1.   Introduction.   K-NOBEL is a project to try to predict the future Laureates of Nobel prize of Physics using K-index to rank the researchers. Another parameter, h-index, is used to evaluate the error threshold, since h-index is used by Web of Science as one of the indices to predict the Laureates of Nobel prize.

The program has the following structure:

#**include <stdio.h>**
#**include <stdlib.h>**
⟨ Include headers 28 ⟩
⟨ Macro declarations 6 ⟩
⟨ Type definitions 34 ⟩
⟨ Data structures 22 ⟩
⟨ Internal variables 11 ⟩
⟨ Static functions 3 ⟩

2.   int $main$(**int** $argc$, **char** $**argv$)
{
    ⟨ Local variables 23 ⟩
    ⟨ Parse program arguments 10 ⟩
    ⟨ Initialize the variables that need memory allocation 25 ⟩
    ⟨ Load the ids of Nobel Laureates 43 ⟩
    ⟨ Load authors information 26 ⟩
    ⟨ Calculate K index 51 ⟩
    ⟨ Sort the authors 65 ⟩
    ⟨ Write results to a file 67 ⟩
    ⟨ Write a table with the twelve larger ks in latex format 69 ⟩
    ⟨ Free up memory 27 ⟩
    ⟨ Print information about flags 12 ⟩
    return 0;
}

3.   Some internal functions are defined to embed repetitive tasks like check null pointers and print error messages.

⟨ Static functions 3 ⟩ ≡
    static FILE $*Fopen$(**char** $*filename$, **char** $*mode$)
    {
        FILE $*f$;
        $f = fopen(filename, mode)$;
        if $(\neg f)$ {
            $fprintf(stderr, \texttt{"Could\textvisiblespace not\textvisiblespace open\textvisiblespace \%s\textbackslash n"}, filename)$;
            $exit(-1)$;
        }
        return f;
    }
See also sections 4, 5, 7, 17, 18, 19, 20, 31, 35, 36, 37, 38, 39, 40, 41, 47, 62, 63, and 64.

This code is used in section 1.

4.   ⟨ Static functions 3 ⟩ +≡
    static void $Fclose$(**FILE** $*f$)
    {
        if (f) $fclose(f)$;
    }

5.   The *mem_free* function check the nullity of the address pointed by *ptr* before deallocation.

⟨ Static functions 3 ⟩ +≡
  static void *mem_free* (**void** *∗ptr*, **int line**)
  {
    if (*ptr*) *free*(*ptr*);
  }

6.   ⟨ Macro declarations 6 ⟩ ≡
#**define** CALLOC(*count*, *nbytes*)*mem_calloc* ((*count*), (*nbytes*), (**int**) __LINE__)
See also sections 8 and 9.

This code is used in section 1.

7.   ⟨ Static functions 3 ⟩ +≡
  static void ∗ *mem_calloc* (**int** *count*, **int** *nbytes*, **int line**) { **void** *∗ptr*;
  ptr = *calloc*(*count*, *nbytes*); **if** (¬*ptr*) { *fprintf* (*stderr*, "%d:␣Null␣pointer\n", **line**) , *abort*( ); }
    return *ptr*; }

8.   FREE macro wraps *mem_free* with proper arguments and zeroed *ptr*.

⟨ Macro declarations 6 ⟩ +≡
#**define** FREE(*ptr*) ((**void**)(*mem_free*((*ptr*), __LINE__), (*ptr*) = 0))

9.   The *panic* function is used when the program enters in a condition that was not expected to be in. It stops the program execution and prints a message *msg*. If there was a sure expectation that nothing bad can occurs, a definition of NDEBUG as macro turn off the *panic* function.

⟨ Macro declarations 6 ⟩ +≡
#**undef** *panic*
#**ifdef** NDEBUG
#**define** *panic*(*msg*) ((**void**) 0)
#**else**
  extern void *panic*(**int** *msg*);
#**define** *panic*(*msg*) (*fprintf* (*stderr*, "%d:␣PANIC:␣%s\n", (**int**) __LINE__, *msg*), *abort*( ))
#**endif**

10.   Verbose mode.   The flag `-v` is provided to print the existing comments inside data files and any other useful information to the user.

