}+\alpha^{86}x^{20}+\alpha^{11}x^{19}+\alpha^{124}x^{18}$ $+\alpha^{30}x^{17}+\alpha^{181}x^{16}+\alpha^{84}x^{15}+\alpha^{25}x^{14}+\alpha^{194}x^{13}+\alpha^{87}x^{12}$ $+\alpha^{65}x^{11}+\alpha^{102}x^{10}+\alpha^{190}x^{9}+\alpha^{220}x^{8}+\alpha^{70}x^{7}+\alpha^{27}x^{6}+\alpha^{209}x^{5}$ $+\alpha^{16}x^{4}+\alpha^{89}x^{3}+\alpha^{7}x^{2}+\alpha^{33}x+\alpha^{240}$

62	$x^{62}+\alpha^{65}x^{61}+\alpha^{202}x^{60}+\alpha^{113}x^{59}+\alpha^{98}x^{58}+\alpha^{71}x^{57}+\alpha^{223}x^{56}$ $+\alpha^{248}x^{55}+\alpha^{118}x^{54}+\alpha^{214}x^{53}+\alpha^{94}x^{52}+x^{51}+\alpha^{122}x^{50}$ $+\alpha^{37}x^{49}+\alpha^{23}x^{48}+\alpha^{2}x^{47}+\alpha^{228}x^{46}+\alpha^{58}x^{45}+\alpha^{121}x^{44}$ $+\alpha^{7}x^{43}+\alpha^{105}x^{42}+\alpha^{135}x^{41}+\alpha^{78}x^{40}+\alpha^{243}x^{39}+\alpha^{118}x^{38}$ $+\alpha^{70}x^{37}+\alpha^{76}x^{36}+\alpha^{223}x^{35}+\alpha^{89}x^{34}+\alpha^{72}x^{33}+\alpha^{50}x^{32}$ $+\alpha^{70}x^{31}+\alpha^{111}x^{30}+\alpha^{194}x^{29}+\alpha^{17}x^{28}+\alpha^{212}x^{27}+\alpha^{126}x^{26}$ $+\alpha^{181}x^{25}+\alpha^{35}x^{24}+\alpha^{221}x^{23}+\alpha^{117}x^{22}+\alpha^{235}x^{21}+\alpha^{11}x^{20}$ $+\alpha^{229}x^{19}+\alpha^{149}x^{18}+\alpha^{147}x^{17}+\alpha^{123}x^{16}+\alpha^{213}x^{15}+\alpha^{40}x^{14}$ $+\alpha^{115}x^{13}+\alpha^{6}x^{12}+\alpha^{200}x^{11}+\alpha^{100}x^{10}+\alpha^{26}x^{9}+\alpha^{246}x^{8}$ $+\alpha^{182}x^{7}+\alpha^{218}x^{6}+\alpha^{127}x^{5}+\alpha^{215}x^{4}+\alpha^{36}x^{3}+\alpha^{186}x^{2}+\alpha^{110}x+\alpha^{106}$
64	$x^{64}+\alpha^{45}x^{63}+\alpha^{51}x^{62}+\alpha^{175}x^{61}+\alpha^{9}x^{60}+\alpha^{7}x^{59}+\alpha^{158}x^{58}$ $+\alpha^{159}x^{57}+\alpha^{49}x^{56}+\alpha^{68}x^{55}+\alpha^{119}x^{54}+\alpha^{92}x^{53}+\alpha^{123}x^{52}$ $+\alpha^{177}x^{51}+\alpha^{204}x^{50}+\alpha^{187}x^{49}+\alpha^{254}x^{48}+\alpha^{200}x^{47}+\alpha^{78}x^{46}$ $+\alpha^{141}x^{45}+\alpha^{149}x^{44}+\alpha^{119}x^{43}+\alpha^{26}x^{42}+\alpha^{127}x^{41}+\alpha^{53}x^{40}$ $+\alpha^{160}x^{39}+\alpha^{93}x^{38}+\alpha^{199}x^{37}+\alpha^{212}x^{36}+\alpha^{29}x^{35}+\alpha^{24}x^{34}$ $+\alpha^{145}x^{33}+\alpha^{156}x^{32}+\alpha^{208}x^{31}+\alpha^{150}x^{30}+\alpha^{218}x^{29}+\alpha^{209}x^{28}$ $+\alpha^{4}x^{27}+\alpha^{216}x^{26}+\alpha^{91}x^{25}+\alpha^{47}x^{24}+\alpha^{184}x^{23}+\alpha^{146}x^{22}$ $+\alpha^{47}x^{21}+\alpha^{140}x^{20}+\alpha^{195}x^{19}+\alpha^{195}x^{18}+\alpha^{125}x^{17}+\alpha^{242}x^{16}$ $+\alpha^{238}x^{15}+\alpha^{63}x^{14}+\alpha^{99}x^{13}+\alpha^{108}x^{12}+\alpha^{140}x^{11}+\alpha^{230}x^{10}$ $+\alpha^{242}x^{9}+\alpha^{31}x^{8}+\alpha^{204}x^{7}+\alpha^{11}x^{6}+\alpha^{178}x^{5}+\alpha^{243}x^{4}+\alpha^{217}x^{3}$ $+\alpha^{156}x^{2}+\alpha^{213}x+\alpha^{231}$
66	$x^{66}+\alpha^{5}x^{65}+\alpha^{118}x^{64}+\alpha^{222}x^{63}+\alpha^{180}x^{62}+\alpha^{136}x^{61}+\alpha^{136}x^{60}$ $+\alpha^{162}x^{59}+\alpha^{51}x^{58}+\alpha^{46}x^{57}+\alpha^{117}x^{56}+\alpha^{13}x^{55}+\alpha^{215}x^{54}$ $+\alpha^{81}x^{53}+\alpha^{17}x^{52}+\alpha^{139}x^{51}+\alpha^{247}x^{50}+\alpha^{197}x^{49}+\alpha^{171}x^{48}$ $+\alpha^{95}x^{47}+\alpha^{173}x^{46}+\alpha^{65}x^{45}+\alpha^{137}x^{44}+\alpha^{178}x^{43}+\alpha^{68}x^{42}$ $+\alpha^{111}x^{41}+\alpha^{95}x^{40}+\alpha^{101}x^{39}+\alpha^{41}x^{38}+\alpha^{72}x^{37}+\alpha^{214}x^{36}$ $+\alpha^{169}x^{35}+\alpha^{197}x^{34}+\alpha^{95}x^{33}+\alpha^{7}x^{32}+\alpha^{44}x^{31}+\alpha^{154}x^{30}$ $+\alpha^{77}x^{29}+\alpha^{111}x^{28}+\alpha^{236}x^{27}+\alpha^{40}x^{26}+\alpha^{121}x^{25}+\alpha^{143}x^{24}$ $+\alpha^{63}x^{23}+\alpha^{87}x^{22}+\alpha^{80}x^{21}+\alpha^{253}x^{20}+\alpha^{240}x^{19}+\alpha^{126}x^{18}$ $+\alpha^{217}x^{17}+\alpha^{77}x^{16}+\alpha^{34}x^{15}+\alpha^{232}x^{14}+\alpha^{106}x^{13}+\alpha^{50}x^{12}$ $+\alpha^{168}x^{11}+\alpha^{82}x^{10}+\alpha^{76}x^{9}+\alpha^{146}x^{8}+\alpha^{67}x^{7}+\alpha^{106}x^{6}$ $+\alpha^{171}x^{5}+\alpha^{25}x^{4}+\alpha^{132}x^{3}+\alpha^{93}x^{2}+\alpha^{45}x+\alpha^{105}$
68	$x^{68}+\alpha^{247}x^{67}+\alpha^{159}x^{66}+\alpha^{223}x^{65}+\alpha^{33}x^{64}+\alpha^{224}x^{63}+\alpha^{93}x^{62}$ $+\alpha^{77}x^{61}+\alpha^{70}x^{60}+\alpha^{90}x^{59}+\alpha^{160}x^{58}+\alpha^{32}x^{57}+\alpha^{254}x^{56}$ $+\alpha^{43}x^{55}+\alpha^{150}x^{54}+\alpha^{84}x^{53}+\alpha^{101}x^{52}+\alpha^{190}x^{51}+\alpha^{205}x^{50}$ $+\alpha^{133}x^{49}+\alpha^{52}x^{48}+\alpha^{60}x^{47}+\alpha^{202}x^{46}+\alpha^{165}x^{45}+\alpha^{220}x^{44}$ $+\alpha^{203}x^{43}+\alpha^{151}x^{42}+\alpha^{93}x^{41}+\alpha^{84}x^{40}+\alpha^{15}x^{39}+\alpha^{84}x^{38}$ $+\alpha^{253}x^{37}+\alpha^{173}x^{36}+\alpha^{160}x^{35}+\alpha^{89}x^{34}+\alpha^{227}x^{33}+\alpha^{52}x^{32}$ $+\alpha^{199}x^{31}+\alpha^{97}x^{30}+\alpha^{95}x^{29}+\alpha^{231}x^{28}+\alpha^{52}x^{27}+\alpha^{177}x^{26}$ $+\alpha^{41}x^{25}+\alpha^{125}x^{24}+\alpha^{137}x^{23}+\alpha^{241}x^{22}+\alpha^{166}x^{21}+\alpha^{225}x^{20}$ $+\alpha^{118}x^{19}+\alpha^{2}x^{18}+\alpha^{54}x^{17}+\alpha^{32}x^{16}+\alpha^{82}x^{15}+\alpha^{215}x^{14}$ $+\alpha^{175}x^{13}+\alpha^{198}x^{12}+\alpha^{43}x^{11}+\alpha^{238}x^{10}+\alpha^{235}x^{9}+\alpha^{27}x^{8}$ $+\alpha^{101}x^{7}+\alpha^{184}x^{6}+\alpha^{127}x^{5}+\alpha^{3}x^{4}+\alpha^{5}x^{3}+\alpha^{8}x^{2}+\alpha^{163}x+\alpha^{238}$

Tabel A.5 Konversi Nilai Koefisien $\boldsymbol{a}$ dan Nilai Integer Untuk GF(2^8) [2]

Exponent of α	Integer	Integer	Exponent of α	Exponent of α	Integer	Integer	Exponent of α
0	1	-	-	22	234	22	239
1	2	1	0	23	201	23	129
2	4	2	1	24	143	24	28
3	8	3	25	25	3	25	193
4	16	4	2	26	6	26	105
5	32	5	50	27	12	27	248
6	64	6	26	28	24	28	200
7	128	7	198	29	48	29	8
8	29	8	3	30	96	30	76
9	58	9	223	31	192	31	113
10	116	10	51	32	157	32	5
11	232	11	238	33	39	33	138
12	205	12	27	34	78	34	101
13	135	13	104	35	156	35	47
14	19	14	199	36	37	36	225
15	38	15	75	37	74	37	36
16	76	16	4	38	148	38	15
17	152	17	100	39	53	39	33
18	45	18	224	40	106	40	53
19	90	19	14	41	212	41	147
20	180	20	52	42	181	42	142
21	117	21	141	43	119	43	218

Exponent of α	Integer	Integer	Exponent of α	Exponent of α	Integer	Integer	Exponent of α
44	238	44	240	67	194	67	98
45	193	45	18	68	153	68	102
46	159	46	130	69	47	69	221
47	35	47	69	70	94	70	48
48	70	48	29	71	188	71	253
49	140	49	181	72	101	72	226
50	5	50	194	73	202	73	152
51	10	51	125	74	137	74	37
52	20	52	106	75	15	75	179
53	40	53	39	76	30	76	16
54	80	54	249	77	60	77	145
55	160	55	185	78	120	78	34
56	93	56	201	79	240	79	136
57	186	57	154	80	253	80	54
58	105	58	9	81	231	81	208
59	210	59	120	82	211	82	148
60	185	60	77	83	187	83	206
61	111	61	228	84	107	84	143
62	222	62	114	85	214	85	150
63	161	63	166	86	177	86	219
64	95	64	6	87	127	87	189
65	190	65	191	88	254	88	241
66	97	66	139	89	225	89	210

Exponent of α	Integer	Integer	Exponent of α	Exponent of α	Integer	Integer	Exponent of α
90	223	90	19	113	31	113	94
91	163	91	92	114	62	114	155
92	91	92	131	115	124	115	159
93	182	93	56	116	248	116	10
94	113	94	70	117	237	117	21
95	226	95	64	118	199	118	121
96	217	96	30	119	147	119	43
97	175	97	66	120	59	120	78
98	67	98	182	121	118	121	212
99	134	99	163	122	236	122	229
100	17	100	195	123	197	123	172
101	34	101	72	124	151	124	115
102	68	102	126	125	51	125	243
103	136	103	110	126	102	126	167
104	13	104	107	127	204	127	87
105	26	105	58	128	133	128	7
106	52	106	40	129	23	129	112
107	104	107	84	130	46	130	192
108	208	108	250	131	92	131	247
109	189	109	133	132	184	132	140
110	103	110	186	133	109	133	128
111	206	111	61	134	218	134	99
112	129	112	202	135	169	135	13

Exponent of α	Integer	Integer	Exponent of α	Exponent of α	Integer	Integer	Exponent of α
136	79	136	103	159	115	159	46
137	158	137	74	160	230	160	55
138	33	138	222	161	209	161	63
139	66	139	237	162	191	162	209
140	132	140	49	163	99	163	91
141	21	141	197	164	198	164	149
142	42	142	254	165	145	165	188
143	84	143	24	166	63	166	207
144	168	144	227	167	126	167	205
145	77	145	165	168	252	168	144
146	154	146	153	169	229	169	135
147	41	147	119	170	215	170	151
148	82	148	38	171	179	171	178
149	164	149	184	172	123	172	220
150	85	150	180	173	246	173	252
151	170	151	124	174	241	174	190
152	73	152	17	175	255	175	97
153	146	153	68	176	227	176	242
154	57	154	146	177	219	177	86
155	114	155	217	178	171	178	211
156	228	156	35	179	75	179	171
157	213	157	32	180	150	180	20
158	183	158	137	181	49	181	42

Exponent of α	Integer	Integer	Exponent of α	Exponent of α	Integer	Integer	Exponent of α
182	98	182	93	205	167	205	12
183	196	183	158	206	83	206	111
184	149	184	132	207	166	207	246
185	55	185	60	208	81	208	108
186	110	186	57	209	162	209	161
187	220	187	83	210	89	210	59
188	165	188	71	211	178	211	82
189	87	189	109	212	121	212	41
190	174	190	65	213	242	213	157
191	65	191	162	214	249	214	85
192	130	192	31	215	239	215	170
193	25	193	45	216	195	216	251
194	50	194	67	217	155	217	96
195	100	195	216	218	43	218	134
196	200	196	183	219	86	219	177
197	141	197	123	220	172	220	187
198	7	198	164	221	69	221	204
199	14	199	118	222	138	222	62
200	28	200	196	223	9	223	90
201	56	201	23	224	18	224	203
202	112	202	73	225	36	225	89
203	224	203	236	226	72	226	95
204	221	204	127	227	144	227	176

