

November 24, 2020 at 13:23

1. Intro. This program contains two implementations of the Koda–Ruskey algorithm for generating all ideals of a given forest poset [*Journal of Algorithms* **15** (1993), 324–340]. The common goal of both implementations is, in essence, to generate all binary strings $b_0 \dots b_{n-1}$ in which certain bits are required to be less than or equal to specified bits that lie to their right. (For some values of j there is a value of $k > j$ such that we don’t allow b_k to be 0 when $b_j = 1$.) Moreover, each binary string should differ from its predecessor in exactly one bit position; the algorithm therefore defines a generalized reflected Gray code.

The given forest is represented by n pairs of nested parentheses. For example, $()()()()()$ represents five independent bits, while $(((((()))))$ represents five bits with $b_0 \leq b_1 \leq b_2 \leq b_3 \leq b_4$. A more interesting example, $((())(())())$, represents six bits subject to the conditions $b_0 \leq b_1$, $b_1 \leq b_5$, $b_2 \leq b_4$, $b_3 \leq b_4$, $b_4 \leq b_5$. Each pair of parentheses corresponds to a bit that must not exceed the bit of its enclosing pair, if any, and the pairs are ordered by the appearances of their right parentheses.

The first implementation uses n coroutines, which call each other in a hierarchical fashion. The second uses multilinked data structures in a loopless way, so that each generation step performs a bounded number of operations to obtain the next element. I couldn’t resist writing this program, because both implementations turn out to be quite interesting and instructive.

Indeed, I think it’s a worthwhile challenge for people who study the science of computer programming to verify that these two implementations both define the same sequence of bitstrings. Even more challenging would be to derive the second implementation “automatically” from the first.

```
#include <stdio.h>
<Type definitions 4>
<Global variables 3>
int main(int argc, char *argv[])
{
    register int j, k, l;
    <Process the command line, parsing the given forest 2>;
    printf("Bitstrings generated from \"%s\":\n", argv[1]);
    <Generate the strings with a coroutine implementation 9>;
    printf("\nTrying again, looplessly:\n");
    <Generate the strings with a loopless implementation 15>;
    return 0;
}
```

2. In this step we parse the forest into an array of “scopes”: $scope[j]$ is the index of the smallest descendant of node j , including node j itself.

```
#define abort(m,i)
    { fprintf(stderr,m,argv[i]); return -1; }
#define stacksize 100 /* max levels in the forest */
#define forestsize 100 /* max nodes in the forest */
⟨ Process the command line, parsing the given forest 2 ⟩ ≡
    if (argc ≠ 2 ∨ argv[1][0] ≠ '(') abort("Usage: %s \"nestedparens\\\"\\n\",0);
    for (j = k = l = 0; argv[1][k]; k++)
        if (argv[1][k] ≡ '(') {
            stack[l++] = j;
            if (l ≡ stacksize) abort("Stack_overflow---\\\"%s\\\" is too deep for me!\\n\",1);
        } else if (argv[1][k] ≡ ')') {
            if (--l < 0) abort("Extra_right_parenthesis_in\\\"%s\\\"!\\n\",1);
            scope[j++] = stack[l];
            if (j ≡ forestsize) abort("Memory_overflow---\\\"%s\\\" is too big!\\n\",1);
        } else abort("The_forest_spec\\\"%s\\\" should contain only parentheses!\\n\",1);
    if (l) abort("Missing_right_parenthesis_in\\\"%s\\\"!\\n\",1);
    nn = j;
```

This code is used in section 1.

3. ⟨ Global variables 3 ⟩ ≡

```
int stack[stacksize]; /* nodes preceding each open leftparen, while parsing */
int scope[forestsize]; /* table that exhibits each rightparen's influence */
int nn; /* the actual number of nodes in the forest */
```

See also sections 11 and 17.

This code is used in section 1.

4. The coroutine implementation. Our first implementation uses a system of n cooperating programs, each of which represents a node in the forest. For convenience we will call the associated record a “cnode.” If p points to a cnode, $p\text{-child}$ points to the cnode representing its rightmost child, and $p\text{-sib}$ points to the cnode representing its nearest sibling on the left, in the given forest.