#define `VERBOSE_FLAG` `"-v"`

⟨ Parse program arguments 10 ⟩ ≡
   if $(\mathit{argc} \equiv 2 \land \neg \mathit{strncmp}\,(\mathit{argv}[1], \texttt{VERBOSE\_FLAG}, 3))$
      $\mathit{verbose} = 1;$

See also section 14.

This code is used in section 2.

11.   The *verbose* Boolean variable marks if the output of the program is extended with the comments inside data files. The default behavior is to write to the output the name the generated files.

⟨ Internal variables 11 ⟩ ≡
   static int $\mathit{verbose} = 0;$

See also sections 13, 16, 24, 33, and 61.

This code is used in section 1.

12.   Warn the user about the `-v` if the flag was not used.

⟨ Print information about flags 12 ⟩ ≡
   if $(\neg \mathit{verbose})$
      $\mathit{fprintf}\,(\mathit{stderr}, \texttt{"-\_use\_\textbackslash"\%s\_-v\textbackslash"\_to\_print\_information\_about\_data\_set.\textbackslash n"}, \mathit{argv}[0]);$

See also section 15.

This code is used in section 2.

13.   The flag `-vvv` causes the program to print the values of the indices moments before they are reached. It's used to check the correctness of the algorithms used to calculated the indices. The Boolean variable used to mark the mode is *confess*.

#define `CONFESS_FLAG` `"-vvv"`

⟨ Internal variables 11 ⟩ +≡
   static int $\mathit{confess} = 0;$

14.   The program doesn't accept both flags, `-v` and `-vvv`, to avoid an output complexity in terms of information and to set a boundary between the two tasks.

⟨ Parse program arguments 10 ⟩ +≡
   if $(\mathit{argc} \equiv 2 \land \neg \mathit{strncmp}\,(\mathit{argv}[1], \texttt{CONFESS\_FLAG}, 5))$
      $\mathit{confess} = 1;$

15.   The user of the program is warned about the flag `-vvv` if the flag was not used.

⟨ Print information about flags 12 ⟩ +≡
   if $(\neg \mathit{confess})$
      $\mathit{fprintf}\,(\mathit{stderr}, \texttt{"-\_use\_\textbackslash"\%s\_-vvv\textbackslash"\_to\_show\_details\_about\_K-index\_calculation.\textbackslash n"}, \mathit{argv}[0]);$

16. In confess mode, a queue is necessary to not lost previous values of some variable already processed. The queue is implemented using a circular array where the field *front* is the index of the first element and *rear* the index of last element. There's no problem in overwriting some queue elements because only a limited number of values `PREV_NVALS` lesser than the queue length `QLEN` are of interest.

#define `QLEN` 32   /∗ queue length ∗/
#define `PREV_NVALS` 5   /∗ number of elements of interest in the queue ∗/

⟨ Internal variables 11 ⟩ +≡
  static struct queue_struct {
    int *array*[`QLEN`];
    int *front*, *rear*;
  } *queue*;

17. Add the value *idx* in the rear of the queue.

⟨ Static functions 3 ⟩ +≡
  static void *enqueue*(**int** *idx*)
  {
    queue.array[*queue.rear* ++] = *idx*;
    if (*queue.rear* ≡ `QLEN`)
      queue.rear = 0;
  }

18. The function *queue_is_empty* returns 1 when the queue is empty.

⟨ Static functions 3 ⟩ +≡
  static int *queue_is_empty*( )
  {
    return *queue.rear* ≤ *queue.front*;
  }

19. The function *dequeue_from_rear* removes the element in the rear of the queue returning it. There is no need to remove elements in front of the queue.