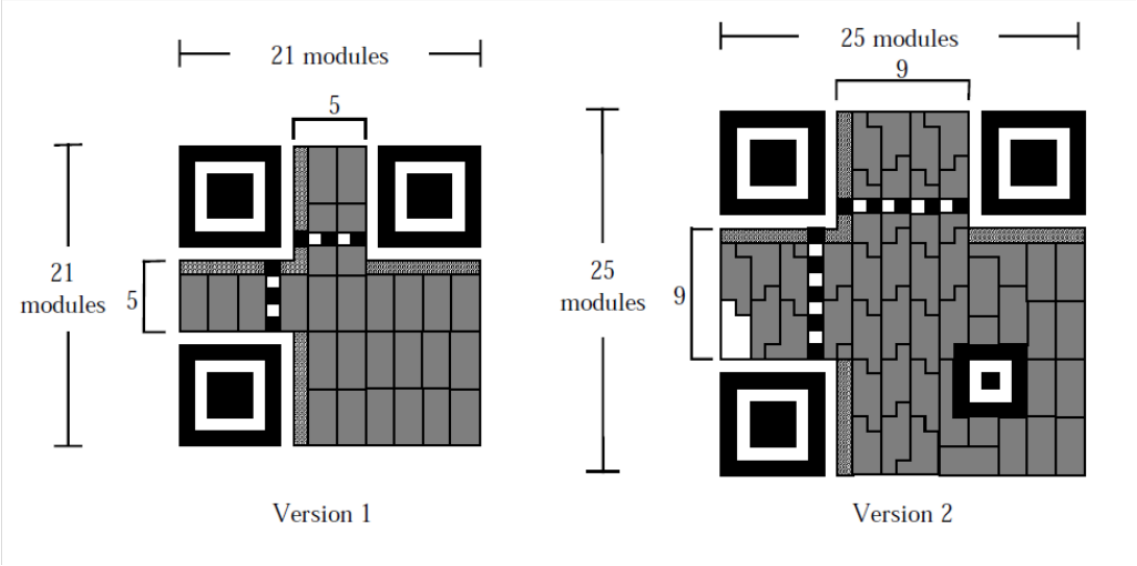
Exponent of α	Integer	Integer	Exponent of α	Exponent of α	Integer	Integer	Exponent of α
228	61	228	156	242	176	242	213
229	122	229	169	243	125	243	233
230	244	230	160	244	250	244	230
231	245	231	81	245	233	245	231
232	247	232	11	246	207	246	173
233	243	233	245	247	131	247	232
234	251	234	22	248	27	248	116
235	235	235	235	249	54	249	214
236	203	236	122	250	108	250	244
237	139	237	117	251	216	251	234
238	11	238	44	252	173	252	168
239	22	239	215	253	71	253	80
240	44	240	79	254	142	254	80
241	88	241	174	255	1	255	175

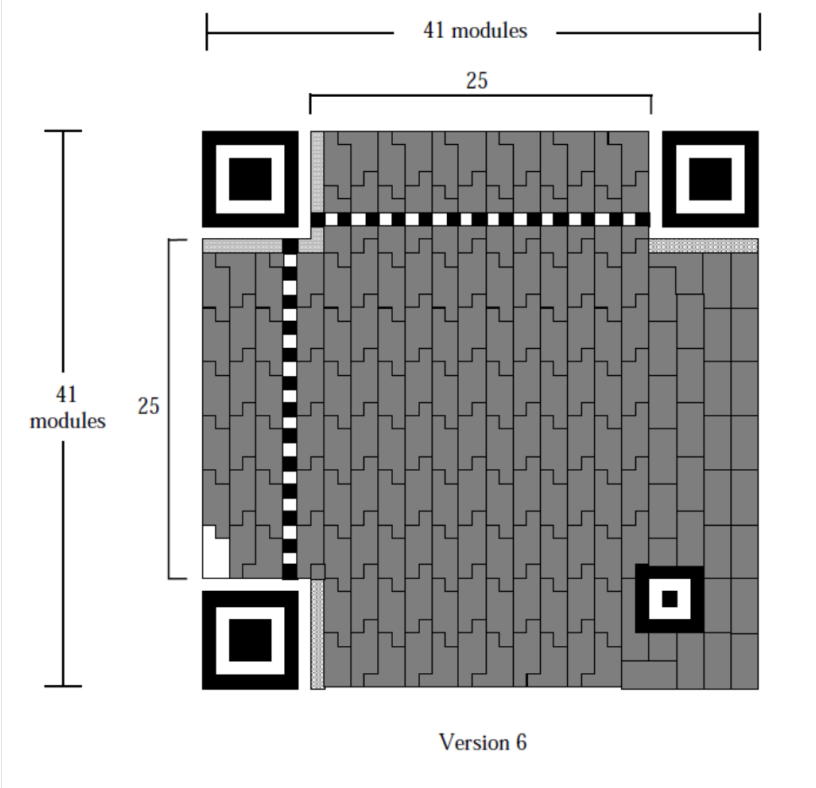
Tabel A.6 *Format Information*[5]

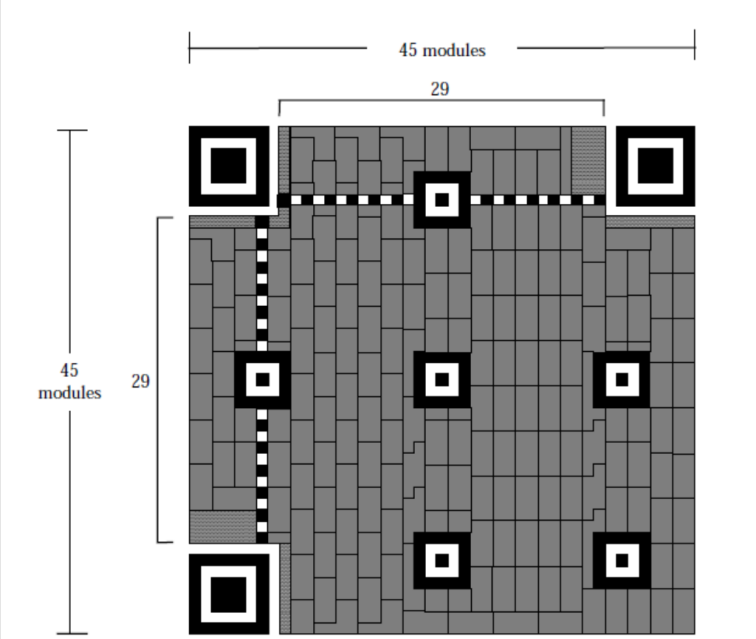
ECC Level	Mask Pattern	Type Information Bits
L	0	111011111000100
L	1	111001011110011
L	2	111110110101010
L	3	111100010011101
L	4	110011000101111
L	5	110001100011000
L	6	110110001000001
L	7	110100101110110
M	0	101010000010010
M	1	101000100100101
M	2	101111001111100
M	3	101101101001011
M	4	100010111111001
M	5	100000011001110
M	6	100111110010111
M	7	100101010100000
Q	0	011010101011111
Q	1	011000001101000
Q	2	011111100110001
Q	3	011101000000110
Q	4	010010010110100
Q	5	010000110000011
Q	6	010111011011010
Q	7	010101111101101
H	0	001011010001001
H	1	001001110111110
H	2	001110011100111
H	3	001100111010000
H	4	000011101100010
H	5	000001001010101
H	6	000110100001100
H	7	000100000111011

LAMPIRAN B

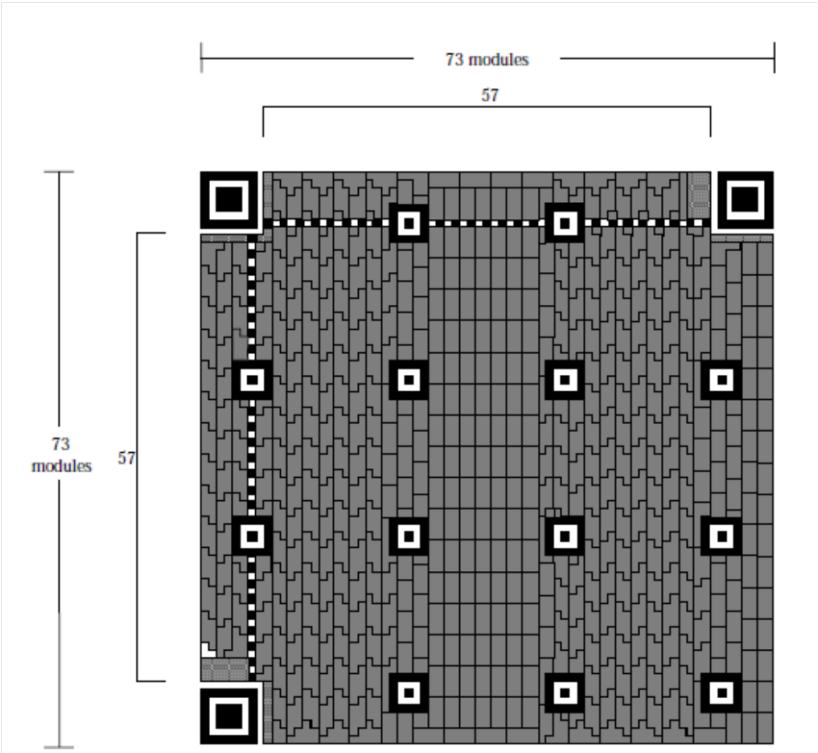
Gambar B.1 Perbedaan Ukuran pada QR Code[2]





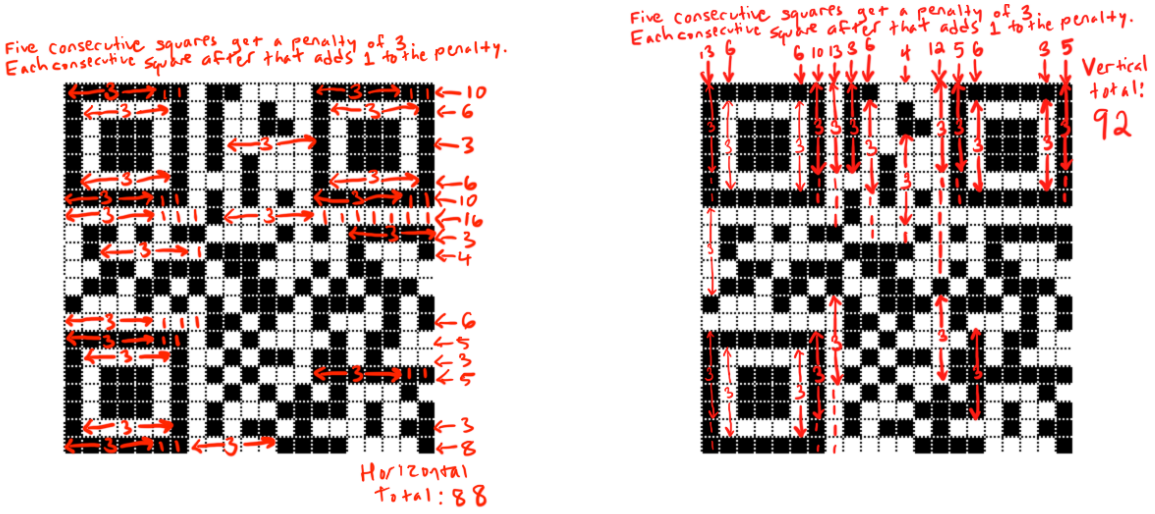


Version 7

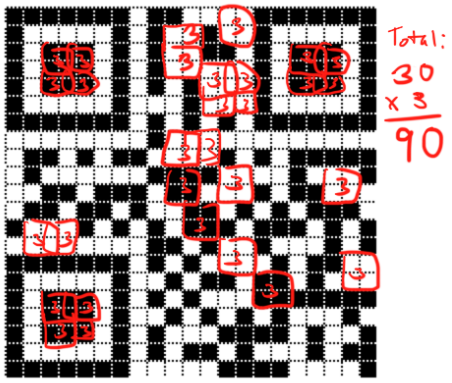


Version 14

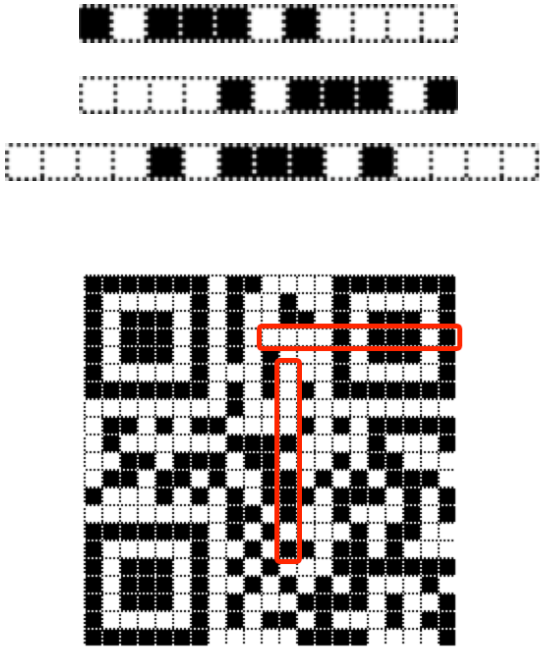
Gambar B.2 Perhitungan *Down Point* berdasarkan Keempat Nilai Penalti[5]



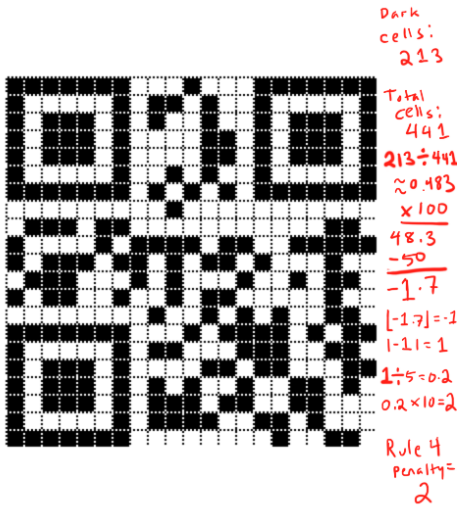
Perhitungan Penalti 1



Perhitungan Penalti 2



Perhitungan Penalti 3



Perhitungan Penalti 4

LAMPIRAN C

Listing Program

- **Mainmenu.java**

```
public class Mainmenu extends Activity{
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        //Memanggil layout yang pertama sekali ditampilkan
        setContentView(R.layout.mainmenu);
    }
       // Bagian ini untuk button pada mainmenu
    public void allclickhandler(View view) {

           Intent i = null;

              switch (view.getId()) {

                  case R.id.btn_mainmenu_qrcreate:
                            i = new Intent(this, ShareActivity.class);
                            startActivity(i);
                            break;
                  case R.id.btn_mainmenu_qrscan:
                            i = new Intent(this, CaptureActivity.class);
                            startActivity(i);
                            break;
```

- **CreateQR.java**

```
private final View.OnKeyListener textListener = new View.OnKeyListener() {
    @Override
    public boolean onKey(View view, int keyCode, KeyEvent event) {
      if (keyCode == KeyEvent.KEYCODE_ENTER && event.getAction() ==
KeyEvent.ACTION_DOWN) {
        String text = ((TextView) view).getText().toString();
        if (text != null && text.length() > 0) {
          launchSearch(text);
        }
        return true;
      }
      return false;
    }
  };
```

