Exercises of programming v. 0.5

Andrea Marin
Mauro Tempesta

February 17, 2017

Contents

1 Simple cycles

This section contains some exercises that aim at improving the skills of the students on the usage of cycles. The exercises must be solved without using any C library but the `stdio.h` and adopting the cycle which is mostly suitable to the proposed solution. In order to achieve this, keep in mind what follows:

- You use the `for` cycle when, once the values of the variables are know, you can predict the number of iterations just by reading the heading of the cycle;
- You use the `do .. while` cycle when the `for` is inappropriate and you are sure that the cycle's block must be executed at least once;
- You use the `while` cycle in all the other cases.

Do not use the *jumping* instructions as *break*, *continue* and *return* to abort a cycle.

1.1 Co-prime numbers

Write a C program that reads two integers from the standard input and decides if they are coprime. The program must write the answer on the standard output. Two numbers are coprime if their only common positive divisor is 1.

Solution. First let us understand the meaning of coprimality. 5 and 12 are coprime because they do not share any divisor but 1 and -1; indeed the same holds for -5 and 12. If we consider 21 and 27 they are not coprime because they share 3 as a common divisor. Notice that 1 is coprime with every number (included 0), but 0 is coprime only with 1.

The easiest solution consists in using the *universal property* pattern. Indeed, if the two input numbers (say n_1 and n_2) are positive, we can reformulate the coprime condition as: *all the numbers in the interval* $[2, \min(n_1, n_2)]$ *are not simultaneously divisors of* n_1 *and* n_2. However, we must consider the case of negative numbers which is easy because we can just work with the absolute values. Finally, the case of $n_1 = 0$ or $n_2 = 0$ can be treated separately; however observe that if we consider the divisors in the interval $[2, \max(n_1, n_2)]$, then we immediately solve the problem when one of the two numbers is 0 and the other is not. So the case $n_1 = 0$ and $n_2 = 0$ must be considered separately anyway. The solution is shown in Table 1. A more efficient solution relies on the Euclide's algorithm for the computation of the largest common divisor. Also in this case pay attention to the case when one or both the input numbers are 0.

1.2 Exercise on sequence, give the sum and the quantity of numbers

Write a C program that reads from the standard input a sequence of numbers; the program must stop reading from the input sequence when it finds two equal consecutive numbers. Compute the sum and the quantity of all the read numbers.

Solution. The easiest way to address the problem consists in using the *sequence pattern*, i.e., we use variable *prec* to store the last read number and variable *cur* to store the current input. At each iteration of the cycle the value of *cur* is stored in *prec* and *cur* is newly read. The other patterns to apply is that of the *accumulator* to store the partial sum of the input sequence and of the *counter* to store the partial counting of the read numbers. Notice that we must do at least two read operations from the standard input, therefore, we do one reading before the cycle block and then we choose the `do .. while` cycle to ensure a second reading.

1.3 Printing the odd numbers in $[1, 1000]$

Write on standard output all the odd numbers included in the interval [1, 1000] with a newline every 10 printed numbers.

Solution. The exercise is very simple, the key-point here is counting the number of written numbers and adding a new line printing every ten. The solution is shown in Table 3. Try to rewrite the solution avoiding the use of variable *printed*.

```c
#include <stdio.h>

int main(int argc, char const *argv[]) {
   int n1, n2;    /*the two input numbers*/
   int coprime;   /*Are they coprime?*/
   int div;       /*Candidate common divisor*/


   printf("Give the first number: ");
   scanf("%d", &n1);
   printf("Give the second number: ");
   scanf("%d", &n2);
   /* compute the absolute value */
   if (n1 < 0)
      n1 = -n1;
   if (n2 < 0)
      n2 = -n2;

   /*Universal property: first assume it is true!*/
   coprime = 1;

   /*Start from divisor  2*/
   div = 2;
   while ( (div <= n1 || div <= n2) && coprime) {
      if(n1 % div == 0 && n2 % div == 0)
         coprime = 0;    /*Counterexample has been found!*/
      div = div + 1;
   }

   /*case of both number equal to zero*/
   if (n1 == 0 && n2 == 0)
      coprime = 0;

   /*output*/
   if (coprime)
      printf("The numbers are coprime.\n");
   else
      printf("The numbers are not coprime.\n");

   return 0;
}
```

Table 1: Decision of coprimality between two integers based on testing a universal property.

```c
#include <stdio.h>

int main(int argc, char const *argv[]) {
   int sum, count; /*accumulator and counter*/
   int cur, prev; /*current and previous values of the seqeunce*/

   scanf("%d", &cur);
   sum = cur;
   count = 1;
   do {
      prev = cur;
      scanf("%d", &cur);
      count = count + 1; /*counter*/
      sum = sum + cur; /*accumulator*/
   } while (prev != cur);
   printf("Read numbers: %d\nSum: %d\n", count, sum);

   return 0;
}
```

Table 2: Read a sequence of integers and stop when two consecutive numbers are equal. Compute the sum and count the quantity or read input.

```c
#include <stdio.h>

int main(int argc, char const *argv[]) {
    int i; /*odd number to be printed*/
    int printed; /*counting the printed numbers*/

    printed = 0;
    for (i = 1; i < 1000; i = i + 2) {
        printf("%d ", i);
        printed = printed + 1;

        if (printed % 10 == 0)
            printf("\n");
    }
    return 0;
}
```

Table 3: Printing the odd numbers in $[1, 1000]$; new line every 10.

