Problem-solving with complex problems

FROM PROPLEM TO PROGRAM: DECOMPOSING A COMPLEX PROBLEM INTO SUB-PROBLEMS

The “divide and conquer” paradigm

- A complex problem can be tackled (and solved) successfully through a "divide and conquer" strategy:
  - decomposition into simpler sub-problems
  - solution of elementary problems
  - recombination of elementary solutions in the solution of the starting problem.
- The sub-problems are not necessarily of the same nature
- The decomposition is not always univocal, that is, there are various ways of decomposing a problem (and its solution)
- The decomposition into sub-problems takes place in terms of:
  - data structure
  - algorithm

Division-Solution-Recombination

- Division into sub-problems:
  - it can be elementary and immediate: just identify the sub-problems
  - it can be complex and require the generation of “ad hoc” data structures for the sub-problems
- Solution of sub-problems:
  - they can be independent (you can solve them in any order)
  - or they require a certain sequence, since the results of a sub-problem are input for other sub-problems
- Recombination of solutions:
  - it can be elementary and immediate: the global solution is already available once the sub-problems have been solved
  - it is necessary to elaborate the (partial) results of the sub-problems, to obtain the final result of the overall problem

Decomposition of data structures

- The data structure can be broken down in various ways and / or levels:
  - no decomposition: the decomposition is only of the algorithmic type. The same (unique) data structure of the original problem is manipulated in all sub-problems
  - data partitioning: the data structure is divided into (homogeneous) portions associated with single sub-problems (e.g. vector or matrix broken down into sections)
  - “ad hoc” data structures for single sub-problems: this is the most general scheme.

Decomposition of algorithms

- An algorithm can be considered broken down into sections, corresponding to:
  - conditional constructs
  - (single iterations of) iterative constructs
  - (calls to) functions
- The individual sections can:
  - manipulate global data
  - receive input data and return results

Independence and/or relation

- Sections / parts can be:
  - independent, if they operate on different data
    - the order of resolution of the sub-problems can be arbitrary
    - no intermediate results are needed between the subproblems

    Example: calculation of the average value, for each row of a matrix
  - Related, if each subproblem depends on the others:
    - the order of resolution of the sub-problems is not arbitrary
    - it is necessary to provide (at the data structure level) the storage of intermediate results

Import-export

- Formulation: two text files are given, containing information on the operations of an import / export company, which has commercial relations with companies belonging to several states
  - a first file (hereinafter referred to as F0) contains the list of states and companies with which commercial relations exist
  - a second file (hereinafter referred to as F1) contains information on a certain number of commercial transactions (with one of the companies listed in F0)

Input data

- File F0 contains the list of states and companies:
  - in the first line two integers <nst> <nsoc> (separated by spaces): total number of states and companies, both surely lesser than 100.
  - in the <nst> subsequent lines the states are listed, one per line, according to the format:

    <state_code> <state_name>

    - <state_code> is an integer code between 0 and 99 (each state has a unique code, it is not guaranteed that the states are listed in ascending order of code)
    - <state_name> is the state name, without spaces
  - The companies are listed in the final <nsoc> lines of the file, as follows:

    <state_code> <company_name>

    - <state_code> is the code of the state to which the company belongs
    - <company_name> is the name of the company

Input data

- F1 contains information on a set of commercial transactions, each of which is represented, on a line of the file, according to the format:

  <company_name> <amount> <date>

  - <company_name> is the name of the company
  - <amount> (integer with sign) represents the amount of the transaction (negative for import, positive for export)
  - <date> is the date of the transaction (dd/mm/yyyy format)

Example

F0

```
5 10
4 Germany
3 France
0 United_States
2 Japan
1 Korea
1 Kia
4 BMW
4 Mercedes
3 Peugeot
3 Citroen
2 Honda
2 Mitsubishi
0 Chrisler
0 General_Motors
0 Chevrolet
```

F1

```
Mitsubishi 101 21/01/2004
Chrisler -30 01/02/2004
General_Motors -2000 10/02/2004
Citroen -700 24/11/2004
Chrisler -367 12/12/2004
...
General_Motors -1400 14/12/2004
Chevrolet -600 16/12/2004
```

Requested processing

- Read F0 and F1, whose names are received as arguments on the command line
- Calculate, and print on video:
  - the total number of transactions (NT) and the overall balances of import and export transactions
    - SI: sum of the amounts related to imports
    - SE: sum of the amounts related to exports
  - for each state, the overall import-export trade balance (sum of the amounts related to all companies belonging to the state)
  - the state for which the import balance is maximum and the state for which the export balance is maximum.