Each cnode corresponds to a coroutine whose job is to generate all the ideals of the subforest it represents. Whenever the coroutine is invoked, it either changes one of the bits in its scope and returns *true*, or it changes nothing and returns *false*. Initially all the bits are 0; when it first returns *false*, it will have generated all legitimate bit patterns, ending with some nonzero pattern. Subsequently it will generate the patterns again in reverse order, ending with all 0s, after which it will return *false* a second time. Invoking it again and again will repeat the same process, going forwards and backwards, ad infinitum.

Each coroutine has the same basic structure, which can be described as follows in an ad hoc extension of C language:

```
coroutine p()
{
    while (1) {
        p-bit = 1; return true;
        while (p-child()) return true;
        return p-sib();
        while (p-child()) return true;
        p-bit = 0; return true;
        return p-sib();
    }
}
```

If either $p\text{-child}$ or $p\text{-sib}$ is Λ , the corresponding coroutine $\Lambda()$ is assumed to simply return *false*.

Suppose $p\text{-child}()$ first returns *false* after it has been called r times; thus $p\text{-child}()$ generates r different patterns, including the initial pattern of all 0s. Similarly, suppose that $p\text{-sib}()$ generates l different patterns before first returning *false*. Then the coroutine $p()$ itself will generate $l(r+1)$ patterns in between the times when it returns *false*. The final bit pattern for p will be the final bit pattern for $p\text{-sib}$, together with either $p\text{-bit} = 1$ and the final bit pattern of $p\text{-child}$ (if l is odd) or with $p\text{-bit} = 0$ and all 0s in $p\text{-child}$ (if l is even).

(Type definitions 4) \equiv

```
typedef enum {
    false, true
} boolean;

typedef struct cnode_struct {
    char bit; /* either 0 or 1; always 1 when a child's bit is set */
    char state; /* the current place in this cnode's coroutine */
    struct cnode_struct *child; /* rightmost child in the given forest */
    struct cnode_struct *sib; /* nearest left sibling in the given forest */
    struct cnode_struct *caller; /* which coroutine invoked this one */
} cnode;
```

See also section 14.

This code is used in section 1.

5. When coroutine p calls coroutine q , it sets p -state to an appropriate number and also sets q -caller = p . Then control passes to q at the place determined by q -state.

When coroutine q wants to return a boolean value, it sets *coresult* to this value; then it passes control to $p = q$ -caller at the place determined by p -state.

This program simulates coroutine linkage with a big switch statement. Actually the notion of “passing control” really means that we simply assign a value to the variable *cur_cnode*.

The value of q -caller for every cnode q is completely determined by the structure of the given forest, so we could set it once and for all during the initialization instead of setting it dynamically as done here. But what the heck.

```
#define cocall(q, s)
    { cur_cnode-state = s;
      if (q) q-caller = cur_cnode, cur_cnode = q;
      else coresult = false;
      goto cogo; }
#define bitchange(b, s)
    { cur_cnode-bit = b, coresult = true; coreturn(s); }
#define coreturn(s)
    { cur_cnode-state = s, cur_cnode = cur_cnode-caller;
      goto cogo; }

⟨ Repeatedly switch to the proper part of the current coroutine 5 ⟩ ≡
cogo: switch (cur_cnode-state) {
    ⟨ Cases for coroutine states 6 ⟩;
    default: abort("%s: Unknown state code (this can't happen)!\n", 0);
}
```

This code is used in section 9.

6. In its initial state 0, a coroutine turns its bit on, returns *true*, and enters state 1.

```
⟨ Cases for coroutine states 6 ⟩ ≡
case 0: bitchange(1, 1);
```

See also sections 7, 8, and 12.

This code is used in section 5.

7. The purpose of state 1 is to run through all bit patterns of the current node’s children, starting with all 0s and ending when they reach their final pattern. At that point we invoke the current node’s nearest left sibling and enter state 3. An intermediate state 2 is defined for the purpose of examining the result after calling the child coroutine.

