1. Permutations of a Multiset. Here it is presented a C language function to generate the successor of a given permutation according to the algorithm discussed in §2. The multiset to be permuted is M. Function *successor*( ) gets permutation p and produces p'. Alphabet A is normalized into the set of the first m integer numbers. A *main*( ) procedure recursively calls function *successor*( ) up to Λ in order to produce permutations from p_0^M to p_∞^M. Specific functions have been provided to compute the orbits of functions of those permutations.

More information can be found in the articles "A Formal Model and an Algorithm for Generating the Permutations of a Multiset," *WSEAS Transactions on Systems*, Vol. 1, No. 1;
available at https://goo.gl/zaatLv
and "Permutation Numbers," *Complex Systems*, Vol. 15, Issue 2;
available at https://goo.gl/Tefm8j.

```
⟨ prologue 2 ⟩
⟨ global variables 3 ⟩
⟨ successor permutation function 4 ⟩
⟨ initialization and normalization 9 ⟩
⟨ main 11 ⟩
```

2. Starting section: headers, constants, and global variables. In particular it is defined constant `EOP` (which stands for "`End Of Permutations`"), constant `MAX_MULTISET_CARD`, an upper threshold for $n = \mathrm{card}_M$, and costant `MAX_ALPHABET_CARD`, an upper threshold for $m = \mathrm{card}_A$.

```
#define EOP Λ
#define MAX_MULTISET_CARD 100
#define MAX_ALPHABET_CARD 10
```

```
⟨ prologue 2 ⟩ ≡
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <assert.h>
```

This code is used in section 1.

3. A set of global variables have been introduced to minimize the parameter exchange in function calls. M is the string to be permuted, whose length is card_M byte (up to MAX_MULTISET_CARD). $\overline{R}$ counts the occurences of each digit, so it consists of card_A cells. The *offset* variable is used to normalize back and forth the permuted string.

⟨ global variables 3 ⟩ ≡

```
char M[MAX_MULTISET_CARD];    /* multiset M */
int card_M;    /* n i.e., its cardinality */
unsigned char *R̄;    /* multisubset R̄ */
int card_A;    /* m, or the number of different symbols in Alphabet A */
int offset;    /* ascii(a_1), used for normalizing M to [0, . . . m − 1] */
int ptoa(char *, int, int);
void initialize(char *, int *, int *, int *);
void printv(char *);
void printv1v2(char *);
void printv1v2v3(char *);
void printv1v2v3v4(char *);
void err(char *);
void printE(char *);
void printD(char *);
void printLogD(char *);
void print2D(char *);
void print3D(char *);
void printLR(char *);
void(*printOrbit)(char *);
char *fname;
FILE *f;
struct strobj {
  char *s, *sprime;
};
typedef struct strobj obj;
void doNought(char *v)
{
  return;
}
unsigned char verbose = 0;
```

This code is used in section 1.

4. The successor operator. Some sort of a Turing machine with two contiguous heads, Head$_{left}$ and Head$_{right}$, initially positioned on the last two characters on the right end of the permutation. They move leftward looking for a couple which is *not* an inversion i.e. $*$Head$_{left}$ is less than $*$Head$_{right}$. As they travel across the string, $\overline{R}$ records the occurrences of encountered characters. If a non-inversion is found, a_i stands below Head$_{left}$. It is substituted by the minimum a_k in $\overline{R}$ which is greater than a_i. Then $\overline{R}$ is linearly scanned producing a zero permutation of $\overline{R}$. If all couples are inversions the string is decreasingly ordered i.e., is a p_∞ in which case *successor*( ) returns EOP.

#define the leftmost symbol of permutation M i.e., v v

```
⟨ successor permutation function 4 ⟩ ≡
  char *successor(char *v, int len)
  {
    char *Head$_{left}$, *Head$_{right}$;
    int i, j, k;
    Head$_{left}$ = &v[len − 2], Head$_{right}$ = &v[len − 1];     /* move the head on the right end of v */
    bzero((char *) $\overline{R}$, card$_A$);     /* $R \leftarrow \emptyset$ */
    ⟨ inspect the permutation right-to-left looking for a non-inversion in $a_i$ 5 ⟩
    if (Head$_{right}$ ≡ the leftmost symbol of permutation M i.e., v)
        /* no inversion means $p = p_\infty$, so $p' = \Lambda$ */
      return EOP;
    $\overline{R}$[*Head$_{left}$]++;     /* $R \cup \{a_i\}$ */
    ⟨ looks for a k which is the minimum j such that $a_j > a_i$ 6 ⟩
    $\overline{R}$[k]−−;     /* $\{a_i\} \cup \mathcal{C}_R\{a_k\}$ */
    *Head$_{left}$++ = k;     /* $a_i \leftarrow a_k$ */
    ⟨ builds $p_0^{\overline{R}}$ 7 ⟩
    return v;
  }
```

This code is used in section 1.