Data structures

- Two tables contain the data read from the files:
  - states table: contains the list of states, identified by code and name. Since the codes are between 0 and 99, it is advisable to use the code as an index in a vector, to allow the conversion from code to name with direct access
  - table of companies: contains the list of companies, with each element reporting the name of the company and the code of the state it belongs to. The table allows you to derive the state code from the name of a company

Data structures

- Variables (scalars) for the required statistics:
  - summation of imports
  - summation of exports
- A vector for the import-export balance of the individual states. This balance can be added (as an additional field) to the states table
- We do NOT need a data structure to store the list of transactions, which can be read and manipulated one at a time.

Representation of the data structure

States table

0	0	“United_States”
1	0	“Korea”
2	0	“Japan”
3	0	“France”
4	0	“Germany”

Table of companies

1	“Kia”
4	“BMW”
4	“Mercedes”
	…
	…
0	“Chevrolet”

Representation of the data structure

States table

0	0	“United_States”
1	0	“Korea”
2	0	“Japan”
3	0	“France”
4	0	“Germany”

State’s balance, initially zero

Table of companies

1	“Kia”
4	“BMW”
4	“Mercedes”
	…
	…
0	“Chevrolet”

Algorithm

- Load tables of states and table of companies from F0, initialize states' balances to 0
- Read one transaction at a time from F1
  - Update number of transactions
  - Update SI and SE, based on filter on the amount:
    - Negative amounts –> SI
    - Positive amount –> SE
  - Find state corresponding to name of company –> Update state's balance
- Find states with maximum positive balance (export) and maximum negative balance (import)
  - For negative balances it is a minimum search problem!

Decomposition in sub-problems

- *Load input data*: read F0 and load information in tables of states and tables of companies
- *Selection sub-problem:* identify the code of a state a company belongs to, starting from the company’s name
- *Selection sub-problem:* compute import/export balance of each state
- *Optimization sub-problem (numerical data):* identify states with maximum import/export balance

Solution

```
#include <stdio.h>
#define MAXC 101
#define MAXSTATES 100
#define MAXSOC 100
/* type struct for states and companies */
typedef struct {
  char name[MAXC]; int balance;
} t_state;
typedef struct {
  char name[MAXC]; int code_state;
} t_society;
/* function prototype */
int searchCodeState(
  t_society tab_soc[], int n, char name[]
);
```

```
int main (int argc, char *argv[]) {
  FILE *fp;
  int nst, nsoc, code, amount,
      i, nt, si, se, maxi, maxe;
  char name[MAXC];
  t_state tab_st[MAXSTATES];
  t_society tab_soc[MAXSOC];
  /* read header of file F0 */
  fp = fopen(argv[1],"r");
  fscanf(fp,“%d%d”,&nst,&nsoc);
  /* Initialize names with empty strings  */
  for (i=0; i<MAXSTATES; i++)
    strcpy(tab_st[i].name,””);
```

Solution

```
#include <stdio.h>
#define MAXC 101
#define MAXSTATES 100
#define MAXSOC 100
/* type struct for stat
typedef struct {
  char name[MAXC]; int
} t_state;
typedef struct {
  char name[MAXC]; int code_state;
} t_society;
/* function prototype */
int searchCodeState(
  t_society tab_soc[], int n, char name[]
);
```

```
int main (int argc, char *argv[]) {
  FILE *fp;
  int nst, nsoc, code, amount,
      i, nt, si, se, maxi, maxe;

  /* read                  0 */
  fp = fopen(              ");
  fscanf(fp,“%d,      &nst,&nsoc);
  /* Initialize names with empty strings  */
  for (i=0; i<MAXSTATES; i++)
    strcpy(tab_st[i].name,””);
```

Names of the states are initialized as empty strings. The codes corresponding to an empty string will not be assigned after the reading

Solution

```
/* read table of states*/
for (i=0; i<nst; i++) {
  fscanf(fp,"%d%s", &code,name);
  strcpy(tab_st[code].name,name);
  tab_st[code].balance = 0;
}
/* read table of companies*/
for (i=0; i<nsoc; i++) {
  fscanf(fp,"%d%s", &code, name);
  strcpy(tab_soc[i].name,name);
  tab_soc[i].code_state = code;
}
fclose(fp);
```

```
/* transactions */
fp = fopen(argv[2],"r");
nt = si = se = 0;
while (fscanf(fp,"%s%d%*s",name,&amount) != EOF) {
  nt++;
  if (amount > 0)
    se += amount;
  else
    si += amount;
  code=searchCodeState(tab_soc,nsoc,name);
  tab_st[code].balance += amount;
}
fclose(fp);
```