⟨ Static functions 3 ⟩ +≡
  static int *dequeue_from_rear*( )
  {
    int *idx*;
    if (*queue.rear* ≤ *queue.front*) {
      panic("Queue␣is␣empty");
    }
    idx = *queue.array*[−− *queue.rear*];
    return *idx*;
  }

20. The queue fields *front* and *rear* are initialized using *queue_reset*.

⟨ Static functions 3 ⟩ +≡
  static void *queue_reset*( )
  {
    queue.front = *queue.rear* = 0;
  }

21. Input data.   The data to be processed comes from CSV (comma-separated values) and TSV (tab-separated vaules) files containing, among other data, the papers and its number of citations (CSV) or number of citings (TSV) of researchers. Each file stores data about one researcher. The citing is the number of citations received by a paper that cites the researcher paper in question. The CSV files are used to calculate the h-index and TSV are used to find the K-index. An index file with author's identification and some information like his/her homepage is used to associate the data files. For example, an author with an Researcher ID equals to "Z-1111-1900" has the papers' citations in a file called "Z-1111-1900.csv" and the papers' citings in a file named "Z-1111-1900.tsv". The data files were saved inside in the value of `DATA_DIRECTORY` macro directory.

#define `DATA_DIRECTORY  "data"`

22. Fetching authors' record.   The macro `AUTHORS_DATA_FN` is set with the file name that contains information about researchers (authors). Each line of the file has the name, Web of Science, Google Scholar or Publons research id and a link to a page containing more information about the author's publications. Not all authors have researcher id, when this occurs, we assign a number and link to the Web of Science page. The author's h-index and K-index are assigned to the fields h and K, respectively.

#define `AUTHORS_DATA_FN` `"authors.idx"`
#define `MAX_STR_LEN` 256

⟨ Data structures 22 ⟩ ≡
  typedef struct author_struct {
    char $name$[`MAX_STR_LEN`];
    char $researchid$[`MAX_STR_LEN`];
    char url[`MAX_STR_LEN`];
    int h;
    int k;
    char $timestamp$[`MAX_STR_LEN`];   /∗ last modification of record ∗/
  } **Author**;
This code is used in section 1.

23. `MAX_LINE_LEN` is the maximum length of each line, the value is very high because some papers have too many authors.

#define `MAX_LINE_LEN` $1 \ll 16$

⟨ Local variables 23 ⟩ ≡
  char ∗fn;   /∗ file name ∗/
  FILE ∗fp;   /∗ file pointer ∗/
  char $buffer$[`MAX_LINE_LEN`];   /∗ buffer to store strings ∗/
  char
  line [`MAX_LINE_LEN`] ;   /∗ store file lines ∗/
  int $i = 0$, $j = 0$;   /∗ general-purpose counters ∗/
See also sections 30, 52, 54, 57, 66, and 68.
This code is used in section 2.

24.   An array of structures is used to store the *authors'* information. The global variable A is set with the number of authors processed at the time it is read.

⟨ Internal variables 11 ⟩ +≡
  static $Array*authors$;   /∗ store authors' info ∗/

25.   The maximum number of authors is dictated by the macro `MAX_N_AUTHORS` and the array of *authors* is initialized using this value.

#define `MAX_N_AUTHORS` #2000

⟨ Initialize the variables that need memory allocation 25 ⟩ ≡
  $authors = Array\_new(\texttt{MAX\_N\_AUTHORS}, \textbf{sizeof}(\textbf{Author}));$
See also section 44.
This code is used in section 2.

26.   Authors basic information was picked from the Web of Science page, more specifically at `https://hcr.clarivate.com/#categories%3Dphysics` that is the page of most cited authors in physics. They are stored in a file named *authors.idx* that is opened to load this information.

⟨ Load authors information 26 ⟩ ≡
  fp = *Fopen*(`AUTHORS_DATA_FN`, `"r"`);
  while (*fgets* (**line** , `MAX_LINE_LEN`, *fp*) ≠ Λ) {
  if (*is_comment* (**line**))
  continue;
  ⟨ Begin to fill authors structure 29 ⟩
  } *Fclose*(*fp*);

This code is used in section 2.

27.   Memory allocated for the array of pointers *authors* is freed.

⟨ Free up memory 27 ⟩ ≡
  Array_free(*authors*);

See also section 46.

This code is used in section 2.

28.   ⟨ Include headers 28 ⟩ ≡
#**include <string.h>**   /∗ strtok() ∗/

See also section 42.

This code is used in section 1.