5. Move the heads up to a couple (Head$_{left}$,Head$_{right}$) such that $*$Head$_{left}$ < $*$Head$_{right}$ or the left end of the permutation. Vector $\overline{R}$[ ] counts the occurences of visited symbols.

```
⟨ inspect the permutation right-to-left looking for a non-inversion in $a_i$ 5 ⟩ ≡
  while (Head$_{right}$ ≠ the leftmost symbol of permutation M i.e., v) {
    $\overline{R}$[*Head$_{right}$]++;     /* add the symbol to $\overline{R}$ */
      /* if (*pl– ¡ *pr–) break; /* shift to left both Head$_{right}$ and Head$_{left}$ */
    if (*Head$_{left}$ < *Head$_{right}$) break;
    Head$_{right}$ = Head$_{left}$−−;
      /* alternatively, if (*Head$_{left}$ < *Head$_{right}$) break; Head$_{right}$ = Head$_{left}$−−; */
}
```

This code is used in section 4.

6. if ($*$Head$_{left}$,$*$Head$_{right}$) is *not* an inversion then we need to substitute Head$_{left}$ (i.e., a_i) with the minimum of its majorants.

```
⟨ looks for a k which is the minimum j such that $a_j > a_i$ 6 ⟩ ≡
  for (k = *Head$_{left}$ + 1; k < card$_M$; k++)
    if ($\overline{R}$[k]) break;
```

This code is used in section 4.

7. Closings: we substituted a_k for a_i and now we build an ordered postfix string i.e., a zero for $\overline{R}$. This is made easy because we have $\overline{R}$ which orderly counts the occurrence of the symbols in $\overline{R}$.

$\langle$ builds $p_0^{\overline{R}}$ 7 $\rangle \equiv$

```
  for $(i = 0;\ i < \mathrm{card}_A;\ i{+}{+})$
    for $(j = 0;\ j < \overline{R}[i];\ j{+}{+})$ $*\mathrm{Head}_{left}{+}{+} = i;$
```

This code is used in section 4.