Solution

```
/* read table of states*/
for (i=0; i<nst; i++) {
  fscanf(fp,"%d%s", &code,name);
  strcpy(tab_st[code].name,name);
  tab_st[code].balance = 0;
}
/* read table of companies*/
for (i=0; i<nsoc; i++) {
  fscanf(fp,"%d%s", &code, name);
  strcpy(tab_soc[i].name,name);
  tab_soc[i].code_state = code;
}
fclose(fp);
```

```
/* transactions */
fp = fopen(argv[2],"r");
nt = si = se = 0;
while (fscanf(fp,"%s%d%*s",name,&amount) != EOF) {
  nt++;
  if (amount > 0)
    se += amount;
}
fclose(fp);
```

Index-value relation
The information about states are stored in the element that has the code as the index. Balance is initialized with 0

Solution

```
/* read table of states*/
for (i=0; i<nst; i++) {
  fscanf(fp,"%d%s", &code,name);
  strcpy(tab_st[code].name,name);
  tab_st[code].balance = 0;
}
/* read table of companies*/
for (i=0; i<nsoc; i++) {
  fscanf(fp,"%d%s", &code, name);
  strcpy(tab_soc[i].name,name);
  tab_soc[i].code_state = code;
}
fclose(fp);
```

Array as a container
The information about companies is progressively stored in array, with increasing indexes

```
/* transactions */
fp = fopen(argv[2],"r");
                                                ) {

    se += amount;
  else
    si += amount;
  code=searchCodeState(tab_soc,nsoc,name);
  tab_st[code].balance += amount;
}
fclose(fp);
```

Solution

```
/* read table of states*/
for (i=0; i<nst; i++) {
  fscanf(fp,"%d%s", &code,name);
  strcpy(tab_st[code].name,name);
  tab_st[code].balance = 0;
}
/* read table of companies*/
for (i=0; i<nsoc; i++) {
```

```
fclose(fp);
```

Transactions
They are not stored in a vector, but processed as they are read, one by one:
- summations
- update of state’s balance

```
/* transactions */
fp = fopen(argv[2],"r");
nt = si = se = 0;
while (fscanf(fp,"%s%d%*s",name,&amount) != EOF) {
  nt++;
  if (amount > 0)
    se += amount;
  else
    si += amount;
  code=searchCodeState(tab_soc,nsoc,name);
  tab_st[code].balance += amount;
}
fclose(fp);
```

Solution

```
/* read table of states*/
for (i=0; i<nst; i++) {
  fscanf(fp,"%d%s", &code,name);
}
/* read table of companies*/
for (i=0; i<nsoc; i++) {
  fscanf(fp,"%d%s", &code, name);
  strcpy(tab_soc[i].name,name);
  tab_soc[i].code_state = code;
}
fclose(fp);
```

Filter on data:
If amount > 0 -> export
otherwise   -> import

```
/* transactions */
fp = fopen(argv[2],"r");
nt = si = se = 0;
while (fscanf(fp,"%s%d%*s",name,&amount) != EOF) {
  nt++;
  if (amount > 0)
    se += amount;
  else
    si += amount;
  code=searchCodeState(tab_soc,nsoc,name);
  tab_st[code].balance += amount;
}
fclose(fp);
```

Solution

```
/* read table of states*/
for (i=0; i<nst; i++) {
  fscanf(fp,"%d%s", &code,name);
  strcpy(tab_st[code].name,name);

}
/* read table of companies*/
for (i=0; i<nsoc; i++) {
  fscanf(fp,"%d%s", &code, name);
  strcpy(tab_soc[i].name,name);
  tab_soc[i].code_state = code;
}
fclose(fp);
```

Name-code conversion, implemented separately (search sub-problem)

```
/* transactions */
fp = fopen(argv[2],"r");
nt = si = se = 0;
while (fscanf(fp,"%s%d%*s",name,&amount) != EOF) {
  nt++;
  if (amount > 0)
    se += amount;
  else
    si += amount;
  code=searchCodeState(tab_soc,nsoc,name);
  tab_st[code].balance += amount;
}
fclose(fp);
```

