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 files 10 ⟩
⟨ Data structures 6 ⟩
⟨ Internal variables 5 ⟩
⟨ Static functions 3 ⟩
```

2.
```
int main(int argc, char **argv)
{
  ⟨ Parse program arguments 4 ⟩
  ⟨ Load the ids of Nobel Laureates 14 ⟩
  ⟨ Load authors information 8 ⟩
  ⟨ Calculate h index 17 ⟩
  ⟨ Calculate K index 22 ⟩
  ⟨ Sort the authors 31 ⟩
  ⟨ Write results to a file 32 ⟩
  ⟨ Write a table with the twelve larger ks in latex format 33 ⟩
  ⟨ Free up memory 34 ⟩
  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 (¬f) {
      fprintf(stderr, "Could␣not␣open␣%s\n", filename);
      exit(−1);
    }
    return f;
  }
  static void Fclose(FILE *f)
  {
    if (f) fclose(f);
  }
```

See also sections 9, 12, 16, 27, 28, 29, and 30.

This code is used in section 1.

4. The only flag provided is `-v` to print the existing comments inside data files and any other useful information to the user. Any other parameter entered to the program is ignored and causes the program execution without any parameters at all.

```
#define VERBOSE_FLAG "-v"
```

⟨ Parse program arguments 4 ⟩ ≡

```
if (argc ≡ 2 ∧ ¬strncmp(argv[1], VERBOSE_FLAG, 3)) {
  verbose = 1;
}
```

This code is used in section 2.

5. 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 5 ⟩ ≡

```
static int verbose;
```

See also sections 7, 13, and 26.

This code is used in section 1.

6. Authors. 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 or Google Scholar or Publons research id and a link to a page containing more information about the citations. Not all authors have researcher id, when this occurs, we assign a number and link to the Web of Science page. The data structure for author loads this information, and indeed the author's h-index and K-index.

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

⟨ Data structures 6 ⟩ ≡

```
struct author {
  char name[MAX_STR_LEN];
  char researchid[MAX_STR_LEN];
  char url[MAX_STR_LEN];
  int h;
  int k;
};
```

This code is used in section 1.

7. An array of structures is used to store the *authors'* information. `MAX_LINE_LEN` is the maximum length of each line, the value is very high because some papers have too many authors. Some variables are made internal (static) and global because the program is so short and the risk to have inconsistencies is low. This kind of programming technique imposes an attention to details along the program, as an example, the counters must be zeroed each time of using.

```
#define MAX_LINE_LEN 1 ≪ 16
```

⟨ Internal variables 5 ⟩ +≡

```
static struct author **authors;  /* store authors' info */
static struct author *aut;  /* temporary variable */
static char *fn, *p;  /* file name and generic pointer */
static FILE *fp;  /* file pointer */
static char buffer[MAX_STR_LEN];  /* buffer to store strings */
static char
line [MAX_LINE_LEN] ;  /* store file lines */
static int A = 0;  /* store the number of authors */
static int i = 0, j = 0;  /* general-purpose counters */
```

8. 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 openned to load this information. The global counter A stores the number of authors and it is used along the program.

⟨ Load authors information 8 ⟩ ≡

```
fp = Fopen(AUTHORS_DATA_FN, "r"); while ( fgets ( line , MAX_LINE_LEN, fp ) ≠ Λ ) { if ( is_comment (
    line ) ) continue;  /* reallocate the array of authors struct with to pointer elements */
authors = (struct author **) realloc(authors, get_no_authors( ) * sizeof(struct author *));
⟨ Begin to fill authors structure 11 ⟩
} Fclose(fp);
```

This code is used in section 2.

9. The number of research authors is calculated by adding one to global variable A that is the next free array index.