8. Prints a permutation and computes $\nu(p)$.

```
  void printv(char *v)
  {
    int i;
    long l;
    static long old_l;
    static int num;

    ++num;
#define QUANTUM 1871100
    if (num % QUANTUM ≡ 0 ∨ num % QUANTUM ≡ 1) printf("%d\t", num);
    for (l = 0L, i = 0; i < card_M; i++) {
      l = l * card_A + v[i];
      if (num % QUANTUM ≡ 0 ∨ num % QUANTUM ≡ 1) putchar(v[i] + offset);
    }
    if (num % QUANTUM ≡ 0 ∨ num % QUANTUM ≡ 1) putchar('\n');
    if (old_l)
      if (l − old_l > 0) fprintf(f, "%f\n", log(l − old_l));
    old_l = l;
  }
  void printE(char *v)
  {
    int i;
    long l;
    static long old_l;
    static int num;

    ++num;
    if (verbose) printf("%d\t", num);
    for (l = 0L, i = 0; i < card_M; i++) {
      l = l * card_A + v[i];
      if (verbose) putchar(v[i] + offset);
    }
    if (verbose) putchar('\n');      /* num(p) */
    fprintf(f, "%d\n", l);
    old_l = l;
  }
  void printD(char *v)
  {
    int i;
    long l;
    static long old_l;
    static int num;

    ++num;
    if (verbose) printf("%d\t", num);
    for (l = 0L, i = 0; i < card_M; i++) {
      l = l * card_A + v[i];
      if (verbose) putchar(v[i] + offset);
    }
    if (verbose) putchar('\n');      /* D(num(p)) */
    fprintf(f, "%ld\n", l − old_l);
    old_l = l;
  }
```

```
void printLogD(char *v)
{
  int i;
  long l;
  static long old_l;
  static int num;
  ++num;
  if (verbose) printf("%d\t", num);
  for (l = 0L, i = 0; i < card_M; i++) {
    l = l * card_A + v[i];
    if (verbose) putchar(v[i] + offset);
  }
  if (verbose) putchar('\n');      /* Log (D(num(p))) */
  if (old_l)
    if (l − old_l > 0) fprintf(f, "%f\n", log(l − old_l));
  old_l = l;
}
void printLR(char *v)
{
  int i;
  char *Head_left, *Head_right;
  static char *p;
  if (p ≡ Λ) {
    putchar('\\');
    putchar('{');
    for (i = 0; i < card_M − 1; i++) {
      putchar(v[i] + offset);
      putchar(',');
      putchar('␣');
    }
    putchar(v[i] + offset);
    printf("\\},␣\\emptyset␣\\rightarrow␣\\linebreak\n");
    p = (char *) 1;
  }
  Head_left = &v[card_M − 2], Head_right = &v[card_M − 1];      /* move the head on the right end of v */
  while (Head_right ≠ v) {
    if (*Head_left < *Head_right) break;
    Head_right = Head_left−−;
  }
  if (Head_right ≡ v) {
    printf("\\linebreak\n");
    printf("\\emptyset,␣\\{\n");
    for (i = 0; i < card_M − 1; i++) {
      putchar(v[i] + offset);
      putchar(',');
      putchar('␣');
    }
    putchar(v[i] + offset);
    printf("\\}\n");
  }
  else {
    p = v;
    putchar('\\');
    putchar('{');
    while (p < Head_left) {
      putchar((*p++) + offset);
      putchar(',');
      putchar('␣');
    }
    putchar((*p++) + offset);
    putchar('\\');
    putchar('}');
    putchar(',');
    putchar('␣');
    putchar('\\');
    putchar('{');
    while (p < v + card_M − 1) {
      putchar((*p++) + offset);
      putchar(',');
      putchar('␣');
    }
    putchar((*p++) + offset);
    printf("\\}␣\\rightarrow\n");
  }    /* 0123 LR -¿ 0,1, 2, 3 0132 LR -¿ 0, 1, 3, 2 0213 LR -¿ 0, 2, 1, 3 0231 LR -¿ 0, 2, 3, 1 0312 LR
         -¿ 0, 3, 1, 2 0321 LR -¿ 0, 3, 2, 1 .... ...... 3210 -¿ 3 2 1 0 LR */
}
int ptoa(char *p, int l, int base)
{
  int i, res;
  for (res = i = 0; i < l; i++) res = res * base + p[i];
  return res;
}
void printv1v2(char *v)
{
  int l;
  l = card_M;
  assert(l % 2 ≡ 0);
  l ≫= 1;
  printf("%d,␣%d\n", ptoa(v, l, card_A), ptoa(v + l, l, card_A));
}
void printv1v2v3(char *v)
{
  int l;
  l = card_M;
  assert(l % 3 ≡ 0);
  l /= 3;
  printf("%d,␣%d,␣%d\n", ptoa(v, l, card_A), ptoa(v + l, l, card_A), ptoa(v + l + l, l, card_A));
}
void printv1v2v3v4(char *v)
{
  int l;
  l = card_M;
  assert(l % 4 ≡ 0);
  l /= 4;
  printf("%d,␣%d,␣%d,␣%d\n", ptoa(v, l, card_A), ptoa(v + l, l, card_A), ptoa(v + l + l, l, card_A),
      ptoa(v + 3 ∗ l, l, card_A));
}
void dump(char ∗R̄)
{
  int i;

  printf("overlinedR=");
  for (i = 0; i < card_M; i++) printf("%1d", R̄[i]);
  printf("\n");
}
```

9. Initialization: the permutation is normalized, its length is computed in $*sl$.

⟨ initialization and normalization 9 ⟩ ≡

```
  void initialize(char ∗s, int ∗sl, int ∗cl, int ∗offset)
  {
    int min, max;
    int i;
    char ∗p = s;
    char c;

    ∗sl = strlen(s);
    min = 255, max = 0;
    while (c = ∗p++) {
      if (c < min) min = c;
      if (c > max) max = c;
    }
    ∗offset = min;     /∗ offset will be used by printv( ) ∗/
    ∗cl = max − min + 1;     /∗ card_A, or the number of classes ∗/
    R̄ = (unsigned char ∗) malloc(∗cl);     /∗ normalization in 0..max-min ∗/
    for (i = 0; i < ∗sl; i++) s[i] −= min;
  }
```

This code is used in section 1.