29.   The fields are separated by semicolon inside *authors.idx*, a record in the file looks like

<div align="center">

`L-000-000;Joe Doe;http//joedoe.joe`

</div>

where the first field `L-000-000` is the Research ID or ORCID, when the author doesn't have an identifier, a custom number is assigned. The second field `Joe Doe` is the author name and the third field is the link to the page that contains information about author's publications. A structure is loaded with these data and a pointer to this structure is passed to the array *authors*. Lately, *h*-index and *K*-index will be calculated and assigned to the proper field in the structure.

#**define** `IDX_SEP  ";\n"`

⟨ Begin to fill authors structure 29 ⟩ ≡

  $i = 0$;   /∗ information index ∗/

  $ptr = strtok$ (**line** , `IDX_SEP`) ;

  while ($ptr \neq \Lambda$) {

    switch (i) {

    case 0: $strncpy(aut.researchid, ptr, $`MAX_STR_LEN`$)$;

      break;

    case 1: $strncpy(aut.name, ptr, $`MAX_STR_LEN`$)$;

      break;

    case 2: $strncpy(aut.url, ptr, $`MAX_STR_LEN`$)$;

      break;

    case 3: $aut.h = atoi(ptr)$;

      if ($aut.h \leq 0$) {

        $fprintf(stderr, $`"==>␣h=%d␣<==\n"`$, aut.h)$;

        $panic($`"Wrong␣value␣of␣h−index,␣run␣confess␣mode."`$)$;

      }   /∗ Initialize K too ∗/

      $aut.k = 0$;

      break;

    case 4: $strncpy(aut.timestamp, ptr, $`MAX_STR_LEN`$)$;

      break;

    default: **break**;

    }

    $ptr = strtok(\Lambda, $`IDX_SEP`$)$;

    $i{+}{+}$;

  }

  if ($\neg is\_nobel\_laureate(\& aut)$) {

    $Array\_append(authors, \& aut)$;

  }

This code is used in section 26.

30.   *aut* is used to point to new allocated **Author** structure adress while the fields are assigned with the proper values.

⟨ Local variables 23 ⟩ +≡

  Author *aut*;   /∗ temporary variable ∗/

31.   In all custom files used to parse the data, the hash character "#" is used to indicate that after it the
following tokens must be interpreted as comments.

⟨ Static functions 3 ⟩ +≡

   int *is_comment* (**char** ∗ **line**) {

   if (¬ **line**) **goto** *exit_is_comment*;

   if (**line** [0] ≡ '#') {

   if (*verbose*) *fprintf* (*stderr*, `"%s"`, **line**) ;

   return 1; }

exit_is_comment: **return** 0; }

32.   Fetching Nobel laureates.   We have to discard researchers that already was awarded with the Nobel Prize. Up to 2018, there was 935 laureates. We put more chairs in the room to accommodate future laureates. A simple array is used to store the IDs and a linear search is performed. As the number of winners is not high, this simple scheme, even though not so efficient, is used to avoid complexities.

#define `N_LAUREATES` 935
#define `MORE_ROOM` 128

33.   The Nobel Laureates identifier are inserted in the *list* array;

⟨ Internal variables 11 ⟩ +≡
  static *Array* ∗*list*;

34.   The *Array* data structure is used to manage sequential allocation of related elements. The data is copied to *array* field, the *cap* field is the maximum number of elements provided by the array, *length* is the number of elements occupied in the array and *size* is the number of bytes occupied by each element. All elements are of the same size.

⟨ Type definitions 34 ⟩ ≡
  typedef struct array_struct {
    void ∗*array*;
    int *cap*;   /∗ capacity of the array in number of elements ∗/
    int *length*;   /∗ number of elements used ∗/
    int *size*;   /∗ size in bytes of each element of the array ∗/
  } **Array**;

This code is used in section 1.

35.   To create an array, memory is allocated for the structure and the data.

⟨ Static functions 3 ⟩ +≡
  static Array ∗*Array_new*(**int** *capacity*, **int** *size*)
  {
    Array ∗*ary*;
    ary = `CALLOC`(1, **sizeof**(**Array**));
    ary⃗*array* = `CALLOC`(*capacity*, *size*);
    ary⃗*cap* = *capacity*;
    ary⃗*size* = *size*;
    ary⃗*length* = 0;
    return *ary*;
  }

36.   An element is get by accessing the ith element in the array taking into account the size of each element.

⟨ Static functions 3 ⟩ +≡
  static void ∗*Array_get*(**Array** ∗*ary*, **int** *i*)
  {
    assert(*ary*);
    assert($i \geq 0 \wedge i < ary$⃗*length*);
    return *ary*⃗*array* + *i* ∗ *ary*⃗*size*;
  }

37.   An element is put in the array by copying the bytes of the element *elem* starting at the proper position *i* and taking into account the size of each element.