1.4 Compute the integer $r-$th root of a positive number

Write a C program that reads from standard input two positive integers, n and r, and compute the floor approximation of the $r-$root of n:

$$x = \lfloor \sqrt[r]{n} \rfloor \qquad r, n \in \mathbb{N}^+$$

Solution. The problem can be a little difficult because we do not want to use the C Mathematics library. Let us imagine we are able to compute in some way x^r. In this case the exercise can be reformulated as follows: *find the smallest x such that $x^r > n$*, i.e., we can see the solution as an application of the *find the edge* pattern: so we start from $x = 1$ and we try all the successive values until we find the first for which $x^r > n$. Then the solution is $x - 1$. However computing x^r is easy because it is sufficient to multiply x by itself r times. The solution is depicted by Table 4, observe carefully where the initialisation of the variables are put.

```c
#include <stdio.h>

int main(int argc, char const *argv[]) {
   int n, r;  /*input variables*/
   int root, pow;
   int i;

   printf("Write n: ");
   scanf("%d", &n);
   printf("Write r: ");
   scanf("%d", &r);

   if (n > 0 && r > 0) {
      root = 1;
      pow = 1;
      while (pow <= n) {
         root = root + 1;
     pow = 1;
         for (i = 0; i < r; i++)
            pow = pow * radice;
      }

      printf("The floor approximation of
        the %d-root of %d is %d.\n", r, n, radice - 1);
   } else
      printf("Invalid input.\n");

   return 0;
}
```

Table 4: Compute $\lfloor \sqrt[r]{n} \rfloor$.

2 Recursion

2.1 Some notes on dynamic environments and variable scope

High level programming languages use identifiers to denote several entities among which variable names, function names and their formal parameters. In this short node we will mainly focus on the environment associated with variable and parameter identifiers for imperative languages. Formally, we can see the environment as a function ρ that maps each variable identifier with the memory location which is allocated for it. *Store* function σ associates a memory location with the value stored therein. Hence, if id is a variable identifier, ℓ its memory location and v its value, we have:

$$id \rightarrow_\rho \ell \rightarrow_\sigma v\,.$$

The *global environment* consists of the associations between identifiers and locations which are shared among different program units. These associations are created by the main program and can be exported by modules or libraries. The global environment maintains the associations from the very beginning of the program execution to its end. In the following code, variables `x,y` are the only global ones:

```
#include<stdio.h>
int x;

int foo(int a, int b) {
   int d;
   d = a+b;
   return d;
}

int y;
int main() {
   int c, d;
   ...
   return 0;
}
```

The *local environment* is formed by the environment associations that are created by the declarations at the beginning of a code block or by the parameter passing upon a function call. How do we create an association between an identifier and a memory location in a local environment? At the beginning of a block we can declare a variable and this creates the environment association between the identifier and the memory location. At the end of the block this association is *destroyed*. Notice that by *destroyed* we mean that the association is completely erased from ρ and the memory that has been allocated for the variables belonging to the local environment is freed. This mechanism is called *dynamic local environment* and is in contrast with the *static local environment* that does not destroy the associations but makes them invisible outside the block. The default mechanism used by *C* and most of the programming languages is the dynamic local environment. This implies that every time we enter the same code block all the local variables are newly instantiated and the values stored by previous executions are lost. Let us make an example:

```
int main() {
   int i, k;
   for (i=0; i<5; i++) {
      int c=0;
      k = ++c;
   }
   printf(``%d'', c);
}
```

Since we have a dynamic local environment, variable c is created and destroyed at each iteration and hence the program prints 1. Is it possible to have a static local environment in C? The answer is positive, and in fact, we need to resort to the keyword *static*. For instance the code:

```
int main() {
   int i, k;
   for (i=0; i<5; i++) {
      static int c=0;
      k = ++c;
   }
   printf(``%d'', c);
}
```

prints the value 5. Anyway, in our examples we will avoid the use of static local environments (and they should be avoided in general unless inevitable).

It is important to bear in mind that local identifiers:

1. are never accessible outside their blocks
2. are associated with their memory location as soon as we enter in a block
   (a) Every time we begin the block with the dynamic local environment
   (b) Only the first time we enter the block with the static local environment
3. destroyed (dynamic) or deactivated (static) at the end of the block.

Henceforth, we will work with dynamic local environments without further specifications.

Code blocks can be seen as functions without name and parameters. In fact, functions have their local environment that follows the rules that we have given so far. The local environment of C functions consists of the associations identifier/memory location for:

1. the local variables
2. the formal parameters

For instance, the local environment of the following function consists of the associations for the identifiers x, y:

```
float foo(int x) {
   int y;
   ...
}
```

Recall that the parameter passing in C follows the *by value* or *by copy* semantics. This means that once the formal parameters are instantiated, they are automatically initialised with the values obtained by the run-time evaluation of the expressions forming the list of actual parameters. Hence, the formal parameters *must not* be initialised or read within a function.

An important consequence of the dynamic local environment implemented in C is that when we use recursion we have to bear in mind that the environment associated with a function call is completely independent of the environment associated with the other function calls even if they are still pending! For instance, let us consider the following recursive function:

```
int factorial(int x) {
   int y;
   if (x>0)
      y = x*factorial(x-1);
   else
```

```
        y = 1;
    return y;
}
```

If we use 2 as actual parameters for x this will cause the dynamic creation of three sets of associations each of which corresponding to a local environment for factorial. The local environment containing the association for `x,y` at the first call for `x=2` is completely independent and not accessible of the local environment of the successive call, when `x=1`.

2.2 Marbles - 1

We have an infinite number of blue and red marbles. Write a function that computes the number of sequences of n marbles such that it never happens that three marbles with the same colours are consecutive.
Examples:

- If $n = 0$ the function computes 1
- If $n = 3$ the function computes 6 since the possible sequences are (R, B, B), (R, B, R), (R, R, B), (B, R, R), (B, R, B), (B, B, R)
- If $n = 4$ the functions computes 10

Solution Assume we can count the number of solutions for sequences of length $n - 1$ given the colour of the last two marbles. How do we apply the recursion? Imagine that we have to create a sequence of n marbles given the color of the last two. Then, if the last two marbles have the same color, we can add to the sequence only a marble of the other color, otherwise we can add a marble for each color. Clearly, now we can count in how many ways we can build sequences of $n-1$ marbles given the colors of the last two marbles. The base case is $n = 0$ which gives 1 possible sequence. The solution is shown in Table 5.