10. error print procedure

```
  void err(char ∗s)
  {
    fprintf(stderr, "error␣\"%s\"␣-␣bailing␣out.\n", s);
    exit(−1);
  }
```

11. General main

```
⟨ main 11 ⟩ ≡
  int main(int argc, char *argv[ ])
  {
    int i, status;
    obj o;
    int compute(obj);
    if (argc < 2) err("too␣few␣args.␣Valid␣args:␣[edloi23]␣and␣LR");
    o.sprime = '\0';
    for (i = 1; i < argc; i++) {
      if (argv[i][0] ≡ '-')
        switch (argv[i][1]) {
        case 'L':
          if (argv[i][2] ≡ 'R') {
            printOrbit = printLR;
            printf("\\(\n");
          }
          else err("args.␣Valid␣args:␣-[edloi23]␣and␣-LR\n");
          break;
        case 'e': printOrbit = printE;
          printf("printOrbit␣=␣num(p)\n");
          break;
        case 'd': printOrbit = printD;
          printf("printOrbit␣=␣D(num(p))\n");
          break;
        case 'l': printOrbit = printLogD;
          printf("printOrbit␣=␣log(␣D(num(p))␣)\n");
          break;
        case '2': printOrbit = print2D;
          printf("printOrbit␣=␣(␣num(p_l),␣num(p_r)␣)\n");
          break;
        case '3': printOrbit = print3D;
          printf("printOrbit␣=␣(␣num(p_l),␣num(p_c),␣num(p_r)␣)\n");
          break;
        case 'o': fname = strdup(argv[++i]);
          if (fname ≡ Λ) err("args");
          break;
        case 'i': o.sprime = strdup(argv[++i]);
          o.s = strdup(o.sprime);
          break;
        case 'v': verbose = 1;
          break;
        default: err("args.␣Valid␣args:␣-[edloi23]␣and␣-LR\n");
        }
      else err("args.␣Valid␣args:␣[edloi23]␣and␣-LR\n");
    }
    if (printOrbit ≡ Λ) {
      fprintf(stderr, "no␣orbit␣printing␣was␣chosen␣(-[edl23]␣or␣-LR)\n");
      if (fname ≠ Λ) err("specify␣how␣to␣print␣orbits␣([edl23]␣or␣-LR)");
      printOrbit = doNought;
    }
    if (o.sprime ≡ '\0') {
      err("no␣input␣string");
    }
    if (fname ≡ Λ) fname = "istogram";
    f = fopen(fname, "w");
    if (f ≡ Λ) err("can't␣open␣istogram␣file");
    compute(o);
    if (printOrbit ≡ printLR) printf("\\)\n");
    fclose(f);
    return 0;
  }
  int compute(obj o)
  {
    char *p = o.s;
    char *p' = o.sprime;
    int i;

    strcpy(M, p);
    initialize(M, &card_M, &card_A, &offset);
#ifdef PRINT
    printf("cardM=%d,␣cardA=%d,␣p=%s,␣pprime=%s\n", card_M, card_A, p, p');
#endif
    for (i = 0; i < card_M; i++) p'[i] −= offset;
    do {
      (*printOrbit)(M);
      if (¬successor(M, card_M)) break;
    } while (M ∧ memcmp(M, p', card_M));
  }
  void print2D(char *v)
  {
    int l;

    l = card_M;
    assert(l % 2 ≡ 0);
    l ≫= 1;
    fprintf(f, "%d,␣%d\n", ptoa(v, l, card_A), ptoa(v + l, l, card_A));
    if (verbose) printf("%d,␣%d\n", ptoa(v, l, card_A), ptoa(v + l, l, card_A));
  }
  void print3D(char *v)
  {
    int l;

    l = card_M;
    assert(l % 3 ≡ 0);
    l /= 3;
    fprintf(f, "%d,␣%d,␣%d\n", ptoa(v, l, card_A), ptoa(v + l, l, card_A), ptoa(v + l + l, l, card_A));
    if (verbose) printf("%d,␣%d,␣%d\n", ptoa(v, l, card_A), ptoa(v + l, l, card_A), ptoa(v + l + l, l, card_A));
  }    /* END OF FILE PERM.W */
```

This code is used in section 1.

⟨ builds $p_0^{\overline{R}}$ 7 ⟩ Used in section 4.
⟨ global variables 3 ⟩ Used in section 1.
⟨ initialization and normalization 9 ⟩ Used in section 1.
⟨ inspect the permutation right-to-left looking for a non-inversion in a_i 5 ⟩ Used in section 4.
⟨ looks for a k which is the minimum j such that $a_j > a_i$ 6 ⟩ Used in section 4.
⟨ main 11 ⟩ Used in section 1.
⟨ prologue 2 ⟩ Used in section 1.
⟨ successor permutation function 4 ⟩ Used in section 1.