```
⟨ Static functions 3 ⟩ +≡
  static int get_no_authors( )
  {
    return A + 1;
  }
```

10. ⟨ Include files 10 ⟩ ≡

```
#include <string.h>      /* strtok() */
```

This code is used in section 1.

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

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

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 11 ⟩ ≡
  aut = (struct author *) malloc(sizeof(struct author));
  i = 0;      /* information index */
  char *p; p = strtok ( line , IDX_SEP ) ;
  while (p ≠ Λ) {
    switch (i) {
    case 0: strncpy(aut→researchid, p, MAX_STR_LEN);
      break;
    case 1: strncpy(aut→name, p, MAX_STR_LEN);
      break;
    case 2: strncpy(aut→url, p, MAX_STR_LEN);
      break;
    default: break;
    }
    p = strtok(Λ, IDX_SEP);
    i++;
  }
  if (¬is_nobel_laureate(aut)) {
    authors[A++] = aut;
  }
```

This code is used in section 8.

12. 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)
      printf ("%s", line ) ;
  return 1; }
exit_is_comment: return 0; }
```

13. Nobel Laureates. We have to discard researchers that already was laureated with the Nobel Prize. Up to 2018, there was 935 laureates that awarded Nobel Prize. We put more chairs in the room to accomodate future laureated researchers. 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
⟨ Internal variables 5 ⟩ +≡
  static struct arr {
    char array[N_LAUREATES + MORE_ROOM][MAX_STR_LEN];
    int n;      /* number of elements used */
  } list;
```

14. 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 14 ⟩ ≡
  fp = Fopen(NOBEL_FN, "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 in the list 15 ⟩
  } Fclose(fp);
```

This code is used in section 2.

15. 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 in the list 15 ⟩ ≡
  strncpy (list.array[list.n++], line , sizeof ( line ) ) ;
```

This code is used in section 14.

16. The function *is_nobel_laureate* check in the laureated list 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(struct author *a)
  {
    int i;
    char *id = a->researchid;
    for (i = 0; i < list.n; i++) {
      if (strncmp(list.array[i], id, sizeof (id)) ≡ 0) return 1;
    }
    return 0;
  }
```

17. ***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;
- Click on the link *Create Citation Report*;
- The h-index is showed at the top of the page.

To calculate the h-index in batch mode, we downloaded a file with the data by clicking on the button *Export Data: Save To Text File* and selecting *Records from ...* that saves the same data, with limit of 500 records, where each field it the record is separated by the sign stored in the macro `CSV_SEP`. The files were saved with a ".csv" extension inside `DATA_DIRECTORY`.

```
#define DATA_DIRECTORY "data/"    /* directory containing all data */
#define H_EXT ".csv"    /* file used to calculate h-index extension */
```

⟨ Calculate h index 17 ⟩ ≡

```
for (i = 0; i < A; i++) {    /* for each author */
  int h = 0;    /* temporary h-index */
  ⟨ Process csv file 18 ⟩
  authors[i]→h = h;
}
```

This code is used in section 2.

18. ⟨ Process csv file 18 ⟩ ≡

```
strncpy(buffer, DATA_DIRECTORY, sizeof (DATA_DIRECTORY));
strncat(buffer, authors[i]→researchid, sizeof (authors[i]→researchid));
strncat(buffer, H_EXT, sizeof (H_EXT));
fn = buffer;
fp = fopen(fn, "r"); if (fp) { while ( fgets ( line , sizeof ( line ) , fp ) ≠ Λ )
{
  ⟨ Parse the line counting citations 19 ⟩
}
fclose(fp); }
else {
  perror(fn);
  exit(−2);
}
```

This code is used in section 17.

19. The head of the citations file contains some lines that must be ignored. These lines contains the words "AUTHOR", "Article Group for:", "Timespan=All" and "T̈itle" in the beginning of the line (ignore double quotes without escape). There is also an empty line or a line that starts with a new line special command. Surviving to these rules, the line is a paper record of an author, along with collaborators, and is parsed to count the number of citations.