The purpose of state 3 is simply to return to whoever called us, passing along the information in *coresult*, which tells whether any of our left siblings has changed one of its bits. Then we will continue in state 4.

```
⟨ Cases for coroutine states 6 ⟩ +=
case 1: cocall(cur_cnode-child, 2);
case 2: if (coresult) coreturn(1);
        cocall(cur_cnode-sib, 3);
case 3: coreturn(4);
```

8. State 4 is rather like state 1, except that the child coroutine is now running through its bit patterns in reverse order. Finally it reduces them all to 0s, and returns *false* the next time we attempt to invoke it. At that point we reset the current bit, return *true*, and enter state 6.

State 6 invokes the sibling coroutine, leading to state 7. And state 7 is like state 3, but it takes us back to state 0 instead of state 4.

```

⟨ Cases for coroutine states 6 ⟩ +≡
case 4: cocall(cur_cnode-child, 5);
case 5: if (coresult) coreturn(4);
        bitchange(0, 6);
case 6: cocall(cur_cnode-sib, 7);
case 7: coreturn(0);

```

9. Hey, the implementation is done already, except that we have to get it started and write the code that controls it at the outermost level.

```

⟨ Generate the strings with a coroutine implementation 9 ⟩ ≡
{
    register cnode *cur_cnode;
    ⟨ Initialize the cnode structure 10 ⟩;
    ⟨ Repeatedly switch to the proper part of the current coroutine 5 ⟩;
}

```

This code is used in section 1.

10. We allocate a special cnode to represent the external world outside of the given forest.

```

#define root_cnode cnode_table[nn].child
⟨ Initialize the cnode structure 10 ⟩ ≡
    scope[nn] = 0;
    for (k = 0; k ≤ nn; k++)
        if (scope[k] < k) {
            cnode_table[k].child = cnode_table + k - 1;
            for (j = k - 1; scope[j] > scope[k]; j = scope[j] - 1) cnode_table[j].sib = cnode_table + scope[j] - 1;
        }
    cur_cnode = cnode_table + nn;
    goto upward_step;

```

This code is used in section 9.

```

11. ⟨ Global variables 3 ⟩ +≡
    cnode cnode_table[forestsize + 1];    /* the cnodes */
    boolean coresult;    /* value returned by a coroutine */

```

12. States 8 and greater are reserved for the external (outermost) level, which simply invokes the coroutine for the entire forest and prints out the results, until the bit patterns have been generated in both the forward and reverse directions.

⟨ Cases for coroutine states 6 ⟩ +≡

```

case 8: if (coresult) {
    upward_step: ⟨Print out all the current cnode bits 13⟩;
    cocall(root_cnode, 8);
}
printf("...and now we generate them in reverse:\n");
goto downward_step;
case 9: if (coresult) {
    downward_step: ⟨Print out all the current cnode bits 13⟩;
    cocall(root_cnode, 9);
}
break;

```

13. ⟨Print out all the current cnode bits 13⟩ ≡

```

for (k = 0; k < nn; k++) putchar('0' + cnode_table[k].bit);
putchar('\\n');

```

This code is used in section 12.

14. The loopless implementation. Our coroutine implementation solves the generation problem in a nice and natural fashion, but it can be inefficient if the given forest has numerous nodes of degree one. For example, a one-tree forest like $((\dots())\dots)$ with n pairs of parentheses will need approximately $\binom{n}{2}$ coroutine invocations to generate $n + 1$ bitstrings.

Our second implementation reduces the work in such cases to $O(n)$; in fact, it needs only a bounded number of operations to generate each bitstring after the first. It does, however, need a slightly more complex data structure with four link fields.

The basic idea is to work with a dynamically varying list of nodes called the current *fringe* of the forest. The fringe consists of all node whose bit is 1, together with their children. We maintain it as a doubly linked list, so that $p\text{-left}$ and $p\text{-right}$ are the neighbors of p on the left and right. A special node *head* is provided to make the list circular; thus $head\text{-right}$ and $head\text{-left}$ are the leftmost and rightmost fringe nodes.