2.3 Marbles - 2

We are in the same setting of Exercise 2.2, but we want to count all the sequences in which the blue marbles apper only in sub-sequences with length exactly 2 with the exception of the last marble

```
int count_supp(int k, char last, char prev) {
    if (k==0)
        return 1;
    else
        if (last==prev)
            return (last=='R') ? count_supp(k-1, 'B', last) : count_supp(k-1, 'R', last);
        else
            return count_supp(k-1,'R',last)+count_supp(k-1,'B',last);

}

int count(int n) {
    return count_supp(n, 'x', 'y');
}
```

Table 5: Solution of the Marbles - 1 excercise

which can be a single blue.
Examples:

- If $n = 0$ the function computes 1
- If $n = 1$ the possible sequences are (R) and (B), i.e., the function computes 2
- If $n = 2$ the valid sequences are (R, R), (R, B), (B, B), i.e., the function computes 3
- If $n = 3$ the valid sequences are (R, B, B), (R, R, B), (B, B, R), (R, R, R), i.e., the algorithm computes 4
- If $n = 4$ the valid sequences are (R, R, R, R), (B, B, R, R), (R, B, B, R), (R, R, B, B), (R, R, R, B), (B, B, R, B), i.e., the algorithm computes 6

Solution The reasoning follows that of the previous excercise, i.e., we keep track of the last two marbles. The code is shown in Table 6

2.4 Marbles - 3

We are in the same setting of Exercise 2.2, but we want to count all the sequences in which it never happens that neither two blue marbles nor thress red marbles appear consectuvie.
Examples:

- If $n = 0$ the function computes 1
- If $n = 3$ the valid sequences are (R, B, R), (R, R, B), (B, R, R), (B, R, B), i.e., the function computes 4
- If $n = 4$ then the function computes 5

Solution The reasoning follows that of Excercise 2.2, i.e., we keep track of the last two marbles. The code is shown in Table 7

```
int count_supp(int k, char last, char prev) {
   if (k==0)
      return 1;
   else
      if (last=='B' && prev=='B')
         return count_supp(k-1, 'R', last);
      else
         if (last=='B' && prev!='B')
            return count_supp(k-1, 'B', last);
         else
            return count_supp(k-1, 'B', last) + count_supp(k-1, 'R', last);
}

int count(k) {
   return count_supp(k, 'x', 'y');
}
```

Table 6: Solution of the Marbles - 2 excercise

2.5 From iterative to recursive

We desire to compute the square root of a float number $n > 1$ according to the following algorithm:

1. `inf` and `sup` take values 1 and n, respectively
2. `x` takes value $(inf + sup)/2$
3. If $|n - x^2| < t$ then x is the approximating square root of n
4. If $x^2 > n$ then `sup` takes value x and jump to step 3
5. If $x^2 < n$ then `inf` takes value x and jump to step 3

where t denotes the tolerance. Write the recursive function `float mysqrt(float n, float inf, float sup, float t)` and a function call that computes the square root of 14.3 with a precision of $t = 0.00001$.

Solution The only difficulty in the exercise is understanding the signature of function `mysqrt`. We may interpret it as: *Find the square root inside the interval* $[inf, sup]$ *with tolerance* t. Therefore the solution is that shown in Table 8.

```
   int count_supp(int k, char last, char prev) {
   if (k==0)
      return 1;
   else
      if (last=='R' && prev=='R')
         return count_supp(k-1, 'B', last);
      else
         if (last=='B')
            return count_supp(k-1,'R',last);
         else
            return count_supp(k-1,'R',last) + count_supp(k-1,'B',last);

}


int count(k) {
   return count_supp(k, 'x', 'y');
}
```

Table 7: Solution of the Marbles - 3 excercise

```c
#include<math.h>
#include<stdio.h>

float mysqrt(float n, float inf, float sup, float t) {
  float x = (inf+sup)/2.0;
  if (fabs(n-x*x) < t)
    return x;
  else
    if (x*x>n)
      return mysqrt(n, inf, x, t);
    else
      return mysqrt(n, x, sup, t);
}

int main() {
  float res;
  res = mysqrt(14.3, 1.0, 14.3, 0.00001);
  printf("The square root of 14.3 is approximatively %f\n", res);
  return 0;
}
```

Table 8: Recursive computation of the square root of a float number.

3 Arrays

3.1 What you have to bear in mind about arrays

We begin this section by discussing some rules that you have to keep in mind when you work with arrays in C.

Rule 1: Array length
According to the standard ANSI C89 and C90 the array length must be a *constant integer expression*, i.e., the size of the array must be known at compile time. Hence, the following code is **not** correct:

```
int n;
scanf(''%d'', &n);
if (n>0) {
   int vett[n];
   int i;
   for (i=0; i<n; i++) {
      scanf(''%d'', &n);
   }
   ...
}
```

This limitation has been removed in the standard ANSI C99 and is supported by many but not all the compilers. For this reason we will work according to the standard C89.

Rule 2: Indexes
The element index in an array ranges from 0 (the first element) to $DIM - 1$ (last element) where DIM is the array size.

Rule 3: Initialisation of an array
When an array is declared it can be also initialised. For instance

```
float vett[] = {12.4, 14.0, 1.8};
```

declares an array of 3 elements whose values are those specified in after the symbol =. Notice that since we do not specify the array length this is automatically inferred at *compile time* by counting the number of elements in the initialisation. According to the standard C89 the elements in the initialisation must be constants. What happens if we specify both the array size and the list of elements? Let us consider the following example:

```
float vett[DIM] = {12.4, 14.0, 1.8};
```

where `DIM` is a previously defined constant. If `DIM`<3 then the compiler raises an error, if `DIM`>3 then the elements from position 3 to `DIM`-1 are filled with 0s. We can also initialise a string seen as a vector of chars:

```
char buffer[] = ''Hello'';
```

We are allocating an array of 6 chars: for are used to store the chars $H \ldots o$ and one for the char representing the end of string.