```
⟨ Parse the line counting citations 19 ⟩ ≡
  if ( strstr ( line , "AUTHOR" ) ≠ Λ ∨ strstr ( line , "IDENTIFICADORES␣DE␣AUTOR:" ) ≠ Λ )
  {
    continue;
  }
  else if ( strstr ( line , "Article␣Group␣for:" ) ≠ Λ )
  {
    continue;
  }
  else if ( strstr ( line , "Timespan=All" ) ≠ Λ ∨ strstr ( line , "Tempo␣estipulado=Todos␣os␣anos" )
      ≠ Λ )
  {
    continue;
  }
  else if ( strstr ( line , "\"Title\"," ) ≠ Λ ∨ strstr ( line , "\"Autores\"," ) ≠ Λ )
  {
    continue;
  }
  else if ( line [0] ≡ '\n' )
  {     /* start with new line */
    continue;
  }
  else {
    ⟨ Count the citations and check if the h-index was found 20 ⟩
  }
```

This code is used in section 18.

20. To count the citations and check if the h-index exists, the line is tokenized generating fields to be evaluated. The marks to divide the line are set to `CSV_SEP` macro. The first `SKIP_FIELDS` fields are ignored because contain author's name, paper's name, journal's name and volume and information that is not citation. Citations start after `SKIP_FIELDS` fields and are classified by year starting in 1900, so the first citations' numbers normally are zero. In the citations region, they are accumulated until the last year is found. If their summation is lesser than a counter of papers, the counter is decremented, and the h-index was found. This value is assigned to a field h the author structure to be written in the end of the program.

```
#define CSV_SEP ",\"\n"
#define SKIP_FIELDS 30
```

⟨ Count the citations and check if the h-index was found 20 ⟩ ≡

```
{ int c = 0;
j = 0; p = strtok ( line , CSV_SEP ) ;
while (p ≠ Λ) {
  if (j > SKIP_FIELDS) {
    c += atoi(p);
  }
  p = strtok(Λ, CSV_SEP);
  j++;
}
if (h > c) {      /* found h */
  h--;
  break;      /* stop reading file */
}
h++; }
```

This code is used in section 19.

21. ***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`. The files were saved with a ".tsv" extension inside `DATA_DIRECTORY`. All authors' files are parsed and K-index is calculated.

22. ⟨ Calculate K index 22 ⟩ ≡

```
for (i = 0; i < A; i++) {      /* for each author */
  ⟨ Process tsv file 23 ⟩
}
```

This code is used in section 2.

23. 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 23 ⟩ ≡

```
strncpy(buffer, DATA_DIRECTORY, sizeof (DATA_DIRECTORY));
strncat(buffer, authors[i]→researchid, sizeof (authors[i]→researchid));
strncat(buffer, K_EXT, sizeof (K_EXT));
fn = buffer;
fp = fopen(fn, "r"); if (fp) { int k = 1;      /* temporary K-index */
while ( fgets ( line , sizeof ( line ) , fp ) ≠ Λ )
{
  ⟨ Parse the line counting citings 24 ⟩
}
fclose(fp); }
else {
  perror(fn);
  exit(−2);
}
```

This code is used in section 22.

24. 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 24 ⟩ ≡
  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 25 ⟩
  }
```

This code is used in section 23.

25. `K_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 `K_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 K_SKIP 7     /* number of fields that can be skipped with safety */
⟨ Find the citings and check if the K-index was found 25 ⟩ ≡
  { int c = 0;
  j = 0; p = strtok ( line , TSV_SEP ) ;
  while (p ≠ Λ) {
    if (j > K_SKIP) {
      enqueue(p);
    }
    j++;
    p = strtok(Λ, TSV_SEP);
  }
  for (j = 0; j < 3; j++) {
    p = dequeue( );
    if (p ≡ Λ) queue_panic( );
  }
  c = atoi(p);
  queue_reset( );
  if (k > c) {     /* found k */
    k--;
    authors[i]¬k = k;
    break;
  }
  k++; }
```

This code is used in section 24.

26. Queue. A humble queue is implemented to store few pointers using FIFO policy. The queue is composed by an array of pointers and an index *idx* that marks the top element of the queue.