A fringe node is said to be either *active* or *passive*. Every node is active when it joins the fringe, but it becomes passive for at least a short time when its bit changes value; at such times the node is essentially shifting direction between going forward or backward, as in the coroutine implementation. (A passive node corresponds roughly to a coroutine that is asking its siblings to make the next move.) We save time jumping across such call-chains by using a special link field called the *focus*: If p is a passive fringe node whose righthand neighbor $p\text{-right}$ is active, $p\text{-focus}$ is the rightmost active node to the left of p in the fringe; otherwise $p\text{-focus} = p$. (The special *head* node is always considered to be active, for purposes of this definition, but it is not strictly speaking a member of the fringe.)

The loopless implementation works with records called lnodes, just as the coroutine implementation worked with cnodes. Besides the dynamic *bit* and *left* and *right* and *focus* fields already mentioned, each lnode also has a static field called *lchild*, representing its leftmost child. (There is no need for an *rchild* field, since $p\text{-rchild} = p - 1$ when $p\text{-lchild} \neq \Lambda$.)

If p is not in the fringe, $p\text{-focus}$ should equal p . Also, $p\text{-left}$ and $p\text{-right}$ are assumed to equal the nearest siblings of p to the left and right, respectively, if such siblings exist; otherwise $p\text{-left}$ and/or $p\text{-right}$ are undefined.

⟨ Type definitions 4 ⟩ $\vdash \equiv$

```
typedef struct lnode_struct {
    char bit;          /* either 0 or 1; always 1 when a child's bit is set */
    struct lnode_struct *left, *right; /* neighbors in the forest and/or fringe */
    struct lnode_struct *lchild;      /* leftmost child */
    struct lnode_struct *focus;      /* red-tape cutter for efficiency */
} lnode;
```

15. Here now is the basic outline of the loopless implementation:

```

⟨Generate the strings with a loopless implementation 15⟩ ≡
{
  register lnode *p, *q, *r;
  ⟨Initialize the lnode structure, putting all roots into the fringe 16⟩;
  while (1) {
    ⟨Print out all the current lnode bits 22⟩;
    ⟨Set p to the rightmost active node of the fringe, and activate everything to its right 18⟩;
    if (p ≠ head) {
      if (p→bit ≡ 0) {
        p→bit = 1; /* moving forward */
        ⟨Insert the children of p after p in the fringe 19⟩;
      } else {
        p→bit = 0; /* moving backward */
        ⟨Delete the children of p from the fringe 20⟩;
      }
    } else if (been_there_and_done_that) break;
    else {
      printf("...and now we generate them in reverse:\n");
      been_there_and_done_that = true; continue;
    }
    ⟨Make node p passive 21⟩;
  }
}

```

This code is used in section 1.

16. Initialization of the lnodes is similar to initialization of the cnodes, but more links need to be set up.

```

#define head (lnode_table + nn)
⟨Initialize the lnode structure, putting all roots into the fringe 16⟩ ≡
for (k = 0; k ≤ nn; k++) {
  lnode_table[k].focus = lnode_table + k;
  if (scope[k] < k) {
    for (j = k - 1; scope[j] > scope[k]; j = scope[j] - 1) {
      lnode_table[j].left = lnode_table + scope[j] - 1;
      lnode_table[scope[j] - 1].right = lnode_table + j;
    }
    lnode_table[k].lchild = lnode_table + j;
  }
}
head→left = head - 1, (head - 1)→right = head;
head→right = head→lchild, head→lchild→left = head;

```

This code is used in section 15.

17. ⟨Global variables 3⟩ +≡
 lnode lnode_table[forestsize + 1]; /* the lnodes */
 boolean been_there_and_done_that;

18. $\langle \text{Set } p \text{ to the rightmost active node of the fringe, and activate everything to its right } 18 \rangle \equiv$
 $q = \text{head-left};$
 $p = q\text{-focus};$
 $q\text{-focus} = q;$

This code is used in section 15.