Rule 4: Assignment
In C89 and C99 you cannot assign a whole array to another array. Examples of **wrong** codes are the following:

```
int v1[10];
int v2[10];
```

```
int main() {
   ...
   v1 = {3, 4, 2, 2, 2, 2, 2, 2, 6, 0};
   v2 = v1;
   ...
}
```

The first assignment (when it cannot be done in the initialisation) must be replaced by

```
v1[0] = 3;
v1[1] = 4;
...
```

while the latter:

```
int i;
for (i=0; i<10; i++)
   v2[i] = v1[i];
```

Notice that this rule holds for all the arrays, including the strings. So we **cannot** assign a value to a string by using the following code:

```
char mystring[10];
...
mystring = ``Hello'';
...
```

The assignment can be replaced by the assignments of the single element of the array or by using the library function `strcpy`.

Rule 5: Comparison of arrays
If we want to test if two arrays contain the same element we cannot use the operator `==` on the whole array, but we must use a comparison element by element. For example, the following code is **wrong**:

```
char mystr[] = ``Hello'';
char buffer[10];
...

scanf(``%s'', buffer);
if (mystr == buffer)
   printf(``You wrote Hello'');
else
   printf(``You wrote something different from Hello'');

...
```

The correct version is:

```
char mystr[] = ``Hello'';
char buffer[10];
int test; int i;
...

scanf(``%s'', buffer);
test = 1;
i = 0;
while ((mystr[i]!='\0' || buffer[i]!='\0') && test) {
   if (mystr[i] != buffer[i])
      test = 0;
   i++;
}
```

```
if (test)
   printf(``You wrote Hello'');
else
   printf(``You wrote something different from Hello'');

...
```

Unfortunately, the comparison of arrays with the operator `==` does not rise any error in compile time: in fact, it is a legitimate operation but its semantics is not that of comparing the two arrays element by element. In case of strings we can use the `strcmp` library function to test the equality and their alphabetical order.

3.2 Given an ordered array eliminate duplicated elements

Given an ordered array of integers and its size, eliminate the duplicated elements. The resulting array must be instantiated dynamically with the minimal size. The function returns the instantiated array and its size.

Solution The solution is rather simple and consists of two phases: first we count the number of different elements in the input vector, the we instantiate the resulting vector in the heap and copy the elements. We also provide a `main` in Table 9.

3.3 Eliminate duplicated elements from a non-ordered array

Write the function:
`int different(int input[], int size, int **vetout);`
while given an array `input` of integers with dimension `size` creates a vector in dynamic memory that contains all the elements of `input` repeated once. Function `different` returns the size of the modified array.

Solution We use array `sup` to spot the repetitions of the elements in `input`. `sup` is allocated in the dynamic memory with the same size of `input`. An element 1 denotes a fresh element, 0 a repeated one. The solution is shown in Table 10.

3.4 Intersection of two sets

Given two arrays of integers representing sets of numbers, implement a function that allocates in the heap a new array (of minimal size) containing the intersection of the two provided sets. The empty set is represented by the `NULL` pointer and size 0. The function returns 1 if the allocation has succeeded, 0 otherwise.

Solution. The solution is similar to the previous exercise. We rely on the array `commons` to track which elements in `v1` occur also in `v2` (notice that `calloc` initializes all the positions of the array to 0). Next we instantiate an array of the minimum size required to store the intersection of the two sets and copy in it the common elements. The solution is reported in Table 11.

```
/*assume size>0*/
int deleteduplicated(int* vect, int size, int** pres, int *psize) {
  int i;
  *psize = 1;
  for (i=1; i<size; i++)
    if (vect[i] != vect[i-1])
      (*psize)++;
  *pres = (int*) malloc (sizeof(int)*(*psize));
  if (*pres) {
    int j=1;
    (*pres)[0]=vect[0];
    for (i=1; i<size; i++) {
      if (vect[i] != vect[i-1]){
        (*pres)[j] = vect[i];
        j++;
      }
    }
    return 1;
  }
  else
    return 0;
}

int main() {
  int vector[] = {5, 5, 9, 9, 9, 10, 14, 15, 15, 20};
  int* result;
  int dimensione;

  if (deleteduplicated(vector, 10, &result, &dimensione)) {
    int i;
    for (i=0; i<dimensione; i++)
      printf(" %d ", result[i]);
  }
  free(result);
  return 0;
}
```

Table 9: Eliminates duplicated elements from an ordered vector.

```
int different(int input[], int size, int **vetout) {
   int *sup = (int*)malloc(sizeof(int) * size);
   int i,j;
   int unequal = 0;

   /*initialisation of the vector of duplicates*/
   for (i=0; i<size; i++)
      sup[i] = 1;

   /*marking of duplicates*/
   for (i=0; i< size; i++)
      if (sup[i]==1) {
         unequal++;
         for (j=i+1; j<size; j++)
            if (input[i]==input[j])
               sup[j] = 0;
      }

    /*output array*/
    *vetout = (int*)malloc(sizeof(int)*unequal);

    /*copy of the elements in the output array*/
    j=0;
    for (i=0; i<size; i++) {
       if (sup[i] == 1) {
          (*vetout)[j]=input[i];
          j++;
       }
    }

    free(sup);
    return unequal;
}

int main() {

   int vect[] = {3,4,2,3,6,7,3,4};
   int *ris;
   int sizeris;


   sizeris = different(vect, 8, &ris);

   for (i=0; i<sizeris; i++)
      printf("\n %d ",ris[i]);

   free(ris);
   return 0;
}
```