⟨Static functions 3⟩ +≡
  static void *Array_put*(**Array** *∗ary*, **int** *i*, **void** *∗elem*)
  {
    assert(*ary*);
    assert(*i* ≥ 0 ∧ *i* < *ary⃗cap*);
    assert(*elem*);
    memcpy(*ary⃗array* + *i* ∗ *ary⃗size*, *elem*, *ary⃗size*);
  }

38.   ⟨Static functions 3⟩ +≡
  static void *Array_append*(**Array** *∗ary*, **void** *∗elem*)
  {
    assert(*ary*);
    assert(*elem*);
    memcpy(*ary⃗array* + *ary⃗length* ∗ *ary⃗size*, *elem*, *ary⃗size*);
    ary⃗length ++;
  }

39.   Memory of array structure are freed by deallocating the data field *array* and the structure itself.

⟨Static functions 3⟩ +≡
  static void *Array_free*(**Array** *∗ary*)
  {
    FREE(*ary⃗array*);
    FREE(*ary*);
  }

40.   *Array_length* returns the number of elements in the *array*.

⟨Static functions 3⟩ +≡
  static int *Array_length*(**Array** *∗array*)
  {
    assert(*array*);
    assert(*array⃗length* ≥ 0);
    return *array⃗length*;
  }

41.   ⟨Static functions 3⟩ +≡
  static int *Array_size*(**Array** *∗array*)
  {
    assert(*array*);
    assert(*array⃗size* > 0);
    return *array⃗size*;
  }

42.   The function *assert* is used to check some invariants or integrity constraints of the data.

⟨Include headers 28⟩ +≡
#include <assert.h>

43.   A file `NOBEL_FN` with the identification number (id) of the Nobel Laureates is used to check if the researcher already win the prize.

  /* file name with ids of Nobel Laureates */

#define `NOBEL_FN` `"laureates.dat"`

⟨Load the ids of Nobel Laureates 43⟩ ≡
  $fp = Fopen(\texttt{NOBEL\_FN}, \texttt{"r"})$; **while** ($fgets$ (**line** , `MAX_LINE_LEN`, fp) ≠ Λ) { **if** ($is\_comment$ (**line**))
      continue;   /∗ Remove the new line ∗/
  line [$strcspn$ (**line** , `"\r\n"`)] = 0;
  ⟨Insert research id at the end of the list 45⟩
  } $Fclose(fp)$;
This code is used in section 2.

44.   The $list$ array is initialized with enough space to put lareuates.

⟨Initialize the variables that need memory allocation 25⟩ +≡
  $list = Array\_new(\texttt{N\_LAUREATES} + \texttt{MORE\_ROOM}, \texttt{MAX\_STR\_LEN})$;

45.   Each new Laureate id is inserted in the array list and the number of elements in the list is incremented. No overflow checking is done.

⟨Insert research id at the end of the list 45⟩ ≡
  $Array\_append$ ($list$, **line**) ;
This code is used in section 43.

46.   ⟨Free up memory 27⟩ +≡
  $Array\_free(list)$;

47.   The function $is\_nobel\_laureate$ check in the list od laureates with IDs if the author a id is in the list. The string comparison does not take into account if an id is prefix of another one because this is very unlikely to occur.

⟨Static functions 3⟩ +≡
  static int $is\_nobel\_laureate(\textbf{Author} *aut)$
  {
    int i;
    char $*id = aut{\rightarrow}researchid$;
    for $(i = 0; \ i < list{\rightarrow}length; \ i{+}{+})$ {
      if $(strncmp(Array\_get(list, i), id, \texttt{MAX\_STR\_LEN}) \equiv 0)$ **return** 1;
    }
    return 0;
  }

48.   Indices calculation.   There is procedure in this program to calculate the scientometric index K. The h is the Hirsch index proposed by Hirsch [J. E. Hirsch, "An index to quantify an individual's scientific research output," *PNAS* **102** (15) 16569–16572, 2005]. It is obtained at Web of Science, so no further procedure is needed. The K stands for Kinouchi index and was prososed by O. Kinouchi *et al.* [O. Kinouchi, L. D. H. Soares, G. C. Cardoso, "A simple centrality index for scientific social recognition", *Physica A* **491** (1), 632–640].

