

1. Intro. Find a comma-free block code of length n , having one code in each cyclic equivalence class, if one exists.

Codewords are represented as hexadecimal numbers.

```
#define maxn 25 /* must be at most 32, to keep the variable names small */
#include <stdio.h>
#include <stdlib.h>
int n; /* command-line parameter */
char a[maxn + 1];
main(int argc, char *argv[])
{
    register int i, j, k;
    register unsigned int x, y, z;
    register unsigned long long m, acc, xy;
    < Process the command line 2 >;
    < Generate the positive clauses 3 >;
    < Generate the negative clauses 5 >;
}
```

2. < Process the command line 2 > \equiv

```
if (argc  $\neq$  2  $\vee$  sscanf(argv[1], "%d", &n)  $\neq$  1) {
    fprintf(stderr, "Usage: %s\n", argv[0]);
    exit(-1);
}
if (n < 2  $\vee$  n > maxn) {
    fprintf(stderr, "n should be between 2 and %d, not %d!\n", maxn, n);
    exit(-2);
}
printf("~sat-commafree%d\n", n);
```

This code is used in section 1.

3. Here I use Algorithm 7.2.1.1F to find the prime binary strings.

< Generate the positive clauses 3 > \equiv

```
f1: a[0] = -1, j = 1;
f2: if (j  $\equiv$  n) < Visit the prime string  $a_1 \dots a_n$  4 >;
f3: for (j = n; a[j]  $\equiv$  1; j--) ;
f4: if (j) {
    a[j] = 1;
    f5: for (k = j + 1; k  $\leq$  n; k++) a[k] = a[k - j];
    goto f2;
}
```

This code is used in section 1.

4. < Visit the prime string $a_1 \dots a_n$ 4 > \equiv

```
{
    for (i = 0; i < n; i++) {
        for (x = 0, k = 0; k < n; k++) x = (x  $\ll$  1) + a[1 + ((i + k) % n)];
        printf("%x", x);
    }
    printf("\n");
}
```

This code is used in section 3.

```

5.  ⟨Generate the negative clauses 5⟩ ≡
    m = (1LL ≪ n) - 1;
    for (x = 0; x < (1 ≪ n); x++) {
        ⟨If x is cyclic, continue 6⟩;
        for (y = 0; y < (1 ≪ n); y++) {
            ⟨If y is cyclic, continue 7⟩;
            ⟨Generate the clauses for x followed by y 9⟩;
        }
    }

```

This code is used in section 1.

```

6.  ⟨If x is cyclic, continue 6⟩ ≡
    acc = (((unsigned long long) x) ≪ n) + x;
    for (k = 1; k < n; k++)
        if (((acc ≫ k) & m) ≡ x) break;
    if (k < n) continue;

```

This code is used in section 5.

```

7.  ⟨If y is cyclic, continue 7⟩ ≡
    acc = (((unsigned long long) y) ≪ n) + y;
    for (k = 1; k < n; k++)
        if (((acc ≫ k) & m) ≡ y) break;
    if (k < n) continue;

```

This code is used in section 5.

```

8.  ⟨If z is cyclic, continue 8⟩ ≡
    acc = (((unsigned long long) z) ≪ n) + z;
    for (k = 1; k < n; k++)
        if (((acc ≫ k) & m) ≡ z) break;
    if (k < n) continue;

```

This code is used in section 9.

```

9.  ⟨Generate the clauses for x followed by y 9⟩ ≡
    xy = (((unsigned long long) x) ≪ n) + y;
    for (j = 1; j < n; j++) {
        z = (xy ≫ j) & m;
        ⟨If z is cyclic, continue 8⟩;
        printf("~%x~%x~%x\n", x, y, z);
    }

```

This code is used in section 5.

10. Index.

a: [1](#).
acc: [1](#), [6](#), [7](#), [8](#).
argc: [1](#), [2](#).
argv: [1](#), [2](#).
exit: [2](#).
fprintf: [2](#).
f1: [3](#).
f2: [3](#).
f3: [3](#).
f4: [3](#).
f5: [3](#).
i: [1](#).
j: [1](#).
k: [1](#).
m: [1](#).
main: [1](#).
maxn: [1](#), [2](#).
n: [1](#).
printf: [2](#), [4](#), [9](#).
sscanf: [2](#).
stderr: [2](#).
x: [1](#).
xy: [1](#), [9](#).
y: [1](#).
z: [1](#).

- ⟨ Generate the clauses for x followed by y 9 ⟩ Used in section 5.
- ⟨ Generate the negative clauses 5 ⟩ Used in section 1.
- ⟨ Generate the positive clauses 3 ⟩ Used in section 1.
- ⟨ If x is cyclic, **continue** 6 ⟩ Used in section 5.
- ⟨ If y is cyclic, **continue** 7 ⟩ Used in section 5.
- ⟨ If z is cyclic, **continue** 8 ⟩ Used in section 9.
- ⟨ Process the command line 2 ⟩ Used in section 1.
- ⟨ Visit the prime string $a_1 \dots a_n$ 4 ⟩ Used in section 3.

SAT-COMMAFREE

	Section	Page
Intro	1	1
Index	10	3