```
⟨ Internal variables 5 ⟩ +≡
  static char *stack[64];
  static int idx = 0;
```

27. Elements are inserted at the top of the queue by invoking *enqueue* and using **char** *p as parameter. The index *idx* is incremented to the number of elements in the queue and *idx* − 1 is the top of the queue.

```
⟨ Static functions 3 ⟩ +≡
  static void enqueue(char *p)
  {
    if (p ≡ Λ) return;
    stack[idx++] = p;
  }
```

28. Elements from the top of the queue are removed by *dequeue* function. If there is no element in the queue, Λ is returned.

```
⟨ Static functions 3 ⟩ +≡
  static char *dequeue( )
  {
    if (idx ≤ 0) return Λ;
    else return stack[--idx];
  }
```

29. When for some reason, an error related with the queue occurs *queue_panic* may be invoked, exiting from the execution program.

```
#define ERR_QUEUE -#1
⟨ Static functions 3 ⟩ +≡
  static void queue_panic( )
  {
    fprintf(stderr, "Queue␣is␣very␣empty.\n");
    exit(ERR_QUEUE);
  }
```

30. To reset the queue, *idx* is reseted to zero.

```
⟨ Static functions 3 ⟩ +≡
  static void queue_reset( )
  {
    idx = 0;
  }
```

31. 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 31 ⟩ ≡
  for (i = 1; i < A; i++) {
    aut = authors[i];
    for (j = i − 1; j ≥ 0 ∧ aut¬k > authors[j]¬k; j−−) {
      authors[j + 1] = authors[j];
    }
    authors[j + 1] = aut;
  }
```

This code is used in section 2.

32. Output. The results are written as a table in markdown format. A space is needed between the bars and the content.

```
⟨ Write results to a file 32 ⟩ ≡
  fn = "rank.md";
  fp = fopen(fn, "w");
  if (¬fp) {
    perror(fn);
    exit(−4);
  }
  fprintf(fp, "| N | Author | h | K |\n");
  fprintf(fp, "|---|--------|---|---|\n");
  for (i = 0; i < A; i++) {
    fprintf(fp, "| %d | [%s](%s) | %d | %d |\n", i + 1, authors[i]→name, authors[i]→url, authors[i]→h,
        authors[i]→k);
  }
  fclose(fp);
  fprintf(stderr, "* Wrote \"%s\"\n", fn);
```

This code is used in section 2.

33. 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 33 ⟩ ≡
  fn = "table.tex";
  fp = fopen(fn, "w");
  if (¬fp) {
    perror(fn);
    exit(−8);
  }
  fprintf(fp, "\\begin{tabular}{cccc} \\\\ \\hline\n");
  fprintf(fp, "\\bf N &\\bf Author &\\bg h &\\bf K \\\\ \\hline\n");
  for (i = 0; i < 12; i++) {
    fprintf(fp, " %d & %s & %d & %d \\\\\n", i + 1, authors[i]→name, authors[i]→h, authors[i]→k);
  }
  fprintf(fp, "\\hline\\end{tabular}\n");
  fclose(fp);
  fprintf(stderr, "* Wrote \"%s\"\n", fn);
```

This code is used in section 2.

34. Memory allocated for the array of pointers *authors* is freed. As the memory deallocation is the last task to be executed, a simple usage notification is appended before the task.

```
⟨ Free up memory 34 ⟩ ≡
  if (¬verbose) fprintf(stderr, "\ninfo: run \"%s -v\" to print more information.\n", argv[0]);
  for (i = 0; i < A; i++) free(authors[i]);
  free(authors);
```

This code is used in section 2.

35. Index.

K-NOBEL