49. h-index.   The number of papers is in decreasing order of citations that the number of citations is greater than the paper position is the h-index. On Web of Science homepage, the procedure to find the h of an author is as follows:

- Search for an author's publications by *Author* or *Author Identifiers*;
- Click on the link *Create Citation Report*;
- The h-index is showed at the top of the page.

The h-index value is stored in the author record structure and saved in "authors.idx" file.

50.   K-index.   If an author receives at least K citations, where each one of these K citations have get at least K citations, then the author's K-index was found. On Web of Science homepage, the procedure to find the K of an author looks like below:

  ⋆ Search for an author's publications;
  ⋆ Click on the link *Create Citation Report*;
  ⋆ Click on the link *Citing Articles without self-citations*;
  ⋆ Traverse the list, stoping when the rank position of the article were greater than the *Times Cited*;
  ⋆ Subtract on from the rank position, this is the K value.

To calculate in batch mode, we downloaded a file with the data to calculate the K by clicking on the button *Export...* and selecting *Fast 5K* format that saves the same data, with limit of 5.000 records, where each field is separated by one or more tabs that is assigned to the macro `TSV_SEP`.

51.   ⟨Calculate K index 51⟩ ≡
  $N = Array\_length(authors);$
  for ($i = 0;\ i < N;\ i{+}{+}$) {   /∗ for each author ∗/
    ⟨Process tsv file 53⟩
  }

This code is used in section 2.

52.   ⟨Local variables 23⟩ +≡
  int $N = 0;$

53.   To open the proper file the Researcher ID is concatenated with `DATA_DIRECTORY` as prefix and the file extension `K_EXT` as suffix.

#define `K_EXT` `"tsv"`

⟨Process tsv file 53⟩ ≡
  $paut = Array\_get(authors, i);$
  $snprintf(buffer, \texttt{MAX\_LINE\_LEN}, \texttt{"\%s/\%s.\%s"}, \texttt{DATA\_DIRECTORY}, 39paut{\rightarrow}researchid, \texttt{K\_EXT});$
  $fn = buffer;$
  $fp = Fopen(fn, \texttt{"r"});$
  $pos = 1;$
  $ncits = 0, old\_ncits = 1000000;$
  if ($confess$) {
    $fprintf(stderr, \texttt{"\%d.\_\%s\textbackslash n"}, i + 1, paut{\rightarrow}name);$
    $queue\_reset();$
  }
  while (*fgets* (**line** , **sizeof** (**line**) , *fp*) ≠ Λ)
  {
    ⟨Parse the line counting citings 55⟩
  }
  $Fclose(fp);$

This code is used in section 51.

54.   ⟨Local variables 23⟩ +≡
  int $pos;$ /∗ temporary variable to store the paper position ∗/
  int $ncits,\ old\_ncits;$ /∗ current and old value of number of citings ∗/

55.   The file with citings has few lines to ignore, basically it is only one that begins with "PT \t" (ignore double quotes). A line that begins with new line command ignored too, but only for caution.

⟨ Parse the line counting citings 55 ⟩ ≡
  if (*strstr* (**line** , `"PT\t"`) ≠ Λ)
  {
    continue;
  }
  else if (**line** [0] ≡ '\n')
  {   /∗ start with new line ∗/
    continue;
  }
  else {
    ⟨ Find the citings and check if the K-index was found 56 ⟩
  }

This code is used in section 53.

56.   `SKIP` represents the fields to be skipped before *Times Cited* value is reached. Its value is not fixed and for this reason it was implemented a tricky way to get the *Times Cited* value: after `SKIP` is passed, each field is accumulated in a queue and when the end of the record is reached, the queue is dequeue three times to get the *Times Cited* value. This position offset of *Times Cited* value from the end is fixed for all files.