Solution

```
/* compute states with max. imp.-exp. */
maxi = maxe = -1;
for (i=0; i<MAXSTATES; i++)
  if (tab_st[i].name[0] != ‘\0’) {
    if (tab_st[i].balance > 0) {
      if (maxe<0 || tab_st[i].balance
                  > tab_st[maxe].balance)
        maxe = i;
      } else if (tab_st[i].balance < 0) {
        if (maxi<0 || tab_st[i].balance
                    < tab_st[maxi].balance)
          maxi = i;
      }
    }
```

```
  /* print global statistics*/
  printf("Num. transactions: %d\n", nt);
  printf("Total imp.-exp.: %d %d\n\n", si, se);
  /* print state statistics */
  for (i=0; i<nst; i++)
    if (tab_st[i].name[0] != ‘\0’)
      printf("STATE: %-30s BALANCE: %8d\n",
       tab_st[i].name,tab_st[i].balance);
  /* print max. import-export */
  if (maxe>=0) printf(“Max. exp -> %s: %d\n",
    tab_st[maxe].name,tab_st[maxe].balance);
  if (maxi>=0) printf(“Max. imp -> %s: %d\n",
    tab_st[maxi].name,tab_st[maxi].balance);
}
```

Solution

```
/* compute states with max. imp.-exp. */
maxi = maxe = -1;
for (i=0; i<MAXSTATES; i++)
  if (tab_st[i].name[0] != '\0') {
    if (tab_st[i].balance > 0) {
      if (maxe<0 || tab_st[i].balance
                  > tab_st[maxe].balance)
        maxe = i;
      } else if (tab_st[i].balance < 0) {
        if (maxi<0 || tab_st[i].balance
                    < tab_st[maxi].balance)
          maxi = i;
      }
    }
```

Search (index of) state with maximum export

```
   /* print global statistics*/
   printf("Num. transactions: %d\n", nt);
   printf("Total imp.-exp.: %d %d\n\n", si, se);
   /* print state statistics */
   for (i=0; i<nst; i++)

        tab_st[i].name,tab_st[i].balance);
   /* print max. import-export */
   if (maxe>=0) printf("Max. exp -> %s: %d\n",
     tab_st[maxe].name,tab_st[maxe].balance);
   if (maxi>=0) printf("Max. imp -> %s: %d\n",
     tab_st[maxi].name,tab_st[maxi].balance);
}
```

Solution

```
/* compute states with max. imp.-exp. */
maxi = maxe = -1;
for (i=0; i<MAXSTATES; i++)
 if (tab_st[i].name[0] != ‘\0’) {
  if (tab_st[i].balance > 0) {
   if (maxe<0 || tab_st[i].balance
         > tab_st[maxe].balance)
    maxe = i;
   } else if (tab_st[i].balance < 0) {
    if (maxi<0 || tab_st[i].balance
          < tab_st[maxi].balance)
     maxi = i;
   }
  }
```

```
  /* print global statistics*/
  printf("Num. transactions: %d\n", nt);
  printf("Total imp.-exp.: %d %d\n\n", si, se);
  /* print state statistics */
  for (i=0; i<nst; i++)
    if (tab_st[i].name[0] != ‘\0’)
      printf("STATE: %-30s BALANCE: %8d\n",
       tab_st[i].name,tab_st[i].balance);
  /* print max. import-export */
  if (r
  if (maxi>=0) printf(“Max. imp -> %s: %d\n",
    tab_st[maxi].name,tab_st[maxi].balance);
}
```

Search (index of) state with maximum import

Solution

Print final results

```
/* com                                    /
maxi =
for (i
  if (tab_st[i].name[0] != ‘\0’) {
    if (tab_st[i].balance > 0) {
      if (maxe<0 || tab_st[i].balance
                  > tab_st[maxe].balance)
        maxe = i;
      } else if (tab_st[i].balance < 0) {
        if (maxi<0 || tab_st[i].balance
                    < tab_st[maxi].balance)
          maxi = i;
      }
    }
```

```
/* print global statistics*/
printf("Num. transactions: %d\n", nt);
printf("Total imp.-exp.: %d %d\n\n", si, se);
/* print state statistics */
for (i=0; i<nst; i++)
  if (tab_st[i].name[0] != ‘\0’)
    printf("STATE: %-30s BALANCE: %8d\n",
     tab_st[i].name,tab_st[i].balance);
/* print max. import-export */
if (maxe>=0) printf(“Max. exp -> %s: %d\n",
  tab_st[maxe].name,tab_st[maxe].balance);
if (maxi>=0) printf(“Max. imp -> %s: %d\n",
  tab_st[maxi].name,tab_st[maxi].balance);
}
```

Solution

```
int searchCodeState(t_society tab_soc[],
                      int n, char name[]) {
  int i;
  for (i=0; i<n; i++) {
    if (strcmp(tab_soc[i].name,name)==0)
      return (tab_soc[i].code_state);
  }
  return -1;  // if not found, returns -1
}
```