Proses Encode

- **Encode.java**

```
public final class Encoder {
//inisialisasi nilai alphanumerik
private static final int[] ALPHANUMERIC_TABLE = {
      -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
//0x00-0x0f
      -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
//0x10-0x1f
      36, -1, -1, -1, 37, 38, -1, -1, -1, -1, 39, 40, -1, 41, 42, 43,
//0x20-0x2f
      0,   1,  2,  3,  4,  5,  6,  7,  8,  9, 44, -1, -1, -1, -1, -1,
//0x30-0x3f
      -1, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
//0x40-0x4f
      25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, -1, -1, -1, -1, -1,
//0x50-0x5f
  };
```

```
private Encoder() {
  }
  // perhitungan down point keempat penalty
  private static int calculateMaskPenalty(ByteMatrix matrix) {
    return MaskUtil.applyMaskPenaltyRule1(matrix)
        + MaskUtil.applyMaskPenaltyRule2(matrix)
        + MaskUtil.applyMaskPenaltyRule3(matrix)
        + MaskUtil.applyMaskPenaltyRule4(matrix);
  }

public static QRCode encode(String content, ErrorCorrectionLevel ecLevel) throws
WriterException {
    return encode(content, ecLevel, null);
  }

  public static QRCode encode(String content,
                              ErrorCorrectionLevel ecLevel,
                              Map<EncodeHintType,?> hints) throws WriterException {
String encoding = hints == null ? null : (String)
hints.get(EncodeHintType.CHARACTER_SET);
    if (encoding == null) {
      encoding = DEFAULT_BYTE_MODE_ENCODING;
    }
// Mode QR code
    Mode mode = chooseMode(content, encoding);
    appendAlphanumericBytes(content, bits);

// mementukan urutan aliran bit

    int provisionalBitsNeeded = headerBits.getSize()
        + mode.getCharacterCountBits(Version.getVersionForNumber(1))
        + dataBits.getSize();
    Version provisionalVersion = chooseVersion(provisionalBitsNeeded, ecLevel);
int bitsNeeded = headerBits.getSize()
        + mode.getCharacterCountBits(provisionalVersion)
        + dataBits.getSize();
    Version version = chooseVersion(bitsNeeded, ecLevel);

    BitArray headerAndDataBits = new BitArray();
    headerAndDataBits.appendBitArray(headerBits);

// menentukan indikator jumlah karakter
    int numLetters = mode == Mode.BYTE ? dataBits.getSizeInBytes() : content.length();
    appendLengthInfo(numLetters, version, mode, headerAndDataBits);

// menentukan data codeword
    headerAndDataBits.appendBitArray(dataBits);

    Version.ECBlocks ecBlocks = version.getECBlocksForLevel(ecLevel);
    int numDataBytes = version.getTotalCodewords() - ecBlocks.getTotalECCodewords();

    // menambah terminator
    terminateBits(numDataBytes, headerAndDataBits);
QRCode qrCode = new QRCode();

    qrCode.setECLevel(ecLevel);
    qrCode.setMode(mode);




    // menambahkan error correction codeword
```

```
    BitArray finalBits = interleaveWithECBytes(headerAndDataBits,
                                               version.getTotalCodewords(),
                                               numDataBytes,
                                               ecBlocks.getNumBlocks());

    QRCode qrCode = new QRCode();

    qrCode.setECLevel(ecLevel);
    qrCode.setMode(mode);
    qrCode.setVersion(version);

    // menerapkan mask pattern
    int dimension = version.getDimensionForVersion();
    ByteMatrix matrix = new ByteMatrix(dimension, dimension);
    int maskPattern = chooseMaskPattern(finalBits, ecLevel, version, matrix);
    qrCode.setMaskPattern(maskPattern);

       // alokasi bit ke dalam Module
    MatrixUtil.buildMatrix(finalBits, ecLevel, version, maskPattern, matrix);
    qrCode.setMatrix(matrix);

    return qrCode;
  }

       // menentukan nilai alphanumerik
static int getAlphanumericCode(int code) {
    if (code < ALPHANUMERIC_TABLE.length) {
      return ALPHANUMERIC_TABLE[code];
    }
    return -1;
  }

public static Mode chooseMode(String content) {
    return chooseMode(content, null);
  }
// memilih mask pattern
private static int chooseMaskPattern(BitArray bits,
                                     ErrorCorrectionLevel ecLevel,
                                     Version version,
                                     ByteMatrix matrix) throws WriterException {

    int minPenalty = Integer.MAX_VALUE;  // dipilih down point terkecil
    int bestMaskPattern = -1;
    for (int maskPattern = 0; maskPattern < QRCode.NUM_MASK_PATTERNS; maskPattern++) {
      MatrixUtil.buildMatrix(bits, ecLevel, version, maskPattern, matrix);
      int penalty = calculateMaskPenalty(matrix);
      if (penalty < minPenalty) {
        minPenalty = penalty;
        bestMaskPattern = maskPattern;
      }
    }
    return bestMaskPattern;
  }

// versi QR code.
private static Version chooseVersion(int numInputBits, ErrorCorrectionLevel ecLevel)
throws WriterException {

    for (int versionNum = 1; versionNum <= 5; versionNum++) {
      Version version = Version.getVersionForNumber(versionNum);

      int numBytes = version.getTotalCodewords();
```

```
      Version.ECBlocks ecBlocks = version.getECBlocksForLevel(ecLevel);
      int numEcBytes = ecBlocks.getTotalECCodewords();

      int numDataBytes = numBytes - numEcBytes;
      int totalInputBytes = (numInputBits + 7) / 8;
      if (numDataBytes >= totalInputBytes) {
        return version;
      }
    }
    throw new WriterException("Data too big");
  }

// penambahan terminator
static void terminateBits(int numDataBytes, BitArray bits) throws WriterException {
    int capacity = numDataBytes << 3;
    if (bits.getSize() > capacity) {
      throw new WriterException("data bits cannot fit in the QR Code" + bits.getSize()
+ " > " +
          capacity);
    }
    for (int i = 0; i < 4 && bits.getSize() < capacity; ++i) {
      bits.appendBit(false);
    }

// pengelompokkan aliran bit menjadi codeword dan penambahan bit "0" untuk codeword
kurang dari 8 bit
    int numBitsInLastByte = bits.getSize() & 0x07;
    if (numBitsInLastByte > 0) {
      for (int i = numBitsInLastByte; i < 8; i++) {
        bits.appendBit(false);
      }
    }

// penambahan padding codeword
    int numPaddingBytes = numDataBytes - bits.getSizeInBytes();
    for (int i = 0; i < numPaddingBytes; ++i) {
      bits.appendBits((i & 0x01) == 0 ? 0xEC : 0x11, 8);
    }
    if (bits.getSize() != capacity) {
          }
  }

// menempatkan error correction
static byte[] generateECBytes(byte[] dataBytes, int numEcBytesInBlock) {
    int numDataBytes = dataBytes.length;
    int[] toEncode = new int[numDataBytes + numEcBytesInBlock];
    for (int i = 0; i < numDataBytes; i++) {
      toEncode[i] = dataBytes[i] & 0xFF;
    }
    new ReedSolomonEncoder(GenericGF.QR_CODE_FIELD_256).encode(toEncode,
numEcBytesInBlock);

    byte[] ecBytes = new byte[numEcBytesInBlock];
    for (int i = 0; i < numEcBytesInBlock; i++) {
      ecBytes[i] = (byte) toEncode[numDataBytes + i];
    }
    return ecBytes;
  }



// encodedata masukan
```

```
static void appendAlphanumericBytes(CharSequence content, BitArray bits) throws
WriterException {
    int length = content.length();
    int i = 0;
    while (i < length) {
      int code1 = getAlphanumericCode(content.charAt(i));
      if (code1 == -1) {
        throw new WriterException();
      }
      if (i + 1 < length) {
        int code2 = getAlphanumericCode(content.charAt(i + 1));
        if (code2 == -1) {
          throw new WriterException();
        }

        bits.appendBits(code1 * 45 + code2, 11);
        i += 2;
      } else {
        bits.appendBits(code1, 6);
        i++;
      }
    }
  }
```

- **MatrixUtil.java**

```
private static final int[][] POSITION_DETECTION_PATTERN =  {
      {1, 1, 1, 1, 1, 1, 1},
      {1, 0, 0, 0, 0, 0, 1},
      {1, 0, 1, 1, 1, 0, 1},
      {1, 0, 1, 1, 1, 0, 1},
      {1, 0, 1, 1, 1, 0, 1},
      {1, 0, 0, 0, 0, 0, 1},
      {1, 1, 1, 1, 1, 1, 1},
  };
  private static final int[][] POSITION_ADJUSTMENT_PATTERN = {
      {1, 1, 1, 1, 1},
      {1, 0, 0, 0, 1},
      {1, 0, 1, 0, 1},
      {1, 0, 0, 0, 1},
      {1, 1, 1, 1, 1},
  };

  // From Appendix E. Table 1, JIS0510X:2004 (p 71). The table was double-checked by
komatsu.
  private static final int[][] POSITION_ADJUSTMENT_PATTERN_COORDINATE_TABLE = {
      {-1, -1, -1, -1,  -1,  -1,  -1},  // Version 1
      { 6, 18, -1, -1,  -1,  -1,  -1},  // Version 2
      { 6, 22, -1, -1,  -1,  -1,  -1},  // Version 3
      { 6, 26, -1, -1,  -1,  -1,  -1},  // Version 4
      { 6, 30, -1, -1,  -1,  -1,  -1},  // Version 5

// Type info cells at the left top corner.
  private static final int[][] TYPE_INFO_COORDINATES = {
      {8, 0},
      {8, 1},
      {8, 2},
      {8, 3},
      {8, 4},
      {8, 5},
      {8, 7},
      {8, 8},
      {7, 8},
```

```
      {5, 8},
      {4, 8},
      {3, 8},
      {2, 8},
      {1, 8},
      {0, 8},
  };

  // From Appendix D in JISX0510:2004 (p. 67)
  private static final int VERSION_INFO_POLY = 0x1f25;  // 1 1111 0010 0101

  // From Appendix C in JISX0510:2004 (p.65).
  private static final int TYPE_INFO_POLY = 0x537;
  private static final int TYPE_INFO_MASK_PATTERN = 0x5412;

  // Set all cells to -1.  -1 means that the cell is empty (not set yet).
  //
  // JAVAPORT: We shouldn't need to do this at all. The code should be rewritten to
begin encoding
  // with the ByteMatrix initialized all to zero.
  static void clearMatrix(ByteMatrix matrix) {
    matrix.clear((byte) -1);
  }

  // Build 2D matrix of QR Code from "dataBits" with "ecLevel", "version" and
"getMaskPattern". On
  // success, store the result in "matrix" and return true.
  static void buildMatrix(BitArray dataBits,
                          ErrorCorrectionLevel ecLevel,
                          Version version,
                          int maskPattern,
                          ByteMatrix matrix) throws WriterException {
    clearMatrix(matrix);
    embedBasicPatterns(version, matrix);
    // Type information appear with any version.
    embedTypeInfo(ecLevel, maskPattern, matrix);
    // Version info appear if version >= 7.
    maybeEmbedVersionInfo(version, matrix);
    // Data should be embedded at end.
    embedDataBits(dataBits, maskPattern, matrix);
  }

  // Embed basic patterns. On success, modify the matrix and return true.
  // The basic patterns are:
  // - Position detection patterns
  // - Timing patterns
  // - Dark dot at the left bottom corner
  // - Position adjustment patterns, if need be
  static void embedBasicPatterns(Version version, ByteMatrix matrix) throws
WriterException {
    // Let's get started with embedding big squares at corners.
    embedPositionDetectionPatternsAndSeparators(matrix);
    // Then, embed the dark dot at the left bottom corner.
    embedDarkDotAtLeftBottomCorner(matrix);

    // Position adjustment patterns appear if version >= 2.
    maybeEmbedPositionAdjustmentPatterns(version, matrix);
    // Timing patterns should be embedded after position adj. patterns.
    embedTimingPatterns(matrix);
  }

  // Embed type information. On success, modify the matrix.
```

```
  static void embedTypeInfo(ErrorCorrectionLevel ecLevel, int maskPattern, ByteMatrix
matrix)
      throws WriterException {
    BitArray typeInfoBits = new BitArray();
    makeTypeInfoBits(ecLevel, maskPattern, typeInfoBits);

    for (int i = 0; i < typeInfoBits.getSize(); ++i) {
      // Place bits in LSB to MSB order.  LSB (least significant bit) is the last
value in
      // "typeInfoBits".
      boolean bit = typeInfoBits.get(typeInfoBits.getSize() - 1 - i);

      // Type info bits at the left top corner. See 8.9 of JISX0510:2004 (p.46).
      int x1 = TYPE_INFO_COORDINATES[i][0];
      int y1 = TYPE_INFO_COORDINATES[i][1];
      matrix.set(x1, y1, bit);

      if (i < 8) {
        // Right top corner.
        int x2 = matrix.getWidth() - i - 1;
        int y2 = 8;
        matrix.set(x2, y2, bit);
      } else {
        // Left bottom corner.
        int x2 = 8;
        int y2 = matrix.getHeight() - 7 + (i - 8);
        matrix.set(x2, y2, bit);
      }
    }
  }

  // Embed version information if need be. On success, modify the matrix and return
true.
  // See 8.10 of JISX0510:2004 (p.47) for how to embed version information.
  static void maybeEmbedVersionInfo(Version version, ByteMatrix matrix) throws
WriterException {
    if (version.getVersionNumber() < 7) {  // Version info is necessary if version >=
7.
      return;  // Don't need version info.
    }
    BitArray versionInfoBits = new BitArray();
    makeVersionInfoBits(version, versionInfoBits);

    int bitIndex = 6 * 3 - 1;  // It will decrease from 17 to 0.
    for (int i = 0; i < 6; ++i) {
      for (int j = 0; j < 3; ++j) {
        // Place bits in LSB (least significant bit) to MSB order.
        boolean bit = versionInfoBits.get(bitIndex);
        bitIndex--;
        // Left bottom corner.
        matrix.set(i, matrix.getHeight() - 11 + j, bit);
        // Right bottom corner.
        matrix.set(matrix.getHeight() - 11 + j, i, bit);
      }
    }
  }

  // Embed "dataBits" using "getMaskPattern". On success, modify the matrix and return
true.
  // For debugging purposes, it skips masking process if "getMaskPattern" is -1.
  // See 8.7 of JISX0510:2004 (p.38) for how to embed data bits.
  static void embedDataBits(BitArray dataBits, int maskPattern, ByteMatrix matrix)
      throws WriterException {
```

```
    int bitIndex = 0;
    int direction = -1;
    // Start from the right bottom cell.
    int x = matrix.getWidth() - 1;
    int y = matrix.getHeight() - 1;
    while (x > 0) {
      // Skip the vertical timing pattern.
      if (x == 6) {
        x -= 1;
      }
      while (y >= 0 && y < matrix.getHeight()) {
        for (int i = 0; i < 2; ++i) {
          int xx = x - i;
          // Skip the cell if it's not empty.
          if (!isEmpty(matrix.get(xx, y))) {
            continue;
          }
          boolean bit;
          if (bitIndex < dataBits.getSize()) {
            bit = dataBits.get(bitIndex);
            ++bitIndex;
          } else {
            // Padding bit. If there is no bit left, we'll fill the left cells with 0,
as described
            // in 8.4.9 of JISX0510:2004 (p. 24).
            bit = false;
          }

          // Skip masking if mask_pattern is -1.
          if (maskPattern != -1 && MaskUtil.getDataMaskBit(maskPattern, xx, y)) {
            bit = !bit;
          }
          matrix.set(xx, y, bit);
        }
        y += direction;
      }
      direction = -direction;  // Reverse the direction.
      y += direction;
      x -= 2;  // Move to the left.
    }
    // All bits should be consumed.
    if (bitIndex != dataBits.getSize()) {
      throw new WriterException("Not all bits consumed: " + bitIndex + '/' +
dataBits.getSize());
    }
  }

  // Return the position of the most significant bit set (to one) in the "value". The
most
  // significant bit is position 32. If there is no bit set, return 0. Examples:
  // - findMSBSet(0) => 0
  // - findMSBSet(1) => 1
  // - findMSBSet(255) => 8
  static int findMSBSet(int value) {
    int numDigits = 0;
    while (value != 0) {
      value >>>= 1;
      ++numDigits;
    }
    return numDigits;
  }
```

```
  // operations. We don't care if cofficients are positive or negative.
  static int calculateBCHCode(int value, int poly) {
    // If poly is "1 1111 0010 0101" (version info poly), msbSetInPoly is 13. We'll
subtract 1
    // from 13 to make it 12.
    int msbSetInPoly = findMSBSet(poly);
    value <<= msbSetInPoly - 1;
    // Do the division business using exclusive-or operations.
    while (findMSBSet(value) >= msbSetInPoly) {
      value ^= poly << (findMSBSet(value) - msbSetInPoly);
    }
    // Now the "value" is the remainder (i.e. the BCH code)
    return value;
  }

  // Make bit vector of type information. On success, store the result in "bits" and
return true.
  // Encode error correction level and mask pattern. See 8.9 of
  // JISX0510:2004 (p.45) for details.
  static void makeTypeInfoBits(ErrorCorrectionLevel ecLevel, int maskPattern, BitArray
bits)
      throws WriterException {
    if (!QRCode.isValidMaskPattern(maskPattern)) {
      throw new WriterException("Invalid mask pattern");
    }
    int typeInfo = (ecLevel.getBits() << 3) | maskPattern;
    bits.appendBits(typeInfo, 5);

    int bchCode = calculateBCHCode(typeInfo, TYPE_INFO_POLY);
    bits.appendBits(bchCode, 10);

    BitArray maskBits = new BitArray();
    maskBits.appendBits(TYPE_INFO_MASK_PATTERN, 15);
    bits.xor(maskBits);

    if (bits.getSize() != 15) {  // Just in case.
      throw new WriterException("should not happen but we got: " + bits.getSize());
    }
  }

  // Make bit vector of version information. On success, store the result in "bits"
and return true.
  // See 8.10 of JISX0510:2004 (p.45) for details.
  static void makeVersionInfoBits(Version version, BitArray bits) throws
WriterException {
    bits.appendBits(version.getVersionNumber(), 6);
    int bchCode = calculateBCHCode(version.getVersionNumber(), VERSION_INFO_POLY);
    bits.appendBits(bchCode, 12);

    if (bits.getSize() != 18) {  // Just in case.
      throw new WriterException("should not happen but we got: " + bits.getSize());
    }
  }

  // Check if "value" is empty.
  private static boolean isEmpty(int value) {
    return value == -1;
  }

  private static void embedTimingPatterns(ByteMatrix matrix) {
    // -8 is for skipping position detection patterns (size 7), and two
horizontal/vertical
    // separation patterns (size 1). Thus, 8 = 7 + 1.
```

```
    for (int i = 8; i < matrix.getWidth() - 8; ++i) {
      int bit = (i + 1) % 2;
      // Horizontal line.
      if (isEmpty(matrix.get(i, 6))) {
        matrix.set(i, 6, bit);
      }
      // Vertical line.
      if (isEmpty(matrix.get(6, i))) {
        matrix.set(6, i, bit);
      }
    }
  }

  // Embed the lonely dark dot at left bottom corner. JISX0510:2004 (p.46)
  private static void embedDarkDotAtLeftBottomCorner(ByteMatrix matrix) throws
WriterException {
    if (matrix.get(8, matrix.getHeight() - 8) == 0) {
      throw new WriterException();
    }
    matrix.set(8, matrix.getHeight() - 8, 1);
  }

  private static void embedHorizontalSeparationPattern(int xStart,
                                                       int yStart,
                                                       ByteMatrix matrix) throws
WriterException {
    for (int x = 0; x < 8; ++x) {
      if (!isEmpty(matrix.get(xStart + x, yStart))) {
        throw new WriterException();
      }
      matrix.set(xStart + x, yStart, 0);
    }
  }

  private static void embedVerticalSeparationPattern(int xStart,
                                                     int yStart,
                                                     ByteMatrix matrix) throws
WriterException {
    for (int y = 0; y < 7; ++y) {
      if (!isEmpty(matrix.get(xStart, yStart + y))) {
        throw new WriterException();
      }
      matrix.set(xStart, yStart + y, 0);
    }
  }

  // Note that we cannot unify the function with embedPositionDetectionPattern()
despite they are
  // almost identical, since we cannot write a function that takes 2D arrays in
different sizes in
  // C/C++. We should live with the fact.
  private static void embedPositionAdjustmentPattern(int xStart, int yStart,
ByteMatrix matrix) {
    for (int y = 0; y < 5; ++y) {
      for (int x = 0; x < 5; ++x) {
        matrix.set(xStart + x, yStart + y, POSITION_ADJUSTMENT_PATTERN[y][x]);
      }
    }
  }

  private static void embedPositionDetectionPattern(int xStart, int yStart, ByteMatrix
matrix) {
    for (int y = 0; y < 7; ++y) {
```

```
      for (int x = 0; x < 7; ++x) {
        matrix.set(xStart + x, yStart + y, POSITION_DETECTION_PATTERN[y][x]);
      }
    }
  }

  // Embed position detection patterns and surrounding vertical/horizontal separators.
  private static void embedPositionDetectionPatternsAndSeparators(ByteMatrix matrix)
throws WriterException {
    // Embed three big squares at corners.
    int pdpWidth = POSITION_DETECTION_PATTERN[0].length;
    // Left top corner.
    embedPositionDetectionPattern(0, 0, matrix);
    // Right top corner.
    embedPositionDetectionPattern(matrix.getWidth() - pdpWidth, 0, matrix);
    // Left bottom corner.
    embedPositionDetectionPattern(0, matrix.getWidth() - pdpWidth, matrix);

    // Embed horizontal separation patterns around the squares.
    int hspWidth = 8;
    // Left top corner.
    embedHorizontalSeparationPattern(0, hspWidth - 1, matrix);
    // Right top corner.
    embedHorizontalSeparationPattern(matrix.getWidth() - hspWidth,
        hspWidth - 1, matrix);
    // Left bottom corner.
    embedHorizontalSeparationPattern(0, matrix.getWidth() - hspWidth, matrix);

    // Embed vertical separation patterns around the squares.
    int vspSize = 7;
    // Left top corner.
    embedVerticalSeparationPattern(vspSize, 0, matrix);
    // Right top corner.
    embedVerticalSeparationPattern(matrix.getHeight() - vspSize - 1, 0, matrix);
    // Left bottom corner.
    embedVerticalSeparationPattern(vspSize, matrix.getHeight() - vspSize,
        matrix);
  }

  // Embed position adjustment patterns if need be.
  private static void maybeEmbedPositionAdjustmentPatterns(Version version, ByteMatrix
matrix) {
    if (version.getVersionNumber() < 2) {  // The patterns appear if version >= 2
      return;
    }
    int index = version.getVersionNumber() - 1;
    int[] coordinates = POSITION_ADJUSTMENT_PATTERN_COORDINATE_TABLE[index];
    int numCoordinates = POSITION_ADJUSTMENT_PATTERN_COORDINATE_TABLE[index].length;
    for (int i = 0; i < numCoordinates; ++i) {
      for (int j = 0; j < numCoordinates; ++j) {
        int y = coordinates[i];
        int x = coordinates[j];
        if (x == -1 || y == -1) {
          continue;
        }
        // If the cell is unset, we embed the position adjustment pattern here.
        if (isEmpty(matrix.get(x, y))) {
          // -2 is necessary since the x/y coordinates point to the center of the
pattern, not the
          // left top corner.
          embedPositionAdjustmentPattern(x - 2, y - 2, matrix);
        }
      }
```

```
    }
  }

}
```