#define `TSV_SEP` `"\t"`
#define `SKIP` 7   /∗ number of fields that can be skipped with safety ∗/

⟨ Find the citings and check if the K-index was found 56 ⟩ ≡
  { *ncits* = 0;
  j = 0; *ptr* = *strtok* (**line** , `TSV_SEP`) ;
  while (*ptr* ≠ Λ) {
    if (j > `SKIP`) {
      push(*ptr*);
    }
    j++;
    ptr = *strtok*(Λ, `TSV_SEP`);
  }
  for (j = 0; j < 3; j++)
    ptr = *pop*( );
  ncits = *atoi*(*ptr*);
  stack_reset( );
  ⟨ Check parsing integrity of citings 58 ⟩
  ⟨ Enqueue temporary index value 59 ⟩
  old_ncits = *ncits*;
  if (*pos* > *ncits*) {   /∗ found k ∗/
    pos −−;
    paut = *Array_get*(*authors*, *i*);
    paut→*k* = *pos*;
    ⟨ Write the last values 60 ⟩
    break;
  }
  pos ++; }

This code is used in section 55.

57.   ⟨ Local variables 23 ⟩ +≡
  char ∗*ptr*;   /∗ Generic pointer ∗/

58.   The articles are listed in descending order of number of citings. For this reason, the old value of number of citings *old_ncits* must not be lesser than current value just parsed *ncits*. The verification stops the program execution if this invariant is not obeyed.

⟨ Check parsing integrity of citings 58 ⟩ ≡
  if (*old_ncits* < *ncits*) {
    fprintf (*stderr*, "==>␣%d<%d␣<==\n", *old_ncits*, *ncits*);
    panic("Previous␣number␣of␣citings␣is␣lesser␣the␣the␣current␣one.");
  }

This code is used in section 56.

59.   ⟨ Enqueue temporary index value 59 ⟩ ≡
  if (*confess*)
    enqueue(*ncits*);

This code is used in section 56.

60.   ⟨ Write the last values 60 ⟩ ≡
  if (*confess*) {
    register int *ii*;
    paut = *Array_get*(*authors*, *i*);
    fprintf (*stderr*, "==>␣found␣K=%d␣<==\n␣<>␣Last␣values\n", *paut⃗k*);
    for (*ii* = 0; *ii* < PREV_NVALS; *ii*++) {
      if (*queue_is_empty*()) **break**;
      fprintf (*stderr*, "␣K:␣pos=%d,␣ncits=%d\n", (*pos*−− + 1), *39dequeue_from_rear*());
    }
  }

This code is used in section 56.

61.   Stack.   A humble stack is implemented to store few pointers using FIFO policy.   The stack is composed by an array of pointers named *data* and an index named *top* to point to the next index to add element in the stack. Three stacks are declared, one for storing temporary values of the fields during K-index calculation, other to store temporary values of citation and other to store temporary values of citings.

#define `STACK_LEN`   #10000

⟨ Internal variables 11 ⟩ +≡
  static struct {
    char ∗*data*[`STACK_LEN`];
    int *top*;
  } *stack*;

62.   Elements are inserted at the top of the stack by invoking *push* and using **char** ∗*ptr* as parameter. The index *idx* is incremented to the number of elements in the stack and $top - 1$ is the index of the element in the top.

⟨ Static functions 3 ⟩ +≡
  static void *push*(**char** ∗*ptr*)
  {
    if (*ptr* ≡ Λ) {
      panic("Tring␣to␣push␣NULL␣value␣to␣the␣stack");
    }
    stack.*data*[*stack*.*top*++] = *ptr*;
    if (*stack*.*top* ≡ `STACK_LEN`) {
      panic("Stack␣overflow");
    }
  }

63.   Elements from the top of the stack are removed by *pop* function. If there is no element in the stack, Λ is returned.

⟨ Static functions 3 ⟩ +≡
  static char ∗*pop*( )
  {
    if (*stack*.*top* ≤ 0) *panic*("Stack␣underflow");
    else return *stack*.*data*[−−*stack*.*top*];
  }

64.   To reset the stack, *top* is assigned to zero.

⟨ Static functions 3 ⟩ +≡
  static void *stack_reset*( )
  {
    stack.*top* = 0;
  }

65.   Sorting.   The authors are classified in descending order according to their K-index. The insertion-sort algorithm is used to simplify the code and according to the number of entries is not so large.