Solution

```
int searchCodeState(t_society tab_soc[],
                      int n, char name[]) {
  int i;
  for (i=0; i<n; i++) {
    if (strcmp(tab_soc[i].name,name)==0)
      return (tab_soc[i].code_state);
  }
  return -1;
}
```

Subproblem of search in table (corrispondence string-number)

Image encoding

- Formulation: a "bitmap" type image is given, represented as a matrix of 600x800 pixels (600 rows, 800 columns)
- The pixels are encoded with a depth of 16 bits, i.e. each pixel is associated with a color, represented by a numerical value between 0 and 2^{16}-1
- Each pixel can be univocally identified by two coordinates **(r,c)**, row and column indexes **$(0 \leq r < 600, 0 \leq c < 800)$**

Encoding in dots and rectangles

- The image, originally stored in a binary file, was subsequently converted into a text file according to the following format
  - The image is considered to be made up of the superimposition of rectangles and points
  - one point corresponds to one pixel
  - a rectangle is the set of all and only the points having coordinates of row r and column c, such that:

    $\mathbf{r0 \leq r \leq r1}$, $\mathbf{c0 \leq c \leq c1}$

    where **r0, r1, c0 e c1** are integer values

Encoding in points and rectangles

- To each rectangle and/or point are associated:
  - A color (integer in the range 0 to 2^{16}-1)
  - A level (integer in the range 0 to 255)
- Each point or rectangle of a higher level must be considered overlapping the lower levels.
- The color of a pixel with coordinates (r, c) is determined by the highest level point or rectangle covering (r, c)

Encoding in dots and rectangles

- Each point is represented in the file by a line

  p <r> <c> <color> <level>

  - **<r> and <c>** are the coordinates of the point
  - **<color> and <level>** (integers) represent the color and level of the point, respectively

- Each rectangle is represented by a line

  r <r0> <r1> <c0> <c1> <color> <level>

  - **<r0>, <r1>, <c0>** and **<c1>** are the limits for the coordinates of the points within the rectangle
  - **<color> and <level>** (integer) are the color and level of all the points belonging to the rectangle

Requested processing

- Read the file, whose name is received as the first argument on the command line (argv [1]). Assume that the lines are <=100.
- Eliminate invisible points and rectangles (totally covered by higher level points and / or rectangles) and rewrite the image (same format) on a second file, with the name received in argv [2]
- Calculate the number of pixels of color equal to a reference pixel P0, whose r,c coordinates are received in argv [3] and argv [4], respectively
- Say which are the rectangles and / or points (either visible or invisible) that cover P0

Data structure

- A table, for the list of rectangles (and points):
  - vector of struct, each representing a rectangle, with coordinates, color and level. The points can be assimilated to rectangles of minimum size
- Pixel matrix:
  - for each pixel you can store
    - level and color (determined based on the highest level rectangle covering the pixel)
    - the index of the highest level rectangle covering the pixel (from which level and color can be obtained)
  - The following solution adopts the second option

Representation of data structure

Matrix of pixels

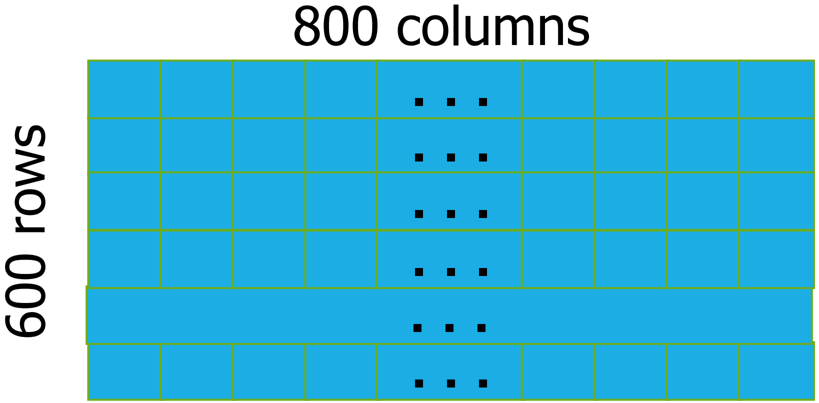
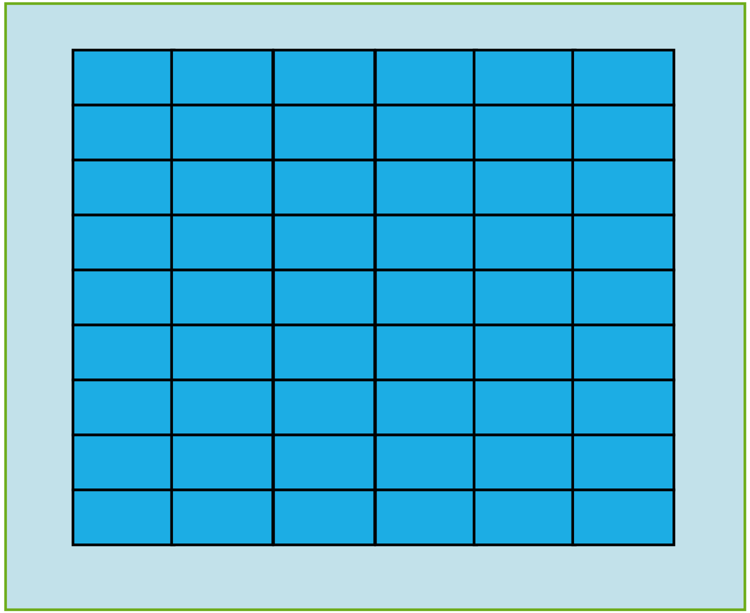


