1. Introduction. K-NOBEL is a project to try to predict the next winners of Nobel prize of physics using K-index as parameter of comparison. Another parameter, h-index, is used to evaluate the error threshold, since h-index is used by Web of Science as one of the parameters to predict the winners of Nobel prize.

The program has the following structure:

```
#include <stdio.h>
#include <stdlib.h>
  ⟨ Include files 6 ⟩
  ⟨ Data structures 3 ⟩
  ⟨ Internal variables 4 ⟩
  ⟨ Static functions 18 ⟩
```

2.
```
int main(int argc, char **argv)
  {
    ⟨ Load authors information 5 ⟩
    ⟨ Calculate h index 8 ⟩
    ⟨ Calculate K index 13 ⟩
    ⟨ Write results to a file 22 ⟩
    ⟨ Free up memory 23 ⟩
    return 0;
  }
```

3. Authors. Information about research authors were stored into `index.csv` file. They consist into name, Web of Science or Google Scolar or Publons research id and a link to a page containing more information about the citations. Not all authors have research 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 MAX_STR_LEN 256
⟨ Data structures 3 ⟩ ≡
  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.

4. An array of structs 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 so many authors as collaborators. 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 imposes an attention to details along the program to not forget to restart the counters, for example.

```
#define MAX_LINE_LEN 1 ≪ 16
⟨ Internal variables 4 ⟩ ≡
  static struct author **authors;    /* store authors' info */
  static char *fn, *p;    /* file name and generic pointer */
  static FILE *fp;    /* file pointer */
  static char buffer[MAX_STR_LEN];    /* buffer to carry strings */
  static char
  line [MAX_LINE_LEN] ;    /* store file lines */
  static int A = 0;    /* number of authors */
  static int i = 0, j = 0;    /* general-purpose counters */
```

See also section 17.

This code is used in section 1.

5. Authors basic information was picked at the Web of Science page, more specifically at `https://hcr.clarivate.com/#categories%3Dphysics` that is the page of highly 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 5 ⟩ ≡
  fn = "data/authors.idx";
  fp = fopen(fn, "r"); if (fp) { while ( fgets ( line , sizeof ( line ) , fp ) ≠ Λ ) { if ( line [0] ≡ '#' )
     /* comments */
  continue;
  A++;      /* reallocate the array of authors struct with to pointer elements */
  authors = (struct author **) realloc(authors, A * sizeof(struct author *));
  ⟨ Begin to fill authors structure 7 ⟩
  } fclose(fp); }
  else {
     perror(fn);
     exit(−2);
  }
```

This code is used in section 2.

6. ⟨ Include files 6 ⟩ ≡

```
#include <string.h>
```

This code is used in section 1.

7. The fields are separated by semicolon inside *authors.idx*, a record in the file is 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 containing information about author's publications. A structure is loades 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 7 ⟩ ≡
  struct author *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++;
  }
  authors[A − 1] = aut;
```

This code is used in section 5.

8. *h*-index. The number of papers 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 appears on the top of the page.

To calculate in batch mode, we downloaded a file with the data to calculate the *h* 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 is separated by comma that is represented by the macro `CSV_SEP`. The files were saved with a ".csv" extension inside `DATA_DIRECTORY`. All authors' files are traversed, parsed and *h*-index is calculated. The results are saved in a file.

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

```
⟨ Calculate h index 8 ⟩ ≡
  for (i = 0; i < A; i++) {    /* for each author */
    int h = 0;    /* temporary h-index */
    ⟨ Process csv file 9 ⟩
    authors[i]→h = h;
  }
```

This code is used in section 2.

9. ⟨ Process csv file 9 ⟩ ≡

```
  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 10 ⟩
  }
  fclose(fp); }
  else {
    perror(fn);
    exit(−2);
  }
```

This code is used in section 8.

10. The head of the citations file contains some line that must be ignored. These lines contains the words "AUTHOR", "Article Group for:", "Timespan=All" and "Title" 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. Passing these rules, the line is a paper record of the author and is parsed to count the number of citations.

```
⟨ Parse the line counting citations 10 ⟩ ≡
  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 11 ⟩
  }
```

This code is used in section 9.

11. To count the citations and check if the *h*-index was found, 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` 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 counter is the *h*-index. This value is assigned to a field in a structure called author to be written at the end of the program.

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

⟨ Count the citations and check if the h-index was found 11 ⟩ ≡

```
  { 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 10.

12. 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 is as follows:

- ⋆ 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 represented by the macro `TSV_SEP`. The files were saved with a ".tsv" extension inside `DATA_DIRECTORY`. All authors' files are traversed, parsed and K-index is calculated. The results are saved in a file.

13. ⟨ Calculate K index 13 ⟩ ≡

```
for (i = 0; i < A; i++) {      /* for each author */
  ⟨ Process tsv file 14 ⟩
}
```

This code is used in section 2.

14. 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 14 ⟩ ≡

```
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 15 ⟩
}
fclose(fp); }
else {
  perror(fn);
  exit(−2);
}
```

This code is used in section 13.

15. The file with citings has few lines to ignore, basically it's 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 15 ⟩ ≡
  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 16 ⟩
  }
```

This code is used in section 14.

16. `K_SKIP` represents the fields to be skiped 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 skiped with safety ∗/
⟨ Find the citings and check if the K-index was found 16 ⟩ ≡
  { 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 15.

17. 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 4 ⟩ +≡
  static char *stack[64];
  static int idx = 0;
```

18. 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 18 ⟩ ≡
  static void enqueue(char *p)
  {
    if (p ≡ Λ) return;
    stack[idx++] = p;
  }
```

See also sections 19, 20, and 21.

This code is used in section 1.

19. 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 18 ⟩ +≡
  static char *dequeue()
  {
    if (idx ≤ 0) return Λ;
    else return stack[--idx];
  }
```

20. 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 18 ⟩ +≡
  static void queue_panic()
  {
    fprintf(stderr, "Queue␣is␣very␣empty.\n");
    exit(ERR_QUEUE);
  }
```

21. To reset the queue, *idx* is zeroed.

```
⟨ Static functions 18 ⟩ +≡
  static void queue_reset()
  {
    idx = 0;
  }
```

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

```
⟨ Write results to a file 22 ⟩ ≡
  fn = "k-nobel.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.

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

```
⟨ Free up memory 23 ⟩ ≡
  for (i = 0; i < A; i++) free(authors[i]);
  free(authors);
```

This code is used in section 2.

24. Index.

K-NOBEL