⟨Sort the authors 65⟩ ≡
    $N = Array\_length(authors)$;
    for $(i = 1; \ i < N; \ i\mathbin{+\!+})$ {
        $memcpy(\&aut, Array\_get(authors, i), Array\_size(authors))$;
        for $(j = i - 1; \ j \geq 0; \ j\mathbin{-\!-})$ {
            $qaut = (\mathbf{Author} \ *) \ Array\_get(authors, j)$;
            if $(aut.k < qaut\rightarrow k)$ **break**;
            $Array\_put(authors, j + 1, qaut)$;
        }
        $Array\_put(authors, j + 1, \&aut)$;
    }
This code is used in section 2.

66.   ⟨Local variables 23⟩ +≡
    register Author $*qaut$;

67.   Output.   The results are written as a table in markdown format to the file name assigned to `RANK_FN`. A space is needed between the bars and the content.

#define `RANK_FN` `"k-nobel.md"`

⟨ Write results to a file 67 ⟩ ≡
  $fp = fopen(\texttt{RANK\_FN}, \texttt{"w"});$
  if $(\neg fp)$ {
   $perror(fn);$
   $exit(-4);$
  }
  $fprintf(fp, \texttt{"|\_N\_|\_Author\_|\_h\_|\_K\_|\textbackslash n"});$
  $fprintf(fp, \texttt{"|---|--------|---|---|\textbackslash n"});$
  $N = Array\_length(authors);$
  for $(i = 0;\ i < N;\ i{+}{+})$ {
   $paut = Array\_get(authors, i);$
   $fprintf(fp, \texttt{"|\_\%d\_|\_[\%s]\,(\%s)\_|\_\%d\_|\_\%d\_|\textbackslash n"}, i+1, paut{\rightarrow}name, paut{\rightarrow}url, paut{\rightarrow}h, paut{\rightarrow}k);$
  }
  $fclose(fp);$
  $fprintf(stderr, \texttt{"*\_Wrote\_\textbackslash"\%s\textbackslash"\textbackslash n"}, \texttt{RANK\_FN});$

This code is used in section 2.

68.   ⟨ Local variables 23 ⟩ +≡
  Author $*paut;$

69.   A table with the twelve larger Ks to be included in the manuscript is written in LaTeX format.

⟨ Write a table with the twelve larger ks in latex format 69 ⟩ ≡
  $fn = \texttt{"table.tex"};$
  $fp = fopen(fn, \texttt{"w"});$
  if $(\neg fp)$ {
   $perror(fn);$
   $exit(-8);$
  }
  $fprintf(fp, \texttt{"\textbackslash\textbackslash begin\{tabular\}\{cccc\}\_\textbackslash\textbackslash\textbackslash\textbackslash\_\textbackslash\textbackslash hline\textbackslash n"});$
  $fprintf(fp, \texttt{"\textbackslash\textbackslash bf\_N\_\&\_\textbackslash\textbackslash bf\_Author\_\&\textbackslash\textbackslash bg\_h\_\&\textbackslash\textbackslash bf\_K\_\textbackslash\textbackslash\textbackslash\textbackslash\_\textbackslash\textbackslash hline\textbackslash n"});$
  for $(i = 0;\ i < 12;\ i{+}{+})$ {
   $paut = Array\_get(authors, i);$
   $fprintf(fp, \texttt{"\_\%d\_\&\_\%s\_\&\_\%d\_\&\_\%d\_\textbackslash\textbackslash\textbackslash\textbackslash\textbackslash n"}, i+1, paut{\rightarrow}name, paut{\rightarrow}h, paut{\rightarrow}k);$
  }
  $fprintf(fp, \texttt{"\textbackslash\textbackslash hline\textbackslash\textbackslash end\{tabular\}\textbackslash n"});$
  $fclose(fp);$
  $fprintf(stderr, \texttt{"*\_Wrote\_\textbackslash"\%s\textbackslash"\textbackslash n"}, fn);$

This code is used in section 2.

70. Index.

K-NOBEL