Table of rectangles



Representation of data structure

Matrix of pixels

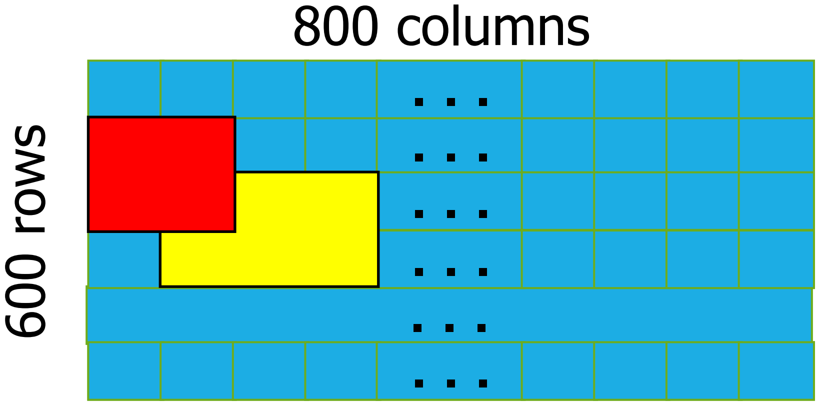
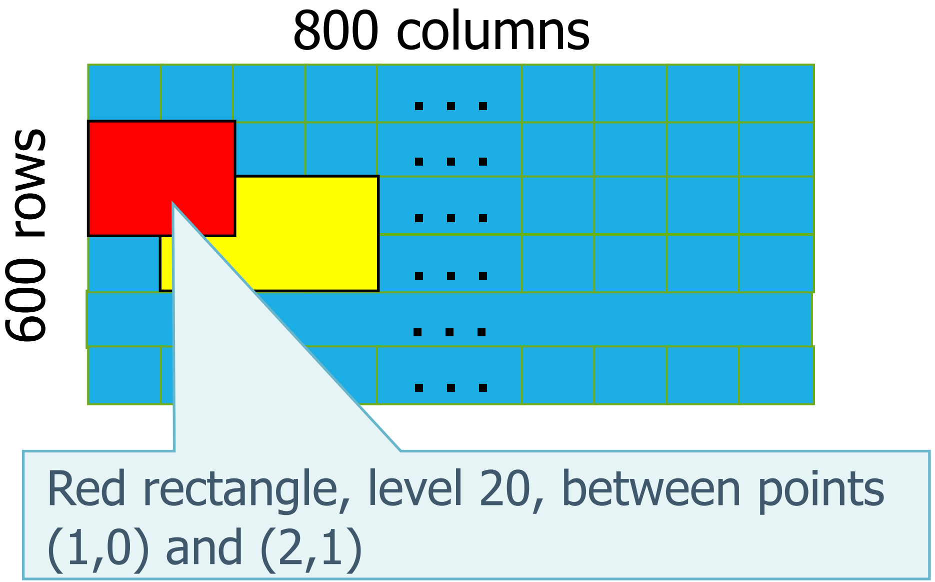