Table 10: Remove duplicates from a non-ordered array

```
int intersection(int v1[], int dim1, int v2[], int dim2, int **res, int *dim_res) {
    char *commons;
    int i, j;

    if (dim1 == 0 || dim2 == 0) {
        *ris = NULL;
        *dim_ris = 0;
        return 1;
    }
    common = calloc(dim1, sizeof(char));
    if (commons == NULL)
        return 0;
    *dim_res = 0;
    for (i = 0; i < dim1; i++) {
        j = 0;
        while (j < dim2 && v2[j] != v1[i])
            j++;
        if (j < dim2) {
            commons[i] = 1;
            (*dim_res)++;
        }
    }
    if (*dim_res == 0)
        *res = NULL;
    else {
        *res = malloc(*dim_res * sizeof(int));
        if (*res == NULL) {
            free(comuni);
            return 0;
        }
        j = 0;
        for (i = 0; i < dim1; i++)
            if (commons[i] == 1) {
                (*res)[j] = v1[i];
                j++;
            }
    }
    free(commons);

    return 1;
}
```

Table 11: Intersection of two sets.

3.5 Union of two sets

Given two arrays of integers representing sets of numbers, implement a function that allocates in the heap a new array (of minimal size) containing the intersection of the two provided sets. The empty set is represented by the `NULL` pointer and size 0. The function returns 1 if the allocation has succeeded, 0 otherwise.

Solution. The solution is very similar to the previous exercise. We use `commons` to track which elements in `v2` occur also in `v1` (notice that `calloc` initializes all the positions of the array to 0). Next we instantiate an array of the minimum size required to store the union of the two sets and copy in it all the elements of `v1` and those in `v2` that do not occur in `v1`. The solution is reported in Table 12.

3.6 Longest sequence - 1

Write a function that takes as input a non-empty array of integers and its size and computes the index of the longest sequence of consecutive, increasing numbers in the array. In case of multiple sequences with the same length, return the index of the one that appears first.

Solution. The solution is pretty simple, as we only need to remember the length and position of the maximum sequence found so far and update these values once we find a sequence that is strictly longer than the previous one. The solution is reported in Table 13.

3.7 Longest sequence - 2

Write a function that takes as input a non-empty string and computes the index of the longest sequence of consecutive characters in the string. In case of multiple sequences with the same length, return the index of the one that appears first.

Solution. The only difference with respect to the previous exercise is the condition of loops, as we need to stop when we find the terminating character `\0`. The solution is reported in Table 14.

3.8 Longest sequence - 3

Write a function that takes as input a non-empty array of integers and its size and computes the index of the longest sequence of consecutive, decreasing numbers in the array. In case of multiple sequences with the same length, return the index of the one that appears first.

Solution. The only difference with respect to the exercise *Longest sequence - 1* is the condition that checks for a decreasing sequence instead of an increasing one. The solution is reported in Table 15.

3.9 Longest sequence - 4

Write a function that takes as input a non-empty string and returns the pointer of the first character of the longest sequence of consecutive characters in the string. In case of multiple sequences with the same length, return the pointer to the one that appears first.

```
int union(int v1[], int dim1, int v2[], int dim2, int **res, int *dim_res) {
    char *commons;
    int num_commons;
    int i, j;

    if (dim1 == 0 && dim2 == 0) {
        *res = NULL;
        *dim_res = 0;
        return 1;
    }
    commons = NULL;
    num_commons = 0;
    if (dim2 != 0) {
        commons = calloc(dim2, sizeof(char));
        if (commons == NULL)
            return 0;
        for (i = 0; i < dim2; i++) {
            j = 0;
            while (j < dim1 && v1[j] != v2[i])
                j++;
            if (j < dim1) {
                commons[i] = 1;
                num_commons++;
            }
        }
    }
    *dim_res = dim1 + dim2 - num_commons;
    *res = malloc(*dim_res * sizeof(int));
    if (*res == NULL) {
        free(commons);
        return 0;
    }
    for (i = 0; i < dim1; i++)
        (*res)[i] = v1[i];
    for (j = 0; j < dim2; j++)
        if (commons[j] == 0) {
            (*res)[i] = v2[j];
            i++;
        }
    free(commons);

    return 1;
}
```

Table 12: Union of two sets.

```
int longest_sequence_1(int vet[], int dim) {
    int i, len, max_len, idx_max_len;

    i = idx_max_len = max_len = 0;
    while (i < dim) {
        len = 1;
        while (i+len < dim && vet[i] + len == vet[i+len]) {
            len++;
        }
        if (len > max_len) {
            max_len = len;
            idx_max_len = i;
        }
        i += len;
    }
    return idx_max_len;
}
```

Table 13: Longest sequence - 1.

```
int longest_sequence_2(char str[]) {
    int i, len, max_len, idx_max_len;

    i = idx_max_len = max_len = 0;
    while (str[i] != '\0') {
        len = 1;
        while (str[i+len] != '\0' && str[i] + len == str[i+len]) {
            len++;
        }
        if (len > max_len) {
            max_len = len;
            idx_max_len = i;
        }
        i += len;
    }
    return idx_max_len;
}
```

Table 14: Longest sequence - 2.

Solution. The only difference with respect to the exercise *Longest sequence - 2* is in the last instruction, since we must return the pointer of the first character of the sequence, not the index. The solution is reported in Table 16.

```
int longest_sequence_3(int vet[], int dim) {
    int i, len, max_len, idx_max_len;

    i = idx_max_len = max_len = 0;
    while (i < dim) {
        len = 1;
        while (i+len < dim && vet[i] - len == vet[i+len]) {
            len++;
        }
        if (len > max_len) {
            max_len = len;
            idx_max_len = i;
        }
        i += len;
    }
    return idx_max_len;
}
```

Table 15: Longest sequence - 3.

```
char* longest_sequence_4(char str[]) {
    int i, len, max_len, idx_max_len;

    i = idx_max_len = max_len = 0;
    while (str[i] != '\0') {
        len = 1;
        while (str[i+len] != '\0' && str[i] + len == str[i+len]) {
            len++;
        }
        if (len > max_len) {
            max_len = len;
            idx_max_len = i;
        }
        i += len;
    }
    return str + idx_max_len;
}
```

Table 16: Longest sequence - 4.