- **MaskUtil.java**

```
final class MaskUtil {

  // Penalty weights from section 6.8.2.1
  private static final int N1 = 3;
  private static final int N2 = 3;
  private static final int N3 = 40;
  private static final int N4 = 10;

  private MaskUtil() {
    // do nothing
  }

  /**
   * Apply mask penalty rule 1 and return the penalty. Find repetitive cells with the
same color and
   * give penalty to them. Example: 00000 or 11111.
   */
  static int applyMaskPenaltyRule1(ByteMatrix matrix) {
    return applyMaskPenaltyRule1Internal(matrix, true) +
applyMaskPenaltyRule1Internal(matrix, false);
  }

  /**
   * Apply mask penalty rule 2 and return the penalty. Find 2x2 blocks with the same
color and give
   * penalty to them. This is actually equivalent to the spec's rule, which is to find
MxN blocks and give a
   * penalty proportional to (M-1)x(N-1), because this is the number of 2x2 blocks
inside such a block.
   */
  static int applyMaskPenaltyRule2(ByteMatrix matrix) {
    int penalty = 0;
    byte[][] array = matrix.getArray();
    int width = matrix.getWidth();
    int height = matrix.getHeight();
    for (int y = 0; y < height - 1; y++) {
      for (int x = 0; x < width - 1; x++) {
        int value = array[y][x];
        if (value == array[y][x + 1] && value == array[y + 1][x] && value == array[y +
1][x + 1]) {
          penalty++;
        }
      }
    }
    return N2 * penalty;
  }

  /**
   * Apply mask penalty rule 3 and return the penalty. Find consecutive cells of
00001011101 or
   * 10111010000, and give penalty to them.  If we find patterns like 000010111010000,
we give
   * penalties twice (i.e. 40 * 2).
   */
  static int applyMaskPenaltyRule3(ByteMatrix matrix) {
    int penalty = 0;
```

```
    byte[][] array = matrix.getArray();
    int width = matrix.getWidth();
    int height = matrix.getHeight();
    for (int y = 0; y < height; y++) {
      for (int x = 0; x < width; x++) {
        // Tried to simplify following conditions but failed.
        if (x + 6 < width &&
            array[y][x] == 1 &&
            array[y][x +  1] == 0 &&
            array[y][x +  2] == 1 &&
            array[y][x +  3] == 1 &&
            array[y][x +  4] == 1 &&
            array[y][x +  5] == 0 &&
            array[y][x +  6] == 1 &&
            ((x + 10 < width &&
                array[y][x +  7] == 0 &&
                array[y][x +  8] == 0 &&
                array[y][x +  9] == 0 &&
                array[y][x + 10] == 0) ||
             (x - 4 >= 0 &&
                array[y][x -  1] == 0 &&
                array[y][x -  2] == 0 &&
                array[y][x -  3] == 0 &&
                array[y][x -  4] == 0))) {
          penalty += N3;
        }
        if (y + 6 < height &&
            array[y][x] == 1  &&
            array[y +  1][x] == 0  &&
            array[y +  2][x] == 1  &&
            array[y +  3][x] == 1  &&
            array[y +  4][x] == 1  &&
            array[y +  5][x] == 0  &&
            array[y +  6][x] == 1 &&
            ((y + 10 < height &&
                array[y +  7][x] == 0 &&
                array[y +  8][x] == 0 &&
                array[y +  9][x] == 0 &&
                array[y + 10][x] == 0) ||
             (y - 4 >= 0 &&
                array[y -  1][x] == 0 &&
                array[y -  2][x] == 0 &&
                array[y -  3][x] == 0 &&
                array[y -  4][x] == 0))) {
          penalty += N3;
        }
      }
    }
    return penalty;
  }

  /**
   * Apply mask penalty rule 4 and return the penalty. Calculate the ratio of dark
cells and give
   * penalty if the ratio is far from 50%. It gives 10 penalty for 5% distance.
   */
  static int applyMaskPenaltyRule4(ByteMatrix matrix) {
    int numDarkCells = 0;
    byte[][] array = matrix.getArray();
    int width = matrix.getWidth();
    int height = matrix.getHeight();
    for (int y = 0; y < height; y++) {
      byte[] arrayY = array[y];
```

```
      for (int x = 0; x < width; x++) {
        if (arrayY[x] == 1) {
          numDarkCells++;
        }
      }
    }
    int numTotalCells = matrix.getHeight() * matrix.getWidth();
    double darkRatio = (double) numDarkCells / numTotalCells;
    int fivePercentVariances = (int) (Math.abs(darkRatio - 0.5) * 20.0); // * 100.0 /
5.0
    return fivePercentVariances * N4;
  }

  /**
   * Return the mask bit for "getMaskPattern" at "x" and "y". See 8.8 of JISX0510:2004
for mask
   * pattern conditions.
   */
  static boolean getDataMaskBit(int maskPattern, int x, int y) {
    int intermediate;
    int temp;
    switch (maskPattern) {
      case 0:
        intermediate = (y + x) & 0x1;
        break;
      case 1:
        intermediate = y & 0x1;
        break;
      case 2:
        intermediate = x % 3;
        break;
      case 3:
        intermediate = (y + x) % 3;
        break;
      case 4:
        intermediate = ((y >>> 1) + (x / 3)) & 0x1;
        break;
      case 5:
        temp = y * x;
        intermediate = (temp & 0x1) + (temp % 3);
        break;
      case 6:
        temp = y * x;
        intermediate = ((temp & 0x1) + (temp % 3)) & 0x1;
        break;
      case 7:
        temp = y * x;
        intermediate = ((temp % 3) + ((y + x) & 0x1)) & 0x1;
        break;
      default:
        throw new IllegalArgumentException("Invalid mask pattern: " + maskPattern);
    }
    return intermediate == 0;
  }

  /**
   * Helper function for applyMaskPenaltyRule1. We need this for doing this
calculation in both
   * vertical and horizontal orders respectively.
   */
  private static int applyMaskPenaltyRule1Internal(ByteMatrix matrix, boolean
isHorizontal) {
    int penalty = 0;
```

```
    int iLimit = isHorizontal ? matrix.getHeight() : matrix.getWidth();
    int jLimit = isHorizontal ? matrix.getWidth() : matrix.getHeight();
    byte[][] array = matrix.getArray();
    for (int i = 0; i < iLimit; i++) {
      int numSameBitCells = 0;
      int prevBit = -1;
      for (int j = 0; j < jLimit; j++) {
        int bit = isHorizontal ? array[i][j] : array[j][i];
        if (bit == prevBit) {
          numSameBitCells++;
        } else {
          if (numSameBitCells >= 5) {
            penalty += N1 + (numSameBitCells - 5);
          }
          numSameBitCells = 1;  // Include the cell itself.
          prevBit = bit;
        }
      }
      if (numSameBitCells > 5) {
        penalty += N1 + (numSameBitCells - 5);
      }
    }
    return penalty;
  }

}
```

Reed-Solomon Error Correction

- **GenericGF.java**

```
public static final GenericGF QR_CODE_FIELD_256 = new GenericGF(0x011D, 256); // x^8 +
x^4 + x^3 + x^2 + 1

private static final int INITIALIZATION_THRESHOLD = 0;

  private int[] expTable;
  private int[] logTable;
  private GenericGFPoly zero;
  private GenericGFPoly one;
  private final int size;
  private final int primitive;
  private boolean initialized = false;

  /**
   * Create a representation of GF(size) using the given primitive polynomial.
   *
   * @param primitive irreducible polynomial whose coefficients are represented by
   *  the bits of an int, where the least-significant bit represents the constant
   *  coefficient
   */
  public GenericGF(int primitive, int size) {
       this.primitive = primitive;
    this.size = size;

    if (size <= INITIALIZATION_THRESHOLD){
       initialize();
    }
  }

  private void initialize(){
    expTable = new int[size];
    logTable = new int[size];
    int x = 1;
```

```
    for (int i = 0; i < size; i++) {
      expTable[i] = x;
      x <<= 1; // x = x * 2; we're assuming the generator alpha is 2
      if (x >= size) {
        x ^= primitive;
        x &= size-1;
      }
    }
    for (int i = 0; i < size-1; i++) {
      logTable[expTable[i]] = i;
    }
    // logTable[0] == 0 but this should never be used
    zero = new GenericGFPoly(this, new int[]{0});
    one = new GenericGFPoly(this, new int[]{1});
    initialized = true;
  }

  private void checkInit(){
       if (!initialized) {
      initialize();
    }
  }

  GenericGFPoly getZero() {
       checkInit();

    return zero;
  }

  GenericGFPoly getOne() {
       checkInit();

    return one;
  }

  /**
   * @return the monomial representing coefficient * x^degree
   */
  GenericGFPoly buildMonomial(int degree, int coefficient) {
       checkInit();

    if (degree < 0) {
      throw new IllegalArgumentException();
    }
    if (coefficient == 0) {
      return zero;
    }
    int[] coefficients = new int[degree + 1];
    coefficients[0] = coefficient;
    return new GenericGFPoly(this, coefficients);
  }

  /**
   * Implements both addition and subtraction -- they are the same in GF(size).
   *
   * @return sum/difference of a and b
   */
  static int addOrSubtract(int a, int b) {
    return a ^ b;
  }

  /**
   * @return 2 to the power of a in GF(size)
```

```
   */
  int exp(int a) {
       checkInit();

    return expTable[a];
  }

  /**
   * @return base 2 log of a in GF(size)
   */
  int log(int a) {
       checkInit();

    if (a == 0) {
      throw new IllegalArgumentException();
    }
    return logTable[a];
  }

  /**
   * @return multiplicative inverse of a
   */
  int inverse(int a) {
       checkInit();

    if (a == 0) {
      throw new ArithmeticException();
    }
    return expTable[size - logTable[a] - 1];
  }

  /**
   * @return product of a and b in GF(size)
   */
  int multiply(int a, int b) {
       checkInit();

    if (a == 0 || b == 0) {
      return 0;
    }
    return expTable[(logTable[a] + logTable[b]) % (size - 1)];
  }

  public int getSize() {
       return size;
  }

}
```