19. $\langle \text{Insert the children of } p \text{ after } p \text{ in the fringe } 19 \rangle \equiv$
 $\text{if } (p\text{-lchild}) \{$
 $q = p\text{-right};$
 $q\text{-left} = p - 1, (p - 1)\text{-right} = q;$
 $p\text{-right} = p\text{-lchild}, p\text{-lchild-left} = p;$
 $\}$

This code is used in section 15.

20. $\langle \text{Delete the children of } p \text{ from the fringe } 20 \rangle \equiv$
 $\text{if } (p\text{-lchild}) \{$
 $q = (p - 1)\text{-right};$
 $p\text{-right} = q, q\text{-left} = p;$
 $\}$

This code is used in section 15.

21. At this point we know that $p\text{-right}$ is active.

$\langle \text{Make node } p \text{ passive } 21 \rangle \equiv$
 $p\text{-focus} = p\text{-left-focus};$
 $p\text{-left-focus} = p\text{-left};$

This code is used in section 15.

22. $\langle \text{Print out all the current lnode bits } 22 \rangle \equiv$
 $\text{for } (k = 0; k < nn; k++) \text{ putchar}('0' + \text{lnode_table}[k].\text{bit});$
 $\text{putchar}('\n');$

This code is used in section 15.

23. I used the following code when debugging.

#define $\text{rel}(f) \ (\text{lnode_table}[k].f ? \text{lnode_table}[k].f - \text{lnode_table} : -1)$
 $\langle \text{Print out the whole lnode structure } 23 \rangle \equiv$
 $\text{for } (k = 0; k \leq nn; k++) \{$
 $\text{printf}(\text{"lnode_}\%d\text{:_bit=}\%d\text{,_"} , k, \text{lnode_table}[k].\text{bit});$
 $\text{printf}(\text{"focus=}\%d\text{,_left=}\%d\text{,_right=}\%d\text{,_lchild=}\%d\text{\n"} , \text{rel}(\text{focus}), \text{rel}(\text{left}), \text{rel}(\text{right}), \text{rel}(\text{lchild}));$
 $\}$

24. Index.

abort: [2](#), [5](#).
argc: [1](#), [2](#).
argv: [1](#), [2](#).
been_there_and_done_that: [15](#), [17](#).
bit: [4](#), [5](#), [13](#), [14](#), [15](#), [22](#), [23](#).
bitchange: [5](#), [6](#), [8](#).
boolean: [4](#), [11](#), [17](#).
caller: [4](#), [5](#).
child: [4](#), [7](#), [8](#), [10](#).
cnode: [4](#), [9](#), [11](#).
cnode_struct: [4](#).
cnode_table: [10](#), [11](#), [13](#).
cocall: [5](#), [7](#), [8](#), [12](#).
cogo: [5](#).
coresult: [5](#), [7](#), [8](#), [11](#), [12](#).
coreturn: [5](#), [7](#), [8](#).
cur_cnode: [5](#), [7](#), [8](#), [9](#), [10](#).
downward_step: [12](#).
false: [4](#), [5](#), [8](#).
focus: [14](#), [16](#), [18](#), [21](#), [23](#).
forestsize: [2](#), [3](#), [11](#), [17](#).
fprintf: [2](#).
head: [14](#), [15](#), [16](#), [18](#).
j: [1](#).
k: [1](#).
l: [1](#).
lchild: [14](#), [16](#), [19](#), [20](#), [23](#).
left: [14](#), [16](#), [18](#), [19](#), [20](#), [21](#), [23](#).
lnode: [14](#), [15](#), [17](#).
lnode_struct: [14](#).
lnode_table: [16](#), [17](#), [22](#), [23](#).
main: [1](#).
nn: [2](#), [3](#), [10](#), [13](#), [16](#), [22](#), [23](#).
p: [15](#).
printf: [1](#), [12](#), [15](#), [23](#).
putchar: [13](#), [22](#).
q: [15](#).
r: [15](#).
rchild: [14](#).
rel: [23](#).
right: [14](#), [16](#), [19](#), [20](#), [21](#), [23](#).
root_cnode: [10](#), [12](#).
scope: [2](#), [3](#), [10](#), [16](#).
sib: [4](#), [7](#), [8](#), [10](#).
stack: [2](#), [3](#).
stacksize: [2](#), [3](#).
state: [4](#), [5](#).
stderr: [2](#).
true: [4](#), [5](#), [6](#), [8](#), [15](#).
upward_step: [10](#), [12](#).