4 Recursion on arrays

4.1 Recursive upcase

Write a C subroutine that upcases a string recursively.

Solution. The key-points to provide a simple solution are:

- if `str` is a non-empty string then `str + 1` is a valid string
- when the subroutine is called the parameters are passed by value. However, in this case what is copied is address of the first element of the string; therefore, any change to the referenced object (a char) represents a side effect of the function call.

The solution is displayed in Table 17.

4.2 String anagrams

Given a string composed of different chars, write a recursive function that prints on standard output all its anagrams.

Solution Suppose we have a string of length n and that we are able to generate all the anagrams of strings of length $n-1$, who can we solve our problem? The answer is rather simple. Indeed, we can swap the first string char with all the remaining ones and then generate the anagrams of a shorter string. Suppose we wish to generate the anagrams of the string `''ABCD''`. These are:

- `'A'` + anagrams of `'BCD'`
- `'B'` + anagrams of `'ACD'`
- `'C'` + anagrams of `'BAD'`
- `'D'` + anagrams of `'BCA'`

The solution is shown in Table 18

4.3 Creation of a maze

In this exercise we want to create a maze of dimension $N \times M$, where N and M are taken as input values. The starting point is the left-bottom corner and the arrival point is the top-right corner. All the cells must be reachable from the starting point. The program must store the maze in a data structure and display it on the standard output.

Solution We first discuss the encoding of the maze. Ideally, we would like to have a $N \times M$ bi-dimensional array allocated in the dynamic memory. We will linearize the data structure in order to simplify the memory management operations. Each cell contains an encoding of the surrounding walls: *Left*, *Up*, *Right*, *Down*. Moreover, we would like to have a marking of a cell, i.e., *Visited*. A cell can have any combination of these attributes, e.g., it may have the left, up, and right walls and be visited. We exploit the binary encoding shown in the following table:

```
void upcaseric(char* str) {
   if (*str!='\0') {
      if (*str >= 'a' && *str <='z')
         *str = *str - 'a' + 'A';
      upcaseric(str+1);
   }
}
```

Table 17: Recursive upcase of a string.

```
void swap(char* c1, char* c2) {
   char c = *c1;
   *c1 = *c2;
   *c2 = c;

}

void permute(char* str, char* from) {
   if (!*from) {
      printf("%s\n", str);
   }
   else {
      int l = strlen(from) - 1;
      int i;
      for (i=0; i<=l; i++) {
         swap(from, from+i);
         permute(str, from+1);
         swap(from, from+i);
      }
   }
}
```

Table 18: Recursive generation of all the anagrams of a string.

Attribute	Binary enc.	Base 10
Left	00001	1
Up	00010	2
Right	00100	4
Down	01000	8
Visited	10000	16

Following this approach we can combine the attributes by using simple binary operations: |, & and $\sim$. For instance the visited cell with left, up and right border has value: `Left|Up|Right|Visited`.

We adopt the following recursive algorithm to construct the maze from a starting point r, c:

1. Mark cell r, c as visited
2. Let $\mathcal{R}$ be the set of unvisited neighbours of r, c
3. If $\mathcal{R} = \emptyset$ terminate
4. Pick a random element (r', c') in $\mathcal{R}$
5. Create the maze from (r', c')
6. Return to step 2

Table 19 shows the definition of the constants and the subroutine which allocates the data structure. Moreover we use a function to linearize the coordinates. The recursive algorithm is implemented by function `createroutes` shown in Table 20. The array `positions` contains the candidate positions for the next visit. The North, South, West and East cells are denoted by codes 0, 1, 2, 3, respectively. In principle there are 4 available cells but some of them are not valid because they are outside the field or because they have already been visited. The cycle `while (i<available)` counts the number of usable cells to keep the caving and ensure that they are all contained in the first positions of the array `positions`. Then, we choose a valid direction via `u`. According to the chosen direction, we remove the maze walls. The recursive call follows.

4.4 Longest sequence of identical chars

Implement the following *recursive* function:
`char* consec(char str[], int *lung);` that returns the address of the string char where the first longest sequence of identical chars start. In `*lung` store the length of longest sequence found. If the string is empty, then return `NULL`. Examples:

- In the string *ippodromo* return the address of the first 'p' and `*lung` is 2
- In the string *aabbbcdddbbe* return the address of the first 'b' and `*lung` is 3

Solution The solution is based on the following recursive scheme:

- If the string is the empty string, then return NULL and `*lung` is 0
- Otherwise count the length of the identical chars in the string prefix and compare the length with the result of the recursive call on the remaining string. Set the result according to the outcome of the comparison.