- **GenericGFPoly.java**

```
final class GenericGFPoly {

  private final GenericGF field;
  private final int[] coefficients;

  /**
   * @param field the {@link GenericGF} instance representing the field to use
   * to perform computations
   * @param coefficients coefficients as ints representing elements of GF(size),
arranged
   * from most significant (highest-power term) coefficient to least significant
   * @throws IllegalArgumentException if argument is null or empty,
```

```
   * or if leading coefficient is 0 and this is not a
   * constant polynomial (that is, it is not the monomial "0")
   */
  GenericGFPoly(GenericGF field, int[] coefficients) {
    if (coefficients.length == 0) {
      throw new IllegalArgumentException();
    }
    this.field = field;
    int coefficientsLength = coefficients.length;
    if (coefficientsLength > 1 && coefficients[0] == 0) {
      // Leading term must be non-zero for anything except the constant polynomial "0"
      int firstNonZero = 1;
      while (firstNonZero < coefficientsLength && coefficients[firstNonZero] == 0) {
        firstNonZero++;
      }
      if (firstNonZero == coefficientsLength) {
        this.coefficients = field.getZero().coefficients;
      } else {
        this.coefficients = new int[coefficientsLength - firstNonZero];
        System.arraycopy(coefficients,
            firstNonZero,
            this.coefficients,
            0,
            this.coefficients.length);
      }
    } else {
      this.coefficients = coefficients;
    }
  }

  int[] getCoefficients() {
    return coefficients;
  }

  /**
   * @return degree of this polynomial
   */
  int getDegree() {
    return coefficients.length - 1;
  }

  /**
   * @return true iff this polynomial is the monomial "0"
   */
  boolean isZero() {
    return coefficients[0] == 0;
  }

  /**
   * @return coefficient of x^degree term in this polynomial
   */
  int getCoefficient(int degree) {
    return coefficients[coefficients.length - 1 - degree];
  }

  /**
   * @return evaluation of this polynomial at a given point
   */
  int evaluateAt(int a) {
    if (a == 0) {
      // Just return the x^0 coefficient
      return getCoefficient(0);
    }
```

```
    int size = coefficients.length;
    if (a == 1) {
      // Just the sum of the coefficients
      int result = 0;
      for (int coefficient : coefficients) {
        result = GenericGF.addOrSubtract(result, coefficient);
      }
      return result;
    }
    int result = coefficients[0];
    for (int i = 1; i < size; i++) {
      result = GenericGF.addOrSubtract(field.multiply(a, result), coefficients[i]);
    }
    return result;
  }

  GenericGFPoly addOrSubtract(GenericGFPoly other) {
    if (!field.equals(other.field)) {
      throw new IllegalArgumentException("GenericGFPolys do not have same GenericGF
field");
    }
    if (isZero()) {
      return other;
    }
    if (other.isZero()) {
      return this;
    }

    int[] smallerCoefficients = this.coefficients;
    int[] largerCoefficients = other.coefficients;
    if (smallerCoefficients.length > largerCoefficients.length) {
      int[] temp = smallerCoefficients;
      smallerCoefficients = largerCoefficients;
      largerCoefficients = temp;
    }
    int[] sumDiff = new int[largerCoefficients.length];
    int lengthDiff = largerCoefficients.length - smallerCoefficients.length;
    // Copy high-order terms only found in higher-degree polynomial's coefficients
    System.arraycopy(largerCoefficients, 0, sumDiff, 0, lengthDiff);

    for (int i = lengthDiff; i < largerCoefficients.length; i++) {
      sumDiff[i] = GenericGF.addOrSubtract(smallerCoefficients[i - lengthDiff],
largerCoefficients[i]);
    }

    return new GenericGFPoly(field, sumDiff);
  }

  GenericGFPoly multiply(GenericGFPoly other) {
    if (!field.equals(other.field)) {
      throw new IllegalArgumentException("GenericGFPolys do not have same GenericGF
field");
    }
    if (isZero() || other.isZero()) {
      return field.getZero();
    }
    int[] aCoefficients = this.coefficients;
    int aLength = aCoefficients.length;
    int[] bCoefficients = other.coefficients;
    int bLength = bCoefficients.length;
    int[] product = new int[aLength + bLength - 1];
    for (int i = 0; i < aLength; i++) {
      int aCoeff = aCoefficients[i];
```

```
      for (int j = 0; j < bLength; j++) {
        product[i + j] = GenericGF.addOrSubtract(product[i + j],
            field.multiply(aCoeff, bCoefficients[j]));
      }
    }
    return new GenericGFPoly(field, product);
  }

  GenericGFPoly multiply(int scalar) {
    if (scalar == 0) {
      return field.getZero();
    }
    if (scalar == 1) {
      return this;
    }
    int size = coefficients.length;
    int[] product = new int[size];
    for (int i = 0; i < size; i++) {
      product[i] = field.multiply(coefficients[i], scalar);
    }
    return new GenericGFPoly(field, product);
  }

  GenericGFPoly multiplyByMonomial(int degree, int coefficient) {
    if (degree < 0) {
      throw new IllegalArgumentException();
    }
    if (coefficient == 0) {
      return field.getZero();
    }
    int size = coefficients.length;
    int[] product = new int[size + degree];
    for (int i = 0; i < size; i++) {
      product[i] = field.multiply(coefficients[i], coefficient);
    }
    return new GenericGFPoly(field, product);
  }

  GenericGFPoly[] divide(GenericGFPoly other) {
    if (!field.equals(other.field)) {
      throw new IllegalArgumentException("GenericGFPolys do not have same GenericGF
field");
    }
    if (other.isZero()) {
      throw new IllegalArgumentException("Divide by 0");
    }

    GenericGFPoly quotient = field.getZero();
    GenericGFPoly remainder = this;

    int denominatorLeadingTerm = other.getCoefficient(other.getDegree());
    int inverseDenominatorLeadingTerm = field.inverse(denominatorLeadingTerm);

    while (remainder.getDegree() >= other.getDegree() && !remainder.isZero()) {
      int degreeDifference = remainder.getDegree() - other.getDegree();
      int scale = field.multiply(remainder.getCoefficient(remainder.getDegree()),
inverseDenominatorLeadingTerm);
      GenericGFPoly term = other.multiplyByMonomial(degreeDifference, scale);
      GenericGFPoly iterationQuotient = field.buildMonomial(degreeDifference, scale);
      quotient = quotient.addOrSubtract(iterationQuotient);
      remainder = remainder.addOrSubtract(term);
    }
```

```
    return new GenericGFPoly[] { quotient, remainder };
  }

  @Override
  public String toString() {
    StringBuilder result = new StringBuilder(8 * getDegree());
    for (int degree = getDegree(); degree >= 0; degree--) {
      int coefficient = getCoefficient(degree);
      if (coefficient != 0) {
        if (coefficient < 0) {
          result.append(" - ");
          coefficient = -coefficient;
        } else {
          if (result.length() > 0) {
            result.append(" + ");
          }
        }
        if (degree == 0 || coefficient != 1) {
          int alphaPower = field.log(coefficient);
          if (alphaPower == 0) {
            result.append('1');
          } else if (alphaPower == 1) {
            result.append('a');
          } else {
            result.append("a^");
            result.append(alphaPower);
          }
        }
        if (degree != 0) {
          if (degree == 1) {
            result.append('x');
          } else {
            result.append("x^");
            result.append(degree);
          }
        }
      }
    }
    return result.toString();
  }

}
```

- **ReedSolomonDecoder.java**

```
public final class ReedSolomonDecoder {

  private final GenericGF field;

  public ReedSolomonDecoder(GenericGF field) {
    this.field = field;
  }

  /**
   * <p>Decodes given set of received codewords, which include both data and error-
correction
   * codewords. Really, this means it uses Reed-Solomon to detect and correct errors,
in-place,
   * in the input.</p>
   *
   * @param received data and error-correction codewords
   * @param twoS number of error-correction codewords available
   * @throws ReedSolomonException if decoding fails for any reason
```

```
   */
  public void decode(int[] received, int twoS) throws ReedSolomonException {
    GenericGFPoly poly = new GenericGFPoly(field, received);
    int[] syndromeCoefficients = new int[twoS];
    boolean dataMatrix = field.equals(GenericGF.DATA_MATRIX_FIELD_256);
    boolean noError = true;
    for (int i = 0; i < twoS; i++) {
      // Thanks to sanfordsquires for this fix:
      int eval = poly.evaluateAt(field.exp(dataMatrix ? i + 1 : i));
      syndromeCoefficients[syndromeCoefficients.length - 1 - i] = eval;
      if (eval != 0) {
        noError = false;
      }
    }
    if (noError) {
      return;
    }
    GenericGFPoly syndrome = new GenericGFPoly(field, syndromeCoefficients);
    GenericGFPoly[] sigmaOmega =
        runEuclideanAlgorithm(field.buildMonomial(twoS, 1), syndrome, twoS);
    GenericGFPoly sigma = sigmaOmega[0];
    GenericGFPoly omega = sigmaOmega[1];
    int[] errorLocations = findErrorLocations(sigma);
    int[] errorMagnitudes = findErrorMagnitudes(omega, errorLocations, dataMatrix);
    for (int i = 0; i < errorLocations.length; i++) {
      int position = received.length - 1 - field.log(errorLocations[i]);
      if (position < 0) {
        throw new ReedSolomonException("Bad error location");
      }
      received[position] = GenericGF.addOrSubtract(received[position],
errorMagnitudes[i]);
    }
  }

  private GenericGFPoly[] runEuclideanAlgorithm(GenericGFPoly a, GenericGFPoly b, int
R)
      throws ReedSolomonException {
    // Assume a's degree is >= b's
    if (a.getDegree() < b.getDegree()) {
      GenericGFPoly temp = a;
      a = b;
      b = temp;
    }

    GenericGFPoly rLast = a;
    GenericGFPoly r = b;
    GenericGFPoly tLast = field.getZero();
    GenericGFPoly t = field.getOne();

    // Run Euclidean algorithm until r's degree is less than R/2
    while (r.getDegree() >= R / 2) {
      GenericGFPoly rLastLast = rLast;
      GenericGFPoly tLastLast = tLast;
      rLast = r;
      tLast = t;

      // Divide rLastLast by rLast, with quotient in q and remainder in r
      if (rLast.isZero()) {
        // Oops, Euclidean algorithm already terminated?
        throw new ReedSolomonException("r_{i-1} was zero");
      }
      r = rLastLast;
      GenericGFPoly q = field.getZero();
```

```
      int denominatorLeadingTerm = rLast.getCoefficient(rLast.getDegree());
      int dltInverse = field.inverse(denominatorLeadingTerm);
      while (r.getDegree() >= rLast.getDegree() && !r.isZero()) {
        int degreeDiff = r.getDegree() - rLast.getDegree();
        int scale = field.multiply(r.getCoefficient(r.getDegree()), dltInverse);
        q = q.addOrSubtract(field.buildMonomial(degreeDiff, scale));
        r = r.addOrSubtract(rLast.multiplyByMonomial(degreeDiff, scale));
      }

      t = q.multiply(tLast).addOrSubtract(tLastLast);
    }

    int sigmaTildeAtZero = t.getCoefficient(0);
    if (sigmaTildeAtZero == 0) {
      throw new ReedSolomonException("sigmaTilde(0) was zero");
    }

    int inverse = field.inverse(sigmaTildeAtZero);
    GenericGFPoly sigma = t.multiply(inverse);
    GenericGFPoly omega = r.multiply(inverse);
    return new GenericGFPoly[]{sigma, omega};
  }

  private int[] findErrorLocations(GenericGFPoly errorLocator) throws
ReedSolomonException {
    // This is a direct application of Chien's search
    int numErrors = errorLocator.getDegree();
    if (numErrors == 1) { // shortcut
      return new int[] { errorLocator.getCoefficient(1) };
    }
    int[] result = new int[numErrors];
    int e = 0;
    for (int i = 1; i < field.getSize() && e < numErrors; i++) {
      if (errorLocator.evaluateAt(i) == 0) {
        result[e] = field.inverse(i);
        e++;
      }
    }
    if (e != numErrors) {
      throw new ReedSolomonException("Error locator degree does not match number of
roots");
    }
    return result;
  }

  private int[] findErrorMagnitudes(GenericGFPoly errorEvaluator,
                                    int[] errorLocations,
                                    boolean dataMatrix) {
    // This is directly applying Forney's Formula
    int s = errorLocations.length;
    int[] result = new int[s];
    for (int i = 0; i < s; i++) {
      int xiInverse = field.inverse(errorLocations[i]);
      int denominator = 1;
      for (int j = 0; j < s; j++) {
        if (i != j) {
          //denominator = field.multiply(denominator,
          //    GenericGF.addOrSubtract(1, field.multiply(errorLocations[j],
xiInverse)));
          // Above should work but fails on some Apple and Linux JDKs due to a Hotspot
bug.
          // Below is a funny-looking workaround from Steven Parkes
          int term = field.multiply(errorLocations[j], xiInverse);
```

```
          int termPlus1 = (term & 0x1) == 0 ? term | 1 : term & ~1;
          denominator = field.multiply(denominator, termPlus1);
        }
      }
      result[i] = field.multiply(errorEvaluator.evaluateAt(xiInverse),
          field.inverse(denominator));
      // Thanks to sanfordsquires for this fix:
      if (dataMatrix) {
        result[i] = field.multiply(result[i], xiInverse);
      }
    }
    return result;
  }

}
```

- **ReedSolomonEncoder.java**

```
public final class ReedSolomonEncoder {

  private final GenericGF field;
  private final List<GenericGFPoly> cachedGenerators;

  public ReedSolomonEncoder(GenericGF field) {
    if (!GenericGF.QR_CODE_FIELD_256.equals(field)) {
      throw new IllegalArgumentException("Only QR Code is supported at this time");
    }
    this.field = field;
    this.cachedGenerators = new ArrayList<GenericGFPoly>();
    cachedGenerators.add(new GenericGFPoly(field, new int[]{1}));
  }

  private GenericGFPoly buildGenerator(int degree) {
    if (degree >= cachedGenerators.size()) {
      GenericGFPoly lastGenerator = cachedGenerators.get(cachedGenerators.size() - 1);
      for (int d = cachedGenerators.size(); d <= degree; d++) {
        GenericGFPoly nextGenerator = lastGenerator.multiply(new GenericGFPoly(field,
new int[] { 1, field.exp(d - 1) }));
        cachedGenerators.add(nextGenerator);
        lastGenerator = nextGenerator;
      }
    }
    return cachedGenerators.get(degree);
  }

  public void encode(int[] toEncode, int ecBytes) {
    if (ecBytes == 0) {
      throw new IllegalArgumentException("No error correction bytes");
    }
    int dataBytes = toEncode.length - ecBytes;
    if (dataBytes <= 0) {
      throw new IllegalArgumentException("No data bytes provided");
    }
    GenericGFPoly generator = buildGenerator(ecBytes);
    int[] infoCoefficients = new int[dataBytes];
    System.arraycopy(toEncode, 0, infoCoefficients, 0, dataBytes);
    GenericGFPoly info = new GenericGFPoly(field, infoCoefficients);
    info = info.multiplyByMonomial(ecBytes, 1);
    GenericGFPoly remainder = info.divide(generator)[1];
    int[] coefficients = remainder.getCoefficients();
    int numZeroCoefficients = ecBytes - coefficients.length;
    for (int i = 0; i < numZeroCoefficients; i++) {
      toEncode[dataBytes + i] = 0;
```

```
    }
    System.arraycopy(coefficients, 0, toEncode, dataBytes + numZeroCoefficients,
coefficients.length);
  }

}
```