- ⟨ Cases for coroutine states 6, 7, 8, 12 ⟩ Used in section 5.
- ⟨ Delete the children of p from the fringe 20 ⟩ Used in section 15.
- ⟨ Generate the strings with a coroutine implementation 9 ⟩ Used in section 1.
- ⟨ Generate the strings with a loopless implementation 15 ⟩ Used in section 1.
- ⟨ Global variables 3, 11, 17 ⟩ Used in section 1.
- ⟨ Initialize the cnode structure 10 ⟩ Used in section 9.
- ⟨ Initialize the lnode structure, putting all roots into the fringe 16 ⟩ Used in section 15.
- ⟨ Insert the children of p after p in the fringe 19 ⟩ Used in section 15.
- ⟨ Make node p passive 21 ⟩ Used in section 15.
- ⟨ Print out all the current cnode bits 13 ⟩ Used in section 12.
- ⟨ Print out all the current lnode bits 22 ⟩ Used in section 15.
- ⟨ Print out the whole lnode structure 23 ⟩
- ⟨ Process the command line, parsing the given forest 2 ⟩ Used in section 1.
- ⟨ Repeatedly switch to the proper part of the current coroutine 5 ⟩ Used in section 9.
- ⟨ Set p to the rightmost active node of the fringe, and activate everything to its right 18 ⟩ Used in section 15.
- ⟨ Type definitions 4, 14 ⟩ Used in section 1.