The code is shown in Table 21.

```
#define EMPTY 0
#define LEFT 1
#define UP 2
#define RIGHT 4
#define DOWN 8
#define VISITED 16

int coord(int r, int c, int dimc) {
   return r*dimc+c;
}

void createmaze(int** maze, int dimr, int dimc) {
   int i, r, c;

   *maze = (int*) malloc((dimr * dimc) * sizeof(int));

   for (i=0; i<dimr*dimc; i++)
      (*maze)[i] = LEFT | UP | RIGHT | DOWN;

   /*select the bottom left corner as starting point*/

   r = dimr-1;
   c = 0;

   (*maze)[coord(r, c, dimc)] |= VISITED;
   (*maze)[coord(r, c, dimc)] &= (~DOWN);

   srand ( time(NULL) );

   createroutes(*maze, dimr, dimc, r, c);

   (*maze)[coord(0,dimc-1,dimc)] &= (~UP);

   /*remove visited information*/
   for (i=0; i<dimr*dimc; i++)
      (*maze)[i] &= (~VISITED);

}
```

Table 19: First part of the maze creation problem.

```
void createroutes(int *maze, int dimr, int dimc, int r, int c) {
   int i, u;
   /* N=0, S=1, W=2, E=3*/
   int positions[4][3] = {{0, r-1, c}, {1, r+1, c}, {2, r,c-1}, {3, r,c+1}};
   int nr, nc;
   int available = 4;
   while (available>0) {
      i = 0;
      while (i<available) {
         if (positions[i][1]<0 || positions[i][1]>=dimr ||
               positions[i][2]<0 || positions[i][2]>=dimc ||
               (maze[coord(positions[i][1],positions[i][2],dimc)]&VISITED)) {

            positions[i][0] = positions[available-1][0];
            positions[i][1] = positions[available-1][1];
            positions[i][2] = positions[available-1][2];

            available--;
         }
         else
            i++;
      }

      if (available) {
         u = rand()%available; /*random number between 0 and available-1*/
         maze[coord(positions[u][1],positions[u][2],dimc)] |= VISITED;
         /*fix the walls*/
         switch(positions[u][0]) {
            case 0:  /*Going North*/
               maze[coord(r, c, dimc)] &= (~UP);
               maze[coord(positions[u][1], positions[u][2], dimc)] &= (~DOWN);
               break;
            case 1: /*Going South*/
               maze[coord(r, c, dimc)] &= (~DOWN);
               maze[coord(positions[u][1], positions[u][2], dimc)] &= (~UP);
               break;
            case 2: /*Going West*/
               maze[coord(r, c, dimc)] &= (~LEFT);
               maze[coord(positions[u][1], positions[u][2], dimc)] &= (~RIGHT);
               break;
            case 3: /*Going East*/
               maze[coord(r, c, dimc)] &= (~RIGHT);
               maze[coord(positions[u][1], positions[u][2], dimc)] &= (~LEFT);
         }

         createroutes(maze, dimr, dimc, positions[u][1], positions[u][2]);

         positions[u][0] = positions[available-1][0];
         positions[u][1] = positions[available-1][1];
         positions[u][2] = positions[available-1][2];

         available--;
      }

   }
}
```

Table 20: Recursive algorithm for the maze creation.

```c
char* consec(char str[], int *lung) {
   char *pc = str;

   if (*str == '\0') {
      *lung = 0;
      return NULL;
   }
   else{
      int cl = 1, ric;
      char* pstr;
      while (*str == *(str+1)) {
         cl++;
         str++;
      }
      pstr = consec(str+1, &ric);
      if (ric>cl) {
         *lung = ric;
         return pstr;
      }
      else {
         *lung = cl;
         return pc;
      }
   }
}

int main(){
   char str1[] = "ippodromo";
   char str2[] = "aabbbcdddbbe";
   char *pc;
   int l;

   pc = consec(str1, &l);
   printf("Sequenza di lunghezza %d: %s\n", l, pc);


   pc = consec(str2, &l);
   printf("Sequenza di lunghezza %d: %s\n", l, pc);
   return 0;
}
```

Table 21: Longest sequence of consecutive chars.

```
typedef struct cell{
    int digit;
    struct cell * next;
} t_cell;

typedef t_cell* t_list;
```

Table 22: Types for managing the numbers as lists.

5 Exercises on simple lists

5.1 Numbers as lists

Write a subroutine that transforms a positive integer number given as parameter into a list of integers. Each node of the list contains a decimal digit of the number, the empty list represents the number 0. The order of the digit must be defined such that the original number (if different from 0) can be read by printing the list elements from the last to the first.

Solution. The creation of the list from the number is a simple application of appending a new cell to the end of the list. The definition of the types is given in Table 22 and the solution is given in Table 23 (iterative) and Table 24 (recursive).

5.2 Sum the lists

Given the number encoding defined by Exercise 5.1, define a subroutine that sums two numbers represented as lists. Do not convert the lists into the standard int type.

Solution. We propose a recursive solution that basically implements the algorithm for the sum of decimal numbers. The function requires a third parameter which is the carry, so when called to sum two lists this should be set to 0. The recursion takes advantage of the number representation that puts the less significant digit in the head of the list. The solution is shown in Table 25 and the data types are those defined in Table 22.

5.3 Remove instances of an element

Given a list of `char`, remove all the cells that contain the value `'a'` of `'A'`.

Solution. The problem can be easily addressed with the recursive function illustrate in Table 26.

5.4 Difference between list and vector

Given a list of integers ℓ, a vector of integers v and its dimension, write a function that returns the elements which are contained by ℓ but not by v.

```
t_list create_list(unsigned int number) {
   t_list result=NULL;
   t_list pc = NULL;

   int digit;

   while (number > 0) {
      digit = number % 10;
      number = number / 10;

      t_list newcell = (t_list) malloc(sizeof (t_cell));
      newcell->digit = digit;
      newcell->next = NULL;
      if (pc) {
         pc->next = newcell;
     pc = pc->next;
      }
      else { /*first element*/
         pc = newcell;
     result = pc;
      }
   }
   return result;
}
```

Table 23: Creation of list from a positive number (iterative solution).

```
t_list create_list(unsigned int number) {
   if (number == 0)
      return NULL;
   else {
      t_list newcell = (t_list) malloc(sizeof(t_cell));
      newcell->digit = number % 10;
      newcell->next = create_list(number/10);
      return newcell;
   }
}
```