Proses decode

- **AligmentPattern.java**

```
public final class AlignmentPattern extends ResultPoint {

  private final float estimatedModuleSize;

  AlignmentPattern(float posX, float posY, float estimatedModuleSize) {
    super(posX, posY);
    this.estimatedModuleSize = estimatedModuleSize;
  }

  /**
   * <p>Determines if this alignment pattern "about equals" an alignment pattern at
the stated
   * position and size -- meaning, it is at nearly the same center with nearly the
same size.</p>
   */
  boolean aboutEquals(float moduleSize, float i, float j) {
    if (Math.abs(i - getY()) <= moduleSize && Math.abs(j - getX()) <= moduleSize) {
      float moduleSizeDiff = Math.abs(moduleSize - estimatedModuleSize);
      return moduleSizeDiff <= 1.0f || moduleSizeDiff <= estimatedModuleSize;
    }
    return false;
  }

  /**
   * Combines this object's current estimate of a finder pattern position and module
size
   * with a new estimate. It returns a new {@code FinderPattern} containing an average
of the two.
   */
  AlignmentPattern combineEstimate(float i, float j, float newModuleSize) {
    float combinedX = (getX() + j) / 2.0f;
    float combinedY = (getY() + i) / 2.0f;
    float combinedModuleSize = (estimatedModuleSize + newModuleSize) / 2.0f;
    return new AlignmentPattern(combinedX, combinedY, combinedModuleSize);
  }

}
```

- **AlignmentPatternFinder.java**

```
final class AlignmentPatternFinder {

  private final BitMatrix image;
  private final List<AlignmentPattern> possibleCenters;
  private final int startX;
  private final int startY;
  private final int width;
  private final int height;
  private final float moduleSize;
  private final int[] crossCheckStateCount;
  private final ResultPointCallback resultPointCallback;
```

```
  /**
   * <p>Creates a finder that will look in a portion of the whole image.</p>
   *
   * @param image image to search
   * @param startX left column from which to start searching
   * @param startY top row from which to start searching
   * @param width width of region to search
   * @param height height of region to search
   * @param moduleSize estimated module size so far
   */
  AlignmentPatternFinder(BitMatrix image,
                         int startX,
                         int startY,
                         int width,
                         int height,
                         float moduleSize,
                         ResultPointCallback resultPointCallback) {
    this.image = image;
    this.possibleCenters = new ArrayList<AlignmentPattern>(5);
    this.startX = startX;
    this.startY = startY;
    this.width = width;
    this.height = height;
    this.moduleSize = moduleSize;
    this.crossCheckStateCount = new int[3];
    this.resultPointCallback = resultPointCallback;
  }

  /**
   * <p>This method attempts to find the bottom-right alignment pattern in the image.
It is a bit messy since
   * it's pretty performance-critical and so is written to be fast foremost.</p>
   *
   * @return {@link AlignmentPattern} if found
   * @throws NotFoundException if not found
   */
  AlignmentPattern find() throws NotFoundException {
    int startX = this.startX;
    int height = this.height;
    int maxJ = startX + width;
    int middleI = startY + (height >> 1);
    // We are looking for black/white/black modules in 1:1:1 ratio;
    // this tracks the number of black/white/black modules seen so far
    int[] stateCount = new int[3];
    for (int iGen = 0; iGen < height; iGen++) {
      // Search from middle outwards
      int i = middleI + ((iGen & 0x01) == 0 ? (iGen + 1) >> 1 : -((iGen + 1) >> 1));
      stateCount[0] = 0;
      stateCount[1] = 0;
      stateCount[2] = 0;
      int j = startX;
      // Burn off leading white pixels before anything else; if we start in the middle
of
      // a white run, it doesn't make sense to count its length, since we don't know
if the
      // white run continued to the left of the start point
      while (j < maxJ && !image.get(j, i)) {
        j++;
      }
      int currentState = 0;
      while (j < maxJ) {
        if (image.get(j, i)) {
          // Black pixel
```

```
          if (currentState == 1) { // Counting black pixels
            stateCount[currentState]++;
          } else { // Counting white pixels
            if (currentState == 2) { // A winner?
              if (foundPatternCross(stateCount)) { // Yes
                AlignmentPattern confirmed = handlePossibleCenter(stateCount, i, j);
                if (confirmed != null) {
                  return confirmed;
                }
              }
              stateCount[0] = stateCount[2];
              stateCount[1] = 1;
              stateCount[2] = 0;
              currentState = 1;
            } else {
              stateCount[++currentState]++;
            }
          }
        } else { // White pixel
          if (currentState == 1) { // Counting black pixels
            currentState++;
          }
          stateCount[currentState]++;
        }
        j++;
      }
      if (foundPatternCross(stateCount)) {
        AlignmentPattern confirmed = handlePossibleCenter(stateCount, i, maxJ);
        if (confirmed != null) {
          return confirmed;
        }
      }

    }

    // Hmm, nothing we saw was observed and confirmed twice. If we had
    // any guess at all, return it.
    if (!possibleCenters.isEmpty()) {
      return possibleCenters.get(0);
    }

    throw NotFoundException.getNotFoundInstance();
  }

  /**
   * Given a count of black/white/black pixels just seen and an end position,
   * figures the location of the center of this black/white/black run.
   */
  private static float centerFromEnd(int[] stateCount, int end) {
    return (float) (end - stateCount[2]) - stateCount[1] / 2.0f;
  }

  /**
   * @param stateCount count of black/white/black pixels just read
   * @return true iff the proportions of the counts is close enough to the 1/1/1
ratios
   *         used by alignment patterns to be considered a match
   */
  private boolean foundPatternCross(int[] stateCount) {
    float moduleSize = this.moduleSize;
    float maxVariance = moduleSize / 2.0f;
    for (int i = 0; i < 3; i++) {
      if (Math.abs(moduleSize - stateCount[i]) >= maxVariance) {
```

```
      return false;
    }
  }
  return true;
}

/**
 * <p>After a horizontal scan finds a potential alignment pattern, this method
 * "cross-checks" by scanning down vertically through the center of the possible
 * alignment pattern to see if the same proportion is detected.</p>
 *
 * @param startI row where an alignment pattern was detected
 * @param centerJ center of the section that appears to cross an alignment pattern
 * @param maxCount maximum reasonable number of modules that should be
 * observed in any reading state, based on the results of the horizontal scan
 * @return vertical center of alignment pattern, or {@link Float#NaN} if not found
 */
private float crossCheckVertical(int startI, int centerJ, int maxCount,
    int originalStateCountTotal) {
  BitMatrix image = this.image;

  int maxI = image.getHeight();
  int[] stateCount = crossCheckStateCount;
  stateCount[0] = 0;
  stateCount[1] = 0;
  stateCount[2] = 0;

  // Start counting up from center
  int i = startI;
  while (i >= 0 && image.get(centerJ, i) && stateCount[1] <= maxCount) {
    stateCount[1]++;
    i--;
  }
  // If already too many modules in this state or ran off the edge:
  if (i < 0 || stateCount[1] > maxCount) {
    return Float.NaN;
  }
  while (i >= 0 && !image.get(centerJ, i) && stateCount[0] <= maxCount) {
    stateCount[0]++;
    i--;
  }
  if (stateCount[0] > maxCount) {
    return Float.NaN;
  }

  // Now also count down from center
  i = startI + 1;
  while (i < maxI && image.get(centerJ, i) && stateCount[1] <= maxCount) {
    stateCount[1]++;
    i++;
  }
  if (i == maxI || stateCount[1] > maxCount) {
    return Float.NaN;
  }
  while (i < maxI && !image.get(centerJ, i) && stateCount[2] <= maxCount) {
    stateCount[2]++;
    i++;
  }
  if (stateCount[2] > maxCount) {
    return Float.NaN;
  }

  int stateCountTotal = stateCount[0] + stateCount[1] + stateCount[2];
```

```
    if (5 * Math.abs(stateCountTotal - originalStateCountTotal) >= 2 *
originalStateCountTotal) {
      return Float.NaN;
    }

    return foundPatternCross(stateCount) ? centerFromEnd(stateCount, i) : Float.NaN;
  }

  /**
   * <p>This is called when a horizontal scan finds a possible alignment pattern. It
will
   * cross check with a vertical scan, and if successful, will see if this pattern had
been
   * found on a previous horizontal scan. If so, we consider it confirmed and conclude
we have
   * found the alignment pattern.</p>
   *
   * @param stateCount reading state module counts from horizontal scan
   * @param i row where alignment pattern may be found
   * @param j end of possible alignment pattern in row
   * @return {@link AlignmentPattern} if we have found the same pattern twice, or null
if not
   */
  private AlignmentPattern handlePossibleCenter(int[] stateCount, int i, int j) {
    int stateCountTotal = stateCount[0] + stateCount[1] + stateCount[2];
    float centerJ = centerFromEnd(stateCount, j);
    float centerI = crossCheckVertical(i, (int) centerJ, 2 * stateCount[1],
stateCountTotal);
    if (!Float.isNaN(centerI)) {
      float estimatedModuleSize = (float) (stateCount[0] + stateCount[1] +
stateCount[2]) / 3.0f;
      for (AlignmentPattern center : possibleCenters) {
        // Look for about the same center and module size:
        if (center.aboutEquals(estimatedModuleSize, centerI, centerJ)) {
          return center.combineEstimate(centerI, centerJ, estimatedModuleSize);
        }
      }
      // Hadn't found this before; save it
      AlignmentPattern point = new AlignmentPattern(centerJ, centerI,
estimatedModuleSize);
      possibleCenters.add(point);
      if (resultPointCallback != null) {
        resultPointCallback.foundPossibleResultPoint(point);
      }
    }
    return null;
  }

}
```

- **FinderPattern.java**

```
public final class FinderPattern extends ResultPoint {

  private final float estimatedModuleSize;
  private int count;

  FinderPattern(float posX, float posY, float estimatedModuleSize) {
    this(posX, posY, estimatedModuleSize, 1);
  }

  private FinderPattern(float posX, float posY, float estimatedModuleSize, int count)
{
```

```
    super(posX, posY);
    this.estimatedModuleSize = estimatedModuleSize;
    this.count = count;
  }

  public float getEstimatedModuleSize() {
    return estimatedModuleSize;
  }

  int getCount() {
    return count;
  }

  void incrementCount() {
    this.count++;
  }

  /**
   * <p>Determines if this finder pattern "about equals" a finder pattern at the
stated
   * position and size -- meaning, it is at nearly the same center with nearly the
same size.</p>
   */
  boolean aboutEquals(float moduleSize, float i, float j) {
    if (Math.abs(i - getY()) <= moduleSize && Math.abs(j - getX()) <= moduleSize) {
      float moduleSizeDiff = Math.abs(moduleSize - estimatedModuleSize);
      return moduleSizeDiff <= 1.0f || moduleSizeDiff <= estimatedModuleSize;
    }
    return false;
  }

  /**
   * Combines this object's current estimate of a finder pattern position and module
size
   * with a new estimate. It returns a new {@code FinderPattern} containing a weighted
average
   * based on count.
   */
  FinderPattern combineEstimate(float i, float j, float newModuleSize) {
    int combinedCount = count + 1;
    float combinedX = (count * getX() + j) / combinedCount;
    float combinedY = (count * getY() + i) / combinedCount;
    float combinedModuleSize = (count * estimatedModuleSize + newModuleSize) /
combinedCount;
    return new FinderPattern(combinedX, combinedY, combinedModuleSize, combinedCount);
  }

}
```

- **DataMask.java**

```
private static final DataMask[] DATA_MASKS = {
      new DataMask000(),
      new DataMask001(),
      new DataMask010(),
      new DataMask011(),
      new DataMask100(),
      new DataMask101(),
      new DataMask110(),
      new DataMask111(),
  };

  private DataMask() {
```

```
  }

  /**
   * <p>Implementations of this method reverse the data masking process applied to a
QR Code and
   * make its bits ready to read.</p>
   *
   * @param bits representation of QR Code bits
   * @param dimension dimension of QR Code, represented by bits, being unmasked
   */
  final void unmaskBitMatrix(BitMatrix bits, int dimension) {
    for (int i = 0; i < dimension; i++) {
      for (int j = 0; j < dimension; j++) {
        if (isMasked(i, j)) {
          bits.flip(j, i);
        }
      }
    }
  }

  abstract boolean isMasked(int i, int j);

  /**
   * @param reference a value between 0 and 7 indicating one of the eight possible
   * data mask patterns a QR Code may use
   * @return DataMask encapsulating the data mask pattern
   */
  static DataMask forReference(int reference) {
    if (reference < 0 || reference > 7) {
      throw new IllegalArgumentException();
    }
    return DATA_MASKS[reference];
  }

  /**
   * 000: mask bits for which (x + y) mod 2 == 0
   */
  private static final class DataMask000 extends DataMask {
    @Override
    boolean isMasked(int i, int j) {
      return ((i + j) & 0x01) == 0;
    }
  }

  /**
   * 001: mask bits for which x mod 2 == 0
   */
  private static final class DataMask001 extends DataMask {
    @Override
    boolean isMasked(int i, int j) {
      return (i & 0x01) == 0;
    }
  }

  /**
   * 010: mask bits for which y mod 3 == 0
   */
  private static final class DataMask010 extends DataMask {
    @Override
    boolean isMasked(int i, int j) {
      return j % 3 == 0;
    }
  }
```

```
  /**
   * 011: mask bits for which (x + y) mod 3 == 0
   */
  private static final class DataMask011 extends DataMask {
    @Override
    boolean isMasked(int i, int j) {
      return (i + j) % 3 == 0;
    }
  }

  /**
   * 100: mask bits for which (x/2 + y/3) mod 2 == 0
   */
  private static final class DataMask100 extends DataMask {
    @Override
    boolean isMasked(int i, int j) {
      return (((i >>> 1) + (j /3)) & 0x01) == 0;
    }
  }

  /**
   * 101: mask bits for which xy mod 2 + xy mod 3 == 0
   */
  private static final class DataMask101 extends DataMask {
    @Override
    boolean isMasked(int i, int j) {
      int temp = i * j;
      return (temp & 0x01) + (temp % 3) == 0;
    }
  }

  /**
   * 110: mask bits for which (xy mod 2 + xy mod 3) mod 2 == 0
   */
  private static final class DataMask110 extends DataMask {
    @Override
    boolean isMasked(int i, int j) {
      int temp = i * j;
      return (((temp & 0x01) + (temp % 3)) & 0x01) == 0;
    }
  }

  /**
   * 111: mask bits for which ((x+y)mod 2 + xy mod 3) mod 2 == 0
   */
  private static final class DataMask111 extends DataMask {
    @Override
    boolean isMasked(int i, int j) {
      return ((((i + j) & 0x01) + ((i * j) % 3)) & 0x01) == 0;
    }
  }
}
```

- **ErrorCorrectionLevel.java**

```
public enum ErrorCorrectionLevel {

  /** L = ~7% correction */
  L(0x01),
  /** M = ~15% correction */
  M(0x00),
  /** Q = ~25% correction */
```

```
  Q(0x03),
  /** H = ~30% correction */
  H(0x02);

  private static final ErrorCorrectionLevel[] FOR_BITS = {M, L, H, Q};

  private final int bits;

  ErrorCorrectionLevel(int bits) {
    this.bits = bits;
  }

  public int getBits() {
    return bits;
  }

  /**
   * @param bits int containing the two bits encoding a QR Code's error correction
level
   * @return ErrorCorrectionLevel representing the encoded error correction level
   */
  public static ErrorCorrectionLevel forBits(int bits) {
    if (bits < 0 || bits >= FOR_BITS.length) {
      throw new IllegalArgumentException();
    }
    return FOR_BITS[bits];
  }


}
```