KODA-RUSKEY

	1	
	2	
	3	
	4	
	5	
	6	
	7	
	8	
	9	
	10	
	11	
	12	
	13	
	14	
	15	
	16	
	17	
	18	
	19	
	20	
	21	
	22	
	23	
	24	
	25	
	26	
	27	
	28	
	29	
	30	
	31	
	32	
	33	
	34	
	35	
	36	
	37	
	38	
	39	
	40	
	41	
	42	
	43	
	44	
	45	
	46	
	47	
	48	
	49	
	50	
	51	
	52	
	53	
	54	
	55	
	56	
	57	
	58	
	59	
	60	
	61	
	62	
	63	
	64	
	65	
	66	
	67	
	68	
	69	
	70	
	71	
	72	
	73	
	74	
	75	
	76	
	77	
	78	
	79	
	80	
	81	
	82	
	83	
	84	
	85	
	86	
	87	
	88	
	89	
	90	
	91	
	92	
	93	
	94	
	95	
	96	
	97	
	98	
	99	
	100	
	101	
	102	
	103	
	104	
	105	
	106	
	107	
	108	
	109	
	110	
	111	
	112	
	113	
	114	
	115	
	116	
	117	
	118	
	119	
	120	
	121	
	122	
	123	
	124	
	125	
	126	
	127	
	128	
	129	
	130	
	131	
	132	
	133	
	134	
	135	
	136	
	137	
	138	
	139	
	140	
	141	
	142	
	143	
	144	
	145	
	146	
	147	
	148	
	149	
	150	
	151	
	152	
	153	
	154	
	155	
	156	
	157	
	158	
	159	
	160	
	161	
	162	
	163	
	164	
	165	
	166	
	167	
	168	
	169	
	170	
	171	
	172	
	173	
	174	
	175	
	176	
	177	
	178	
	179	
	180	
	181	
	182	
	183	
	184	
	185	
	186	
	187	
	188	
	189	
	190	
	191	
	192	
	193	
	194	
	195	
	196	
	197	
	198	
	199	
	200	
	201	
	202	
	203	
	204	
	205	
	206	
	207	
	208	
	209	
	210	
	211	
	212	
	213	
	214	
	215	
	216	
	217	
	218	
	219	
	220	
	221	
	222	
	223	
	224	
	225	
	226	
	227	
	228	
	229	
	230	
	231	
	232	
	233	
	234	
	235	
	236	
	237	
	238	
	239	
	240	
	241	
	242	
	243	
	244	
	245	
	246	
	247	
	248	
	249	
	250	
	251	
	252	
	253	
	254	
	255	
	256	
	257	
	258	
	259	
	260	
	261	
	262	
	263	
	264	
	265	
	266	
	267	
	268	
	269	
	270	
	271	
	272	
	273	
	274	
	275	
	276	
	277	
	278	
	279	
	280	
	281	
	282	
	283	
	284	
	285	
	286	
	287	
	288	
	289	
	290	
	291	
	292	
	293	
	294	
	295	
	296	
	297	
	298	
	299	
	300	
	301	
	302	
	303	
	304	
	305	
	306	
	307	
	308	
	309	
	310	
	311	
	312	
	313	
	314	
	315	
	316	
	317	
	318	
	319	
	320	
	321	
	322	
	323	
	324	
	325	
	326	
	327	
	328	
	329	
	330	
	331	
	332	
	333	
	334	
	335	
	336	
	337	
	338	
	339	
	340	
	341	
	342	
	343	
	344	
	345	
	346	
	347	
	348	
	349	
	350	
	351	
	352	
	353	
	354	
	355	
	356	
	357	
	358	
	359	
	360	
	361	
	362	
	363	
	364	
	365	
	366	
	367	
	368	
	369	
	370	
	371	
	372	
	373	
	374	
	375	
	376	
	377	
	378	
	379	
	380	
	381	
	382	
	383	
	384	
	385	
	386	
	387	
	388	
	389	
	390	
	391	
	392	
	393	
	394	
	395	
	396	
	397	
	398	
	399	
	400	
	401	
	402	
	403	
	404	
	405	
	406	
	407	
	408	
	409	
	410	
	411	
	412	
	413	
	414	
	415	
	416	
	417	
	418	
	419	
	420	
	421	
	422	
	423	
	424	
	425	
	426	
	427	
	428	
	429	
	430	
	431	
	432	
	433	
	434	
	435	
	436	
	437	
	438	
	439	
	440	
	441	
	442	
	443	
	444	
	445	
	446	
	447	
	448	
	449	
	450	
	451	
	452	
	453	
	454	
	455	
	456	
	457	
	458	
	459	
	460	
	461	
	462	
	463	
	464	
	465	
	466	
	467	
	468	
	469	
	470	
	471	
	472	
	473	
	474	
	475	
	476	
	477	
	478	
	479	
	480	
	481	
	482	
	483	
	484	
	485	
	486	
	487	
	488	
	489	
	490	
	491	
	492	
	493	
	494	
	495	
	496	
	497	
	498	
	499	
	500	
	501	
	502	
	503	
	504	
	505	
	506	
	507	
	508	
	509	
	510	
	511	
	512	
	513	
	514	
	515	
	516	
	517	
	518	
	519	
	520	
	521	
	522	
	523	
	524	
	525	
	526	
	527	
	528	
	529	
	530	
	531	
	532	
	533	
	534	
	535	
	536	
	537	
	538	
	539	
	540	
	541	
	542	
	543	
	544	
	545	
	546	
	547	
	548	
	549	
	550	
	551	
	552	
	553	
	554	
	555	
	556	
	557	
	558	
	559	
	560	
	561	
	562	
	563	
	564	
	565	
	566	
	567	
	568	
	569	
	570	
	571	
	572	
	573	
	574	
	575	
	576	
	577	
	578	
	579	
	580	
	581	
	582	
	583	
	584	
	585	
	586	
	587	
	588	
	589	
	590	
	591	
	592	
	593	
	594	
	595	
	596	
	597	
	598	
	599	
	600	
	601	
	602	
	603	
	604	
	605	
	606	
	607	
	608	
	609	
	610	
	611	
	612	
	613	
	614	
	615	
	616	
	617	
	618	
	619	
	620	
	621	