Table of rectangles

1	2	0	1	20	🟥
2	3	1	3	7	🟨

Representation of data structure

Matrix of pixels



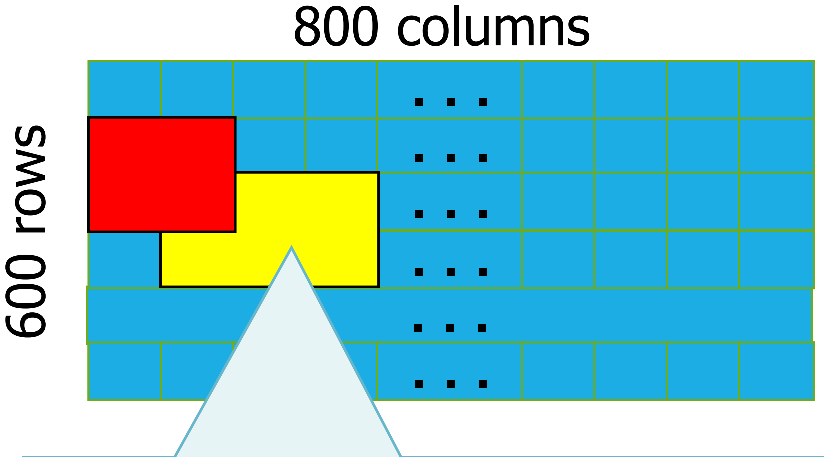
Red rectangle, level 20, between points (1,0) and (2,1)

Table of rectangles

1	2	0	1	20	🟥
2	3	1	3	7	🟨

Representation of data structure

Matrix of pixels



Yellow rectangle, of level 7, between points (2,1) and (3,3)

Table of rectangles

1	2	0	1	20	🟥
2	3	1	3	7	🟨

Algorithm

- Reading the image and loading the first table
- Loading the pixel matrix:
  - it can be seen as a maximum search problem for each of the affected pixels
  - for each point of each rectangle it is determined whether the point is visible (it has the maximum level for the corresponding pixel) or not
- Filter on visible rectangles:
  - it is a selection problem, in which, for a given rectangle, all points must be examined

Algorithm

- Counting pixels with color equal to **p0:**
  - Once the color of p0 is determined, it is a selection problem, applied to the pixel matrix
- Computing rectangles that cover **p0:**
  - It is a selection problem, applied to the list of rectangles

Decomposition into sub-problems

- The subproblems are naturally derived from the individual required results:
    - Load rectangles, load matrix
    - Filter rectangles:
        - selection on the vector of rectangles
        - as a subproblem it considers the points belonging to a given rectangle
    - Counting pixels of color equal to p0:
        - search on the pixel matrix
    - Calculation of rectangles covering p0:
        - selection on rectangles vector

Solution

```
#include <stdio.h>
#define NR 600
#define NC 800
#define MAXC 100
#define MAX_RECT 100

/* type struct for rectangles */
typedef struct {
  int r0, r1, c0, c1;
  int level;
  int color;
} t_rectangle;
/* global variables */
int nr, matr_pixel[NR][NC];
t_rectangle rect[MAX_RECT];
```

```
/* function prototypes (omitted) */
...

int main (int argc, char *argv[]) {
  FILE *fp;
  int i, color, level;
  int r, c, r0, r1, c0, c1;
  char line[MAXC];

  fp = fopen(argv[1],"r");

  for (r=0; r<NR; r++)
    for (c=0; c<NC; c++)
      matr_pixel[r][c] = -1;
```

Solution

```
#include <stdio.h>
#define NR 600
#define NC 800
#define MAXC 100
#def


/* t
typedef struct {
  int r0, r1, c0, c1;
  int level;
  int color;
} t_rectangle;
/* global variables */
int nr, matr_pixel[NR][NC];
t_rectangle rect[MAX_RETT];
```

Initialization of pixel matrix to -1

```

/* function prototypes (omitted) */
...

int main (int argc, char *argv[]) {
  FILE *fp;
  int i, color, level;
  int r, c, r0, r1, c0, c1;
  char line[MAXC];

  fp = fopen(argv[1],"r");

  for (r=0; r<NR; r++)
    for (c=0; c<NC; c++)
      matr_pixel[r][c] = -1;
```

Solution

```
for (i=0; fgets(line,MAXC,fp)!=NULL; i++);
  switch (line[0]) {
    case ‘p’: case ‘P’:
      sscanf (line,“%*c %d %d %d %d”,
        &r0,&c0,&color,&level);
      r1 = r0; c1 = c0;
      break;
    case ‘r’: case ‘R’:
      sscanf (line,“%*c %d %d %d %d %d %d”,
        &r0,&r1,&c0,&c1,&color,&level);
      break;
  }
  rect[i].r0 = r0; rect[i].r1 = r1;
  rect[i].c0 = c0; rect[i].c1 = c1;
  rect[i].color = color;
  rect[i].level = level;
}
```

```
  nr = i;
  fclose(fp);
  colorPixel();
  fp = fopen(argv[2],“w");
  writeVisible(fp);
  fclose(fp);
  r=atoi(argv[3]); c=atoi(argv[4]);
  color = rect[matr_pixel[r][c]].color;
  printf(“%d pixels of color %d\n”,
    countSameColor(color), color);
  printf(“%d rectangles on pixel (%d,%d)\n”,
    countRectangles(r,c), r, c);
  return 0;
}
```