- **FormatInformation.java**

```
final class FormatInformation {

  private static final int FORMAT_INFO_MASK_QR = 0x5412;

  /**
   * See ISO 18004:2006, Annex C, Table C.1
   */
  private static final int[][] FORMAT_INFO_DECODE_LOOKUP = {
      {0x5412, 0x00},
      {0x5125, 0x01},
      {0x5E7C, 0x02},
      {0x5B4B, 0x03},
      {0x45F9, 0x04},
      {0x40CE, 0x05},
      {0x4F97, 0x06},
      {0x4AA0, 0x07},
      {0x77C4, 0x08},
      {0x72F3, 0x09},
      {0x7DAA, 0x0A},
      {0x789D, 0x0B},
      {0x662F, 0x0C},
      {0x6318, 0x0D},
      {0x6C41, 0x0E},
      {0x6976, 0x0F},
      {0x1689, 0x10},
```

```
      {0x13BE, 0x11},
      {0x1CE7, 0x12},
      {0x19D0, 0x13},
      {0x0762, 0x14},
      {0x0255, 0x15},
      {0x0D0C, 0x16},
      {0x083B, 0x17},
      {0x355F, 0x18},
      {0x3068, 0x19},
      {0x3F31, 0x1A},
      {0x3A06, 0x1B},
      {0x24B4, 0x1C},
      {0x2183, 0x1D},
      {0x2EDA, 0x1E},
      {0x2BED, 0x1F},
  };

  /**
   * Offset i holds the number of 1 bits in the binary representation of i
   */
  private static final int[] BITS_SET_IN_HALF_BYTE =
      {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};

  private final ErrorCorrectionLevel errorCorrectionLevel;
  private final byte dataMask;

  private FormatInformation(int formatInfo) {
    // Bits 3,4
    errorCorrectionLevel = ErrorCorrectionLevel.forBits((formatInfo >> 3) & 0x03);
    // Bottom 3 bits
    dataMask = (byte) (formatInfo & 0x07);
  }

  static int numBitsDiffering(int a, int b) {
    a ^= b; // a now has a 1 bit exactly where its bit differs with b's
    // Count bits set quickly with a series of lookups:
    return BITS_SET_IN_HALF_BYTE[a & 0x0F] +
        BITS_SET_IN_HALF_BYTE[(a >>> 4 & 0x0F)] +
        BITS_SET_IN_HALF_BYTE[(a >>> 8 & 0x0F)] +
        BITS_SET_IN_HALF_BYTE[(a >>> 12 & 0x0F)] +
        BITS_SET_IN_HALF_BYTE[(a >>> 16 & 0x0F)] +
        BITS_SET_IN_HALF_BYTE[(a >>> 20 & 0x0F)] +
        BITS_SET_IN_HALF_BYTE[(a >>> 24 & 0x0F)] +
        BITS_SET_IN_HALF_BYTE[(a >>> 28 & 0x0F)];
  }

  /**
   * @param maskedFormatInfo1 format info indicator, with mask still applied
   * @param maskedFormatInfo2 second copy of same info; both are checked at the same
time
   *  to establish best match
   * @return information about the format it specifies, or {@code null}
   *  if doesn't seem to match any known pattern
   */
  static FormatInformation decodeFormatInformation(int maskedFormatInfo1, int
maskedFormatInfo2) {
    FormatInformation formatInfo = doDecodeFormatInformation(maskedFormatInfo1,
maskedFormatInfo2);
    if (formatInfo != null) {
      return formatInfo;
    }
    // Should return null, but, some QR codes apparently
    // do not mask this info. Try again by actually masking the pattern
```

```
    // first
    return doDecodeFormatInformation(maskedFormatInfo1 ^ FORMAT_INFO_MASK_QR,
                                     maskedFormatInfo2 ^ FORMAT_INFO_MASK_QR);
  }

  private static FormatInformation doDecodeFormatInformation(int maskedFormatInfo1,
int maskedFormatInfo2) {
    // Find the int in FORMAT_INFO_DECODE_LOOKUP with fewest bits differing
    int bestDifference = Integer.MAX_VALUE;
    int bestFormatInfo = 0;
    for (int[] decodeInfo : FORMAT_INFO_DECODE_LOOKUP) {
      int targetInfo = decodeInfo[0];
      if (targetInfo == maskedFormatInfo1 || targetInfo == maskedFormatInfo2) {
        // Found an exact match
        return new FormatInformation(decodeInfo[1]);
      }
      int bitsDifference = numBitsDiffering(maskedFormatInfo1, targetInfo);
      if (bitsDifference < bestDifference) {
        bestFormatInfo = decodeInfo[1];
        bestDifference = bitsDifference;
      }
      if (maskedFormatInfo1 != maskedFormatInfo2) {
        // also try the other option
        bitsDifference = numBitsDiffering(maskedFormatInfo2, targetInfo);
        if (bitsDifference < bestDifference) {
          bestFormatInfo = decodeInfo[1];
          bestDifference = bitsDifference;
        }
      }
    }
    // Hamming distance of the 32 masked codes is 7, by construction, so <= 3 bits
    // differing means we found a match
    if (bestDifference <= 3) {
      return new FormatInformation(bestFormatInfo);
    }
    return null;
  }

  ErrorCorrectionLevel getErrorCorrectionLevel() {
    return errorCorrectionLevel;
  }

  byte getDataMask() {
    return dataMask;
  }

  @Override
  public int hashCode() {
    return (errorCorrectionLevel.ordinal() << 3) | (int) dataMask;
  }

  @Override
  public boolean equals(Object o) {
    if (!(o instanceof FormatInformation)) {
      return false;
    }
    FormatInformation other = (FormatInformation) o;
    return this.errorCorrectionLevel == other.errorCorrectionLevel &&
        this.dataMask == other.dataMask;
  }

}
```

