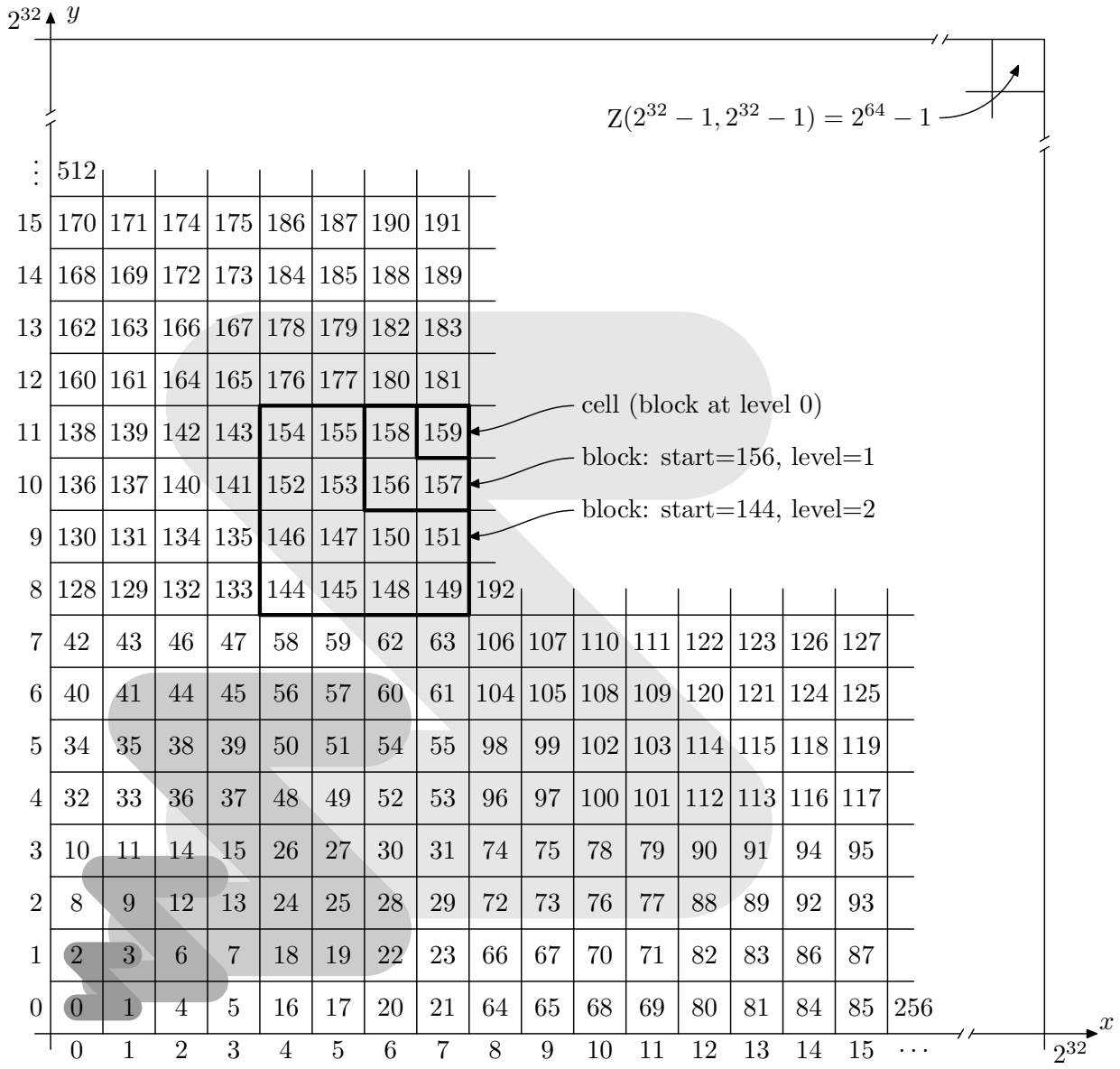
Z Curve (Morton) Ordering

Imagine a two-dimensional grid of cells, indexed by integer x and y coordinates. The cells are numbered in Z curve order, starting at the lower left corner with $\mathrm{Z}(x, y) = \mathrm{Z}(0, 0) = 0$ (see figure). The virtue of the Z curve is good locality: it completes one quadrant before entering the next. Moreover, it is simple to compute Z values from x, y coordinates and vice versa by bit interleaving.

The grid's cells can be partitioned into blocks of 2 by 2 cells, and blocks can be recursively partitioned into larger blocks of 2 by 2 smaller blocks. Cells are said to be blocks at level 0, blocks of 4 cells are at level 1, blocks of 16 cells at level 2, and so on up to the one block at level 32 that covers the entire grid. Blocks are identified by their level n and their lower-left cell's Z value (see figure).



⋮	512															
15	170	171	174	175	186	187	190	191								
14	168	169	172	173	184	185	188	189								
13	162	163	166	167	178	179	182	183								
12	160	161	164	165	176	177	180	181								
11	138	139	142	143	154	155	158	159								
10	136	137	140	141	152	153	156	157								
9	130	131	134	135	146	147	150	151								
8	128	129	132	133	144	145	148	149	192							
7	42	43	46	47	58	59	62	63	106	107	110	111	122	123	126	127
6	40	41	44	45	56	57	60	61	104	105	108	109	120	121	124	125
5	34	35	38	39	50	51	54	55	98	99	102	103	114	115	118	119
4	32	33	36	37	48	49	52	53	96	97	100	101	112	113	116	117
3	10	11	14	15	26	27	30	31	74	75	78	79	90	91	94	95
2	8	9	12	13	24	25	28	29	72	73	76	77	88	89	92	93
1	2	3	6	7	18	19	22	23	66	67	70	71	82	83	86	87
0	0	1	4	5	16	17	20	21	64	65	68	69	80	81	84	85	256
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	⋯

The `ZCurve` utility class provides operations related to the Z curve, using unsigned 32-bit integers for the x and y coordinates, and unsigned 64-bit integers for the interleaved Z values.

`Encode`(x, y) encodes the x and y coordinates into a single Z value z by bit interleaving in about 30 bit-level operations (shift, and, or, xor).

`DecodeX`(z) and `DecodeY`(z) retrieve the x and y coordinates from the Z value z, each in about 15 bit-level operations.

`IsInsideBox`(z, lo, hi) tests if z is within the box defined by lo and hi; instead of decoding the Z values into x and y components, it masks off every other bit and compares the resulting 64-bit values.

`NextInsideBox`(z, lo, hi) finds the least $z' > z$ that is inside the box defined by lo and hi using the algorithm described by Tropf and Herzog ("Multidimensional Range Search in Dynamically Balanced Trees," *Angewandte Informatik*, 2/1981). For example, `NextInsideBox`$(28, 3, 49) = 33$.

`LargestLevel`(z) is the largest block level starting at the given z value; it is $\lfloor \mathrm{NTZ}(z)/2 \rfloor$ where NTZ is the number of trailing zero bits in z. Overloads find the largest block level starting at a given z value and not extending beyond a given maximum Z value, and consider only a given subset of all levels $0 \ldots 32$. For example, `LargestLevel`$(0) = 32$, but `LargestLevel`$(0, 49) = 2$ and `LargestLevel`$(0, 49, [1, 3]) = 1$.

`GetQueryBlocks`$(lo, hi, levels)$ yields the shortest ordered sequence of blocks covering the whole box defined by min and max Z values lo and hi. The idea is to use larger blocks inside the box and smaller blocks along the boundary of the box. If *levels* are provided, only blocks at those levels are used; in this case, the blocks may extend beyond the box.