Solution

```
for (i=0; fgets(line,MAXC,fp)!=NULL; i++);
  switch (line[0]) {
    case ‘p’: case ‘P’:
      sscanf (line,“%*c %d %d %d %d”,
        &r0,&c0,&color,&level);
      r1 = r0; c1 = c0;
      break;
    case ‘r’: case ‘R’:
      sscanf (line,“%*c %d %d %d %d %d %d”,
        &r0,&r1,&c0,&c1,&color,&level);
      break;
  }
  rect[i].r0 = r0; rect[i].r1 = r1;
  rect[i].c0 = c0; rect[i].c1 = c1;
  rect[i].color = color;
  rect[i].level = level;
}
```

Reading of points and rectangles: Points are converted into rectangles of size 1

```
  nr = i;
  fclose(fp);
```

```
  r=atoi(argv[3]); c=atoi(argv[4]);
  color = rect[matr_pixel[r][c]].color;
  printf(“%d pixels of color %d\n”,
    countSameColor(color), color);
  printf(“%d rectangles on pixel (%d,%d)\n”,
    countRectangles(r,c), r, c);
  return 0;
}
```

Solution

```
 for (i=0; fgets(line,MAXC,fp)!=NULL; i++);
   switch (line[0]) {
     case ‘p’: case ‘P’:
```

All sub-problems handled by functions:
- colorPixel
- writeVisible
- countSameColor
- countRectangles

```
   }
   rect[i].r0 = r0; rect[i].r1 = r1;
   rect[i].c0 = c0; rect[i].c1 = c1;
   rect[i].color = color;
   rect[i].level = level;
 }
```

```
  nr = i;
  fclose(fp);
  colorPixel();
  fp = fopen(argv[2],“w");
  writeVisible(fp);
  fclose(fp);
  r=atoi(argv[3]); c=atoi(argv[4]);
  color = rect[matr_pixel[r][c]].color;
  printf(“%d pixels of color %d\n”,
    countSameColor(color);
  printf(“%d rectangles on pixel (%d,%d)\n”,
    countRectangles(r,c);
  return 0;
}
```

Solution

```c
void colorPixel(void) {
  int i, level;
  int p, r, c;

  for (i=0; i<nr; i++)
    for (r=rect[i].r0; r<rect[i].r1; r++)
      for (c=rect[i].c0; c<rect[i].c1; c++) {
        level = rect[i].level;
        p = matr_pixel[r][c];
        if (p<0 || level > rect[p].level)
          matr_pixel[r][c] = i;
      }
}
```

```c
void writeVisible(FILE *fp) {
  int i, r, c, p, visible;
/* a rectangle is visible if at least one covered pixel is visible */
  for (i=0; i<nr; i++) {
    visible = 0;
    for (r=rect[i].r0; r<rect[i].r1 && !visible; r++)
      for (c=rect[i].c0; c<rect[i].c1 && !visible; c++) {
        p = matr_pixel[r][c];
        if (p == i)
          visible = 1;
      }
    if (visible) writeRectangle(fp,i);
  }
}
```

Solution

```
void writeRectangle(FILE *fp, int i) {
  int r0, r1, c0, c1;


  r0=rect[i].r0; r1=rect[i].r1;
  c0=rect[i].c0; c1=rect[i].c1;
  if ((r0 == r1) && (c0 == c1))
    fprintf (fp,“p %d %d”,r0,c0);
  else
    fprintf (fp,“r %d %d %d %d”,r0,r1,c0,c1);
  fprintf (fp,“%d %d\n”,rect[i].level,
    rect[i].color);
}
```

```
int countSameColor(int color) {
  int num=0 ,r, c, p;


  for (r=0; r<NR; r++)
    for (c=0; c<NC; c++) {
      p = matr_pixel[r][c];
      if (color == rect[p].color)
        num++;
    }
  return num;
}
```

Solution

```
int countRectangles(int r, int c) {
  int num=0;
  for (i=0; i<nr; i++) {
    if ((rect[i].r0 <= r && rect[i].c0 <= c)
     && (rect[i].r1 >= r && rect[i].c1 >= c))
      num++;
  }
  return num;
}
```