- **Version.java**

```
public final class Version {

  /**
   * See ISO 18004:2006 Annex D.
   * Element i represents the raw version bits that specify version i + 7
   */
  private static final int[] VERSION_DECODE_INFO = {
      0x07C94, 0x085BC, 0x09A99, 0x0A4D3, 0x0BBF6,
      0x0C762, 0x0D847, 0x0E60D, 0x0F928, 0x10B78,
      0x1145D, 0x12A17, 0x13532, 0x149A6, 0x15683,
      0x168C9, 0x177EC, 0x18EC4, 0x191E1, 0x1AFAB,
      0x1B08E, 0x1CC1A, 0x1D33F, 0x1ED75, 0x1F250,
      0x209D5, 0x216F0, 0x228BA, 0x2379F, 0x24B0B,
      0x2542E, 0x26A64, 0x27541, 0x28C69
  };

  private static final Version[] VERSIONS = buildVersions();

  private final int versionNumber;
  private final int[] alignmentPatternCenters;
  private final ECBlocks[] ecBlocks;
  private final int totalCodewords;

  private Version(int versionNumber,
                  int[] alignmentPatternCenters,
                  ECBlocks... ecBlocks) {
    this.versionNumber = versionNumber;
    this.alignmentPatternCenters = alignmentPatternCenters;
    this.ecBlocks = ecBlocks;
    int total = 0;
    int ecCodewords = ecBlocks[0].getECCodewordsPerBlock();
    ECB[] ecbArray = ecBlocks[0].getECBlocks();
    for (ECB ecBlock : ecbArray) {
      total += ecBlock.getCount() * (ecBlock.getDataCodewords() + ecCodewords);
    }
    this.totalCodewords = total;
  }

  public int getVersionNumber() {
    return versionNumber;
  }

  public int[] getAlignmentPatternCenters() {
    return alignmentPatternCenters;
  }

  public int getTotalCodewords() {
    return totalCodewords;
  }

  public int getDimensionForVersion() {
    return 17 + 4 * versionNumber;
  }

  public ECBlocks getECBlocksForLevel(ErrorCorrectionLevel ecLevel) {
    return ecBlocks[ecLevel.ordinal()];
  }

  /**
   * <p>Deduces version information purely from QR Code dimensions.</p>
   *
```

```
   * @param dimension dimension in modules
   * @return Version for a QR Code of that dimension
   * @throws FormatException if dimension is not 1 mod 4
   */
  public static Version getProvisionalVersionForDimension(int dimension) throws
FormatException {
    if (dimension % 4 != 1) {
      throw FormatException.getFormatInstance();
    }
    try {
      return getVersionForNumber((dimension - 17) >> 2);
    } catch (IllegalArgumentException iae) {
      throw FormatException.getFormatInstance();
    }
  }

  public static Version getVersionForNumber(int versionNumber) {
    if (versionNumber < 1 || versionNumber > 40) {
      throw new IllegalArgumentException();
    }
    return VERSIONS[versionNumber - 1];
  }

  static Version decodeVersionInformation(int versionBits) {
    int bestDifference = Integer.MAX_VALUE;
    int bestVersion = 0;
    for (int i = 0; i < VERSION_DECODE_INFO.length; i++) {
      int targetVersion = VERSION_DECODE_INFO[i];
      // Do the version info bits match exactly? done.
      if (targetVersion == versionBits) {
        return getVersionForNumber(i + 7);
      }
      // Otherwise see if this is the closest to a real version info bit string
      // we have seen so far
      int bitsDifference = FormatInformation.numBitsDiffering(versionBits,
targetVersion);
      if (bitsDifference < bestDifference) {
        bestVersion = i + 7;
        bestDifference = bitsDifference;
      }
    }
    // We can tolerate up to 3 bits of error since no two version info codewords will
    // differ in less than 8 bits.
    if (bestDifference <= 3) {
      return getVersionForNumber(bestVersion);
    }
    // If we didn't find a close enough match, fail
    return null;
  }

  /**
   * See ISO 18004:2006 Annex E
   */
  BitMatrix buildFunctionPattern() {
    int dimension = getDimensionForVersion();
    BitMatrix bitMatrix = new BitMatrix(dimension);

    // Top left finder pattern + separator + format
    bitMatrix.setRegion(0, 0, 9, 9);
    // Top right finder pattern + separator + format
    bitMatrix.setRegion(dimension - 8, 0, 8, 9);
    // Bottom left finder pattern + separator + format
    bitMatrix.setRegion(0, dimension - 8, 9, 8);
```

```
    // Alignment patterns
    int max = alignmentPatternCenters.length;
    for (int x = 0; x < max; x++) {
      int i = alignmentPatternCenters[x] - 2;
      for (int y = 0; y < max; y++) {
        if ((x == 0 && (y == 0 || y == max - 1)) || (x == max - 1 && y == 0)) {
          // No alignment patterns near the three finder paterns
          continue;
        }
        bitMatrix.setRegion(alignmentPatternCenters[y] - 2, i, 5, 5);
      }
    }

    // Vertical timing pattern
    bitMatrix.setRegion(6, 9, 1, dimension - 17);
    // Horizontal timing pattern
    bitMatrix.setRegion(9, 6, dimension - 17, 1);

    if (versionNumber > 6) {
      // Version info, top right
      bitMatrix.setRegion(dimension - 11, 0, 3, 6);
      // Version info, bottom left
      bitMatrix.setRegion(0, dimension - 11, 6, 3);
    }

    return bitMatrix;
  }

  /**
   * <p>Encapsulates a set of error-correction blocks in one symbol version. Most
versions will
   * use blocks of differing sizes within one version, so, this encapsulates the
parameters for
   * each set of blocks. It also holds the number of error-correction codewords per
block since it
   * will be the same across all blocks within one version.</p>
   */
  public static final class ECBlocks {
    private final int ecCodewordsPerBlock;
    private final ECB[] ecBlocks;

    ECBlocks(int ecCodewordsPerBlock, ECB... ecBlocks) {
      this.ecCodewordsPerBlock = ecCodewordsPerBlock;
      this.ecBlocks = ecBlocks;
    }

    public int getECCodewordsPerBlock() {
      return ecCodewordsPerBlock;
    }

    public int getNumBlocks() {
      int total = 0;
      for (ECB ecBlock : ecBlocks) {
        total += ecBlock.getCount();
      }
      return total;
    }

    public int getTotalECCodewords() {
      return ecCodewordsPerBlock * getNumBlocks();
    }
```

```
    public ECB[] getECBlocks() {
      return ecBlocks;
    }
  }

  /**
   * <p>Encapsualtes the parameters for one error-correction block in one symbol
version.
   * This includes the number of data codewords, and the number of times a block with
these
   * parameters is used consecutively in the QR code version's format.</p>
   */
  public static final class ECB {
    private final int count;
    private final int dataCodewords;

    ECB(int count, int dataCodewords) {
      this.count = count;
      this.dataCodewords = dataCodewords;
    }

    public int getCount() {
      return count;
    }

    public int getDataCodewords() {
      return dataCodewords;
    }
  }

  @Override
  public String toString() {
    return String.valueOf(versionNumber);
  }

  /**
   * See ISO 18004:2006 6.5.1 Table 9
   */
  private static Version[] buildVersions() {
    return new Version[]{
        new Version(1, new int[]{},
            new ECBlocks(7, new ECB(1, 19)),
            new ECBlocks(10, new ECB(1, 16)),
            new ECBlocks(13, new ECB(1, 13)),
            new ECBlocks(17, new ECB(1, 9))),
        new Version(2, new int[]{6, 18},
            new ECBlocks(10, new ECB(1, 34)),
            new ECBlocks(16, new ECB(1, 28)),
            new ECBlocks(22, new ECB(1, 22)),
            new ECBlocks(28, new ECB(1, 16))),
        new Version(3, new int[]{6, 22},
            new ECBlocks(15, new ECB(1, 55)),
            new ECBlocks(26, new ECB(1, 44)),
            new ECBlocks(18, new ECB(2, 17)),
            new ECBlocks(22, new ECB(2, 13))),
        new Version(4, new int[]{6, 26},
            new ECBlocks(20, new ECB(1, 80)),
            new ECBlocks(18, new ECB(2, 32)),
            new ECBlocks(26, new ECB(2, 24)),
            new ECBlocks(16, new ECB(4, 9))),
        new Version(5, new int[]{6, 30},
            new ECBlocks(26, new ECB(1, 108)),
            new ECBlocks(24, new ECB(2, 43)),
```

```
        new ECBlocks(18, new ECB(2, 15),
            new ECB(2, 16)),
        new ECBlocks(22, new ECB(2, 11),
            new ECB(2, 12))),
    new Version(6, new int[]{6, 34},
        new ECBlocks(18, new ECB(2, 68)),
        new ECBlocks(16, new ECB(4, 27)),
        new ECBlocks(24, new ECB(4, 19)),
        new ECBlocks(28, new ECB(4, 15))),
    new Version(7, new int[]{6, 22, 38},
        new ECBlocks(20, new ECB(2, 78)),
        new ECBlocks(18, new ECB(4, 31)),
        new ECBlocks(18, new ECB(2, 14),
            new ECB(4, 15)),
        new ECBlocks(26, new ECB(4, 13),
            new ECB(1, 14))),
    new Version(8, new int[]{6, 24, 42},
        new ECBlocks(24, new ECB(2, 97)),
        new ECBlocks(22, new ECB(2, 38),
            new ECB(2, 39)),
        new ECBlocks(22, new ECB(4, 18),
            new ECB(2, 19)),
        new ECBlocks(26, new ECB(4, 14),
            new ECB(2, 15))),
    new Version(9, new int[]{6, 26, 46},
        new ECBlocks(30, new ECB(2, 116)),
        new ECBlocks(22, new ECB(3, 36),
            new ECB(2, 37)),
        new ECBlocks(20, new ECB(4, 16),
            new ECB(4, 17)),
        new ECBlocks(24, new ECB(4, 12),
            new ECB(4, 13))),
    new Version(10, new int[]{6, 28, 50},
        new ECBlocks(18, new ECB(2, 68),
            new ECB(2, 69)),
        new ECBlocks(26, new ECB(4, 43),
            new ECB(1, 44)),
        new ECBlocks(24, new ECB(6, 19),
            new ECB(2, 20)),
        new ECBlocks(28, new ECB(6, 15),
            new ECB(2, 16))),
    new Version(11, new int[]{6, 30, 54},
        new ECBlocks(20, new ECB(4, 81)),
        new ECBlocks(30, new ECB(1, 50),
            new ECB(4, 51)),
        new ECBlocks(28, new ECB(4, 22),
            new ECB(4, 23)),
        new ECBlocks(24, new ECB(3, 12),
            new ECB(8, 13))),
    new Version(12, new int[]{6, 32, 58},
        new ECBlocks(24, new ECB(2, 92),
            new ECB(2, 93)),
        new ECBlocks(22, new ECB(6, 36),
            new ECB(2, 37)),
        new ECBlocks(26, new ECB(4, 20),
            new ECB(6, 21)),
        new ECBlocks(28, new ECB(7, 14),
            new ECB(4, 15))),
    new Version(13, new int[]{6, 34, 62},
        new ECBlocks(26, new ECB(4, 107)),
        new ECBlocks(22, new ECB(8, 37),
            new ECB(1, 38)),
        new ECBlocks(24, new ECB(8, 20),
```

```
            new ECB(4, 21)),
        new ECBlocks(22, new ECB(12, 11),
            new ECB(4, 12))),
    new Version(14, new int[]{6, 26, 46, 66},
        new ECBlocks(30, new ECB(3, 115),
            new ECB(1, 116)),
        new ECBlocks(24, new ECB(4, 40),
            new ECB(5, 41)),
        new ECBlocks(20, new ECB(11, 16),
            new ECB(5, 17)),
        new ECBlocks(24, new ECB(11, 12),
            new ECB(5, 13))),
    new Version(15, new int[]{6, 26, 48, 70},
        new ECBlocks(22, new ECB(5, 87),
            new ECB(1, 88)),
        new ECBlocks(24, new ECB(5, 41),
            new ECB(5, 42)),
        new ECBlocks(30, new ECB(5, 24),
            new ECB(7, 25)),
        new ECBlocks(24, new ECB(11, 12),
            new ECB(7, 13))),
    new Version(16, new int[]{6, 26, 50, 74},
        new ECBlocks(24, new ECB(5, 98),
            new ECB(1, 99)),
        new ECBlocks(28, new ECB(7, 45),
            new ECB(3, 46)),
        new ECBlocks(24, new ECB(15, 19),
            new ECB(2, 20)),
        new ECBlocks(30, new ECB(3, 15),
            new ECB(13, 16))),
    new Version(17, new int[]{6, 30, 54, 78},
        new ECBlocks(28, new ECB(1, 107),
            new ECB(5, 108)),
        new ECBlocks(28, new ECB(10, 46),
            new ECB(1, 47)),
        new ECBlocks(28, new ECB(1, 22),
            new ECB(15, 23)),
        new ECBlocks(28, new ECB(2, 14),
            new ECB(17, 15))),
    new Version(18, new int[]{6, 30, 56, 82},
        new ECBlocks(30, new ECB(5, 120),
            new ECB(1, 121)),
        new ECBlocks(26, new ECB(9, 43),
            new ECB(4, 44)),
        new ECBlocks(28, new ECB(17, 22),
            new ECB(1, 23)),
        new ECBlocks(28, new ECB(2, 14),
            new ECB(19, 15))),
    new Version(19, new int[]{6, 30, 58, 86},
        new ECBlocks(28, new ECB(3, 113),
            new ECB(4, 114)),
        new ECBlocks(26, new ECB(3, 44),
            new ECB(11, 45)),
        new ECBlocks(26, new ECB(17, 21),
            new ECB(4, 22)),
        new ECBlocks(26, new ECB(9, 13),
            new ECB(16, 14))),
    new Version(20, new int[]{6, 34, 62, 90},
        new ECBlocks(28, new ECB(3, 107),
            new ECB(5, 108)),
        new ECBlocks(26, new ECB(3, 41),
            new ECB(13, 42)),
        new ECBlocks(30, new ECB(15, 24),
```

```
            new ECB(5, 25)),
        new ECBlocks(28, new ECB(15, 15),
            new ECB(10, 16))),
    new Version(21, new int[]{6, 28, 50, 72, 94},
        new ECBlocks(28, new ECB(4, 116),
            new ECB(4, 117)),
        new ECBlocks(26, new ECB(17, 42)),
        new ECBlocks(28, new ECB(17, 22),
            new ECB(6, 23)),
        new ECBlocks(30, new ECB(19, 16),
            new ECB(6, 17))),
    new Version(22, new int[]{6, 26, 50, 74, 98},
        new ECBlocks(28, new ECB(2, 111),
            new ECB(7, 112)),
        new ECBlocks(28, new ECB(17, 46)),
        new ECBlocks(30, new ECB(7, 24),
            new ECB(16, 25)),
        new ECBlocks(24, new ECB(34, 13))),
    new Version(23, new int[]{6, 30, 54, 78, 102},
        new ECBlocks(30, new ECB(4, 121),
            new ECB(5, 122)),
        new ECBlocks(28, new ECB(4, 47),
            new ECB(14, 48)),
        new ECBlocks(30, new ECB(11, 24),
            new ECB(14, 25)),
        new ECBlocks(30, new ECB(16, 15),
            new ECB(14, 16))),
    new Version(24, new int[]{6, 28, 54, 80, 106},
        new ECBlocks(30, new ECB(6, 117),
            new ECB(4, 118)),
        new ECBlocks(28, new ECB(6, 45),
            new ECB(14, 46)),
        new ECBlocks(30, new ECB(11, 24),
            new ECB(16, 25)),
        new ECBlocks(30, new ECB(30, 16),
            new ECB(2, 17))),
    new Version(25, new int[]{6, 32, 58, 84, 110},
        new ECBlocks(26, new ECB(8, 106),
            new ECB(4, 107)),
        new ECBlocks(28, new ECB(8, 47),
            new ECB(13, 48)),
        new ECBlocks(30, new ECB(7, 24),
            new ECB(22, 25)),
        new ECBlocks(30, new ECB(22, 15),
            new ECB(13, 16))),
    new Version(26, new int[]{6, 30, 58, 86, 114},
        new ECBlocks(28, new ECB(10, 114),
            new ECB(2, 115)),
        new ECBlocks(28, new ECB(19, 46),
            new ECB(4, 47)),
        new ECBlocks(28, new ECB(28, 22),
            new ECB(6, 23)),
        new ECBlocks(30, new ECB(33, 16),
            new ECB(4, 17))),
    new Version(27, new int[]{6, 34, 62, 90, 118},
        new ECBlocks(30, new ECB(8, 122),
            new ECB(4, 123)),
        new ECBlocks(28, new ECB(22, 45),
            new ECB(3, 46)),
        new ECBlocks(30, new ECB(8, 23),
            new ECB(26, 24)),
        new ECBlocks(30, new ECB(12, 15),
            new ECB(28, 16))),
```

```
new Version(28, new int[]{6, 26, 50, 74, 98, 122},
    new ECBlocks(30, new ECB(3, 117),
        new ECB(10, 118)),
    new ECBlocks(28, new ECB(3, 45),
        new ECB(23, 46)),
    new ECBlocks(30, new ECB(4, 24),
        new ECB(31, 25)),
    new ECBlocks(30, new ECB(11, 15),
        new ECB(31, 16))),
new Version(29, new int[]{6, 30, 54, 78, 102, 126},
    new ECBlocks(30, new ECB(7, 116),
        new ECB(7, 117)),
    new ECBlocks(28, new ECB(21, 45),
        new ECB(7, 46)),
    new ECBlocks(30, new ECB(1, 23),
        new ECB(37, 24)),
    new ECBlocks(30, new ECB(19, 15),
        new ECB(26, 16))),
new Version(30, new int[]{6, 26, 52, 78, 104, 130},
    new ECBlocks(30, new ECB(5, 115),
        new ECB(10, 116)),
    new ECBlocks(28, new ECB(19, 47),
        new ECB(10, 48)),
    new ECBlocks(30, new ECB(15, 24),
        new ECB(25, 25)),
    new ECBlocks(30, new ECB(23, 15),
        new ECB(25, 16))),
new Version(31, new int[]{6, 30, 56, 82, 108, 134},
    new ECBlocks(30, new ECB(13, 115),
        new ECB(3, 116)),
    new ECBlocks(28, new ECB(2, 46),
        new ECB(29, 47)),
    new ECBlocks(30, new ECB(42, 24),
        new ECB(1, 25)),
    new ECBlocks(30, new ECB(23, 15),
        new ECB(28, 16))),
new Version(32, new int[]{6, 34, 60, 86, 112, 138},
    new ECBlocks(30, new ECB(17, 115)),
    new ECBlocks(28, new ECB(10, 46),
        new ECB(23, 47)),
    new ECBlocks(30, new ECB(10, 24),
        new ECB(35, 25)),
    new ECBlocks(30, new ECB(19, 15),
        new ECB(35, 16))),
new Version(33, new int[]{6, 30, 58, 86, 114, 142},
    new ECBlocks(30, new ECB(17, 115),
        new ECB(1, 116)),
    new ECBlocks(28, new ECB(14, 46),
        new ECB(21, 47)),
    new ECBlocks(30, new ECB(29, 24),
        new ECB(19, 25)),
    new ECBlocks(30, new ECB(11, 15),
        new ECB(46, 16))),
new Version(34, new int[]{6, 34, 62, 90, 118, 146},
    new ECBlocks(30, new ECB(13, 115),
        new ECB(6, 116)),
    new ECBlocks(28, new ECB(14, 46),
        new ECB(23, 47)),
    new ECBlocks(30, new ECB(44, 24),
        new ECB(7, 25)),
    new ECBlocks(30, new ECB(59, 16),
        new ECB(1, 17))),
new Version(35, new int[]{6, 30, 54, 78, 102, 126, 150},
```

```
            new ECBlocks(30, new ECB(12, 121),
                new ECB(7, 122)),
            new ECBlocks(28, new ECB(12, 47),
                new ECB(26, 48)),
            new ECBlocks(30, new ECB(39, 24),
                new ECB(14, 25)),
            new ECBlocks(30, new ECB(22, 15),
                new ECB(41, 16))),
        new Version(36, new int[]{6, 24, 50, 76, 102, 128, 154},
            new ECBlocks(30, new ECB(6, 121),
                new ECB(14, 122)),
            new ECBlocks(28, new ECB(6, 47),
                new ECB(34, 48)),
            new ECBlocks(30, new ECB(46, 24),
                new ECB(10, 25)),
            new ECBlocks(30, new ECB(2, 15),
                new ECB(64, 16))),
        new Version(37, new int[]{6, 28, 54, 80, 106, 132, 158},
            new ECBlocks(30, new ECB(17, 122),
                new ECB(4, 123)),
            new ECBlocks(28, new ECB(29, 46),
                new ECB(14, 47)),
            new ECBlocks(30, new ECB(49, 24),
                new ECB(10, 25)),
            new ECBlocks(30, new ECB(24, 15),
                new ECB(46, 16))),
        new Version(38, new int[]{6, 32, 58, 84, 110, 136, 162},
            new ECBlocks(30, new ECB(4, 122),
                new ECB(18, 123)),
            new ECBlocks(28, new ECB(13, 46),
                new ECB(32, 47)),
            new ECBlocks(30, new ECB(48, 24),
                new ECB(14, 25)),
            new ECBlocks(30, new ECB(42, 15),
                new ECB(32, 16))),
        new Version(39, new int[]{6, 26, 54, 82, 110, 138, 166},
            new ECBlocks(30, new ECB(20, 117),
                new ECB(4, 118)),
            new ECBlocks(28, new ECB(40, 47),
                new ECB(7, 48)),
            new ECBlocks(30, new ECB(43, 24),
                new ECB(22, 25)),
            new ECBlocks(30, new ECB(10, 15),
                new ECB(67, 16))),
        new Version(40, new int[]{6, 30, 58, 86, 114, 142, 170},
            new ECBlocks(30, new ECB(19, 118),
                new ECB(6, 119)),
            new ECBlocks(28, new ECB(18, 47),
                new ECB(31, 48)),
            new ECBlocks(30, new ECB(34, 24),
                new ECB(34, 25)),
            new ECBlocks(30, new ECB(20, 15),
                new ECB(61, 16)))
    };
  }

}
```

- **Decoder.java**

```
public final class Decoder {

  private final ReedSolomonDecoder rsDecoder;
```

```
  public Decoder() {
    rsDecoder = new ReedSolomonDecoder(GenericGF.QR_CODE_FIELD_256);
  }

  public DecoderResult decode(boolean[][] image) throws ChecksumException,
FormatException {
    return decode(image, null);
  }

  /**
   * <p>Convenience method that can decode a QR Code represented as a 2D array of
booleans.
   * "true" is taken to mean a black module.</p>
   *
   * @param image booleans representing white/black QR Code modules
   * @return text and bytes encoded within the QR Code
   * @throws FormatException if the QR Code cannot be decoded
   * @throws ChecksumException if error correction fails
   */
  public DecoderResult decode(boolean[][] image, Map<DecodeHintType,?> hints)
      throws ChecksumException, FormatException {
    int dimension = image.length;
    BitMatrix bits = new BitMatrix(dimension);
    for (int i = 0; i < dimension; i++) {
      for (int j = 0; j < dimension; j++) {
        if (image[i][j]) {
          bits.set(j, i);
        }
      }
    }
    return decode(bits, hints);
  }

  public DecoderResult decode(BitMatrix bits) throws ChecksumException,
FormatException {
    return decode(bits, null);
  }

  /**
   * <p>Decodes a QR Code represented as a {@link BitMatrix}. A 1 or "true" is taken
to mean a black module.</p>
   *
   * @param bits booleans representing white/black QR Code modules
   * @return text and bytes encoded within the QR Code
   * @throws FormatException if the QR Code cannot be decoded
   * @throws ChecksumException if error correction fails
   */
  public DecoderResult decode(BitMatrix bits, Map<DecodeHintType,?> hints)
      throws FormatException, ChecksumException {

    // Construct a parser and read version, error-correction level
    BitMatrixParser parser = new BitMatrixParser(bits);
    Version version = parser.readVersion();
    ErrorCorrectionLevel ecLevel =
parser.readFormatInformation().getErrorCorrectionLevel();

    // Read codewords
    byte[] codewords = parser.readCodewords();
    // Separate into data blocks
    DataBlock[] dataBlocks = DataBlock.getDataBlocks(codewords, version, ecLevel);

    // Count total number of data bytes
```

```
    int totalBytes = 0;
    for (DataBlock dataBlock : dataBlocks) {
      totalBytes += dataBlock.getNumDataCodewords();
    }
    byte[] resultBytes = new byte[totalBytes];
    int resultOffset = 0;

    // Error-correct and copy data blocks together into a stream of bytes
    for (DataBlock dataBlock : dataBlocks) {
      byte[] codewordBytes = dataBlock.getCodewords();
      int numDataCodewords = dataBlock.getNumDataCodewords();
      correctErrors(codewordBytes, numDataCodewords);
      for (int i = 0; i < numDataCodewords; i++) {
        resultBytes[resultOffset++] = codewordBytes[i];
      }
    }

    // Decode the contents of that stream of bytes
    return DecodedBitStreamParser.decode(resultBytes, version, ecLevel, hints);
  }

  /**
   * <p>Given data and error-correction codewords received, possibly corrupted by
errors, attempts to
   * correct the errors in-place using Reed-Solomon error correction.</p>
   *
   * @param codewordBytes data and error correction codewords
   * @param numDataCodewords number of codewords that are data bytes
   * @throws ChecksumException if error correction fails
   */
  private void correctErrors(byte[] codewordBytes, int numDataCodewords) throws
ChecksumException {
    int numCodewords = codewordBytes.length;
    // First read into an array of ints
    int[] codewordsInts = new int[numCodewords];
    for (int i = 0; i < numCodewords; i++) {
      codewordsInts[i] = codewordBytes[i] & 0xFF;
    }
    int numECCodewords = codewordBytes.length - numDataCodewords;
    try {
      rsDecoder.decode(codewordsInts, numECCodewords);
    } catch (ReedSolomonException rse) {
      throw ChecksumException.getChecksumInstance();
    }
    // Copy back into array of bytes -- only need to worry about the bytes that were
data
    // We don't care about errors in the error-correction codewords
    for (int i = 0; i < numDataCodewords; i++) {
      codewordBytes[i] = (byte) codewordsInts[i];
    }
  }

}
```