Table 24: Creation of list from a positive number (recursive solution).

```
t_list sum_list(t_list l1, t_list l2, int carry) {

   if (l1==NULL && l2==NULL && carry ==0)
      return NULL;
   else {

      t_list next1 = NULL;    /*next digits*/
      t_list next2 = NULL;
      int next_carry;

      t_list result = (t_list)malloc(sizeof(t_cell));
      result->digit = carry;

      if (l1) {
         result->digit += l1->digit;
         next1 = l1->next;
      }

      if (l2) {
         result->digit += l2->digit;
         next2 = l2->next;
      }

      next_carry = result->digit/10;
      result->digit = result->digit%10;

      result->next = sum_list(next1, next2, next_carry);

      return result;
   }
}
```

Table 25: Number as lists, implementing the sum.

```
struct listchar{
  char info;
  struct listchar* next;
};

typedef struct listchar* Listchar;

void remove_a_rec(Listchar *plist) {
  if (*plist) {
    /*remove 'a' and 'A' from the tail*/
    remove_a_rec(&((*plist)->next));
    if (((*plist)->info == 'a')||(*plist)->info == 'A') {
      Listchar pc = *plist;
      *plist = (*plist)->next;
      free(pc);
    }
  }
}
```

Table 26: Remove values `'a'` and 'A' from a list of `char`.

```
struct listint{
  int info;
  struct listint *next;
};

typedef struct listint *Listint;

Listint difference_list_vect(Listint l, int vect[], int dim){
  if (l) {
    int i;
    int present;
    i = 0;
    present = 0;
    while (!present && i<dim) {
      if (vect[i] == l->info)
        present = 1;
      i++;
    }
    if (!present) {
      Listint newcell = (Listint) malloc(sizeof(struct listint));
      newcell->info = l->info;
      newcell->next = difference_list_vect(l->next, vect, dim);
      return newcell;
    }
    else
      return difference_list_vect(l->next, vect, dim);

  }
  else
    return NULL;
}
```

Table 27: Compute the difference between a list of integers and a vector of integers.

Solution. We propose a recursive solution of the problem. For each cell of the list, we check if its element is contained in the vector; if this case the function returns the list generated by the tail of ℓ and the same vector v. Otherwise, it generates a new cell which is the head of a list whose tail is the given by the same recursive call. The solution is shown in Table 27.

5.5 Identify words in a paragraph

Given a string containing a paragraph, write a function that returns a list of strings in which each cell contains a word of the paragraph. Assume that the every word is separated from the other by a sequence of at least one *blank* `' '`. Do not use any function from library `string.h`.

Solution. We propose a recursive solution and we use an auxiliary function `mysrtlen` that computes the length of the first word in the paragraph. At each recursive call a word is processed. The solution is shown in Table 28.

5.6 Compare lists of string

Given two lists of strings, write a function that decides if they are equals. The comparison between the strings must be done char by char.

```
struct stringlist{
  char *info;
  struct stringlist *next;
};

typedef struct stringlist *Stringlist;

int mystrlen(char *str) {
  int n=0;
  while (*str!='\0' && *str!=' ') {
    str++;
    n++;
  }
  return n;
}


Stringlist find_words(char *par) {
  if (*par == '\0')
    return NULL;
  else
    if (*par == ' ')
      return find_words(par+1);
    else {
      int len, i;
      Stringlist newcell =
         (Stringlist) malloc(sizeof (struct stringlist));
      len = mystrlen(par);
      newcell->info = (char*)malloc((len + 1)*sizeof(char));
      for (i=0; i<len; i++)
        (newcell->info)[i] = par[i];
      (newcell->info)[len] = '\0';
      newcell->next = find_words(par+len);
      return newcell;
    }
}
```

Table 28: Given a paragraph, find a list in which each cell contains a word of the paragraph.

```
struct liststring{
  char *info;
  struct liststring *next;
};

typedef struct liststring *Liststring;

int equals(Liststring l1, Liststring l2) {
  if (!l1 && !l2)
    return 1;
  else {
    if (!l1 || !l2)
      return 0;
    else
      return (!strcmp(l1->info, l2->info) && equals(l1->next, l2->next));
  }
}
```

Table 29: Comparison of two lists of strings.

Solution. The solution is simply based on the recursive comparison of two lists and is shown in Table 29.

5.7 Substring in list

Write a function that decides if, given a list of `char` and a string, the string is contained in the list.

Solution. The solution is shown in Table 30. Observe that we need two nested cycles to deal with the cases such as the following: consider the list containing the chard `a-a-a-c-t` and the string ‘‘`aac`’’; the string is contained in the list, but we we must rollback after the failure in recognising the sequence `a-a-a` to the second char.

5.8 Cumulative sums

Write a function that takes a list of integers and modifies it in such a way that each element is replaced with the sum of all the elements in the list from it up to the end.

Solution. The solution, reported in Table 31, is pretty straightforward. We consider as base cases for the recursion both the empty list and the singleton list, where no operation needs to be performed.

5.9 Average daily temperature

Consider a list of temperatures measurements collected during a day, where each cell contains an integer `time` and a float `temp`. The first field represents the moment when the measure has been taken, expressed as number of minutes since the midnight, while the second is the temperature. Given a list with n measurements of the type $(time_i, temp_i)$ with $i \in [1..n]$, write a function that computes the average temperature of the day according to the formula

$$\frac{\sum_{i=1}^{n}(time_i - time_{i-1}) \cdot temp_i}{time_n}$$

```c
struct cell{
  char c;
  struct cell *next;
};

typedef struct cell *listchar;

int substring(listchar l, char* str) {
  listchar pc;

  int found = 0;
  int i;

  while (l && !found) {
    i = 0;
    pc = l;
    while(pc && str[i] && pc->c == str[i]) {
      pc = pc->next;
      i++;
    }
    found = (str[i]=='\0');
    l = l->next;
  }

  return found;
}
```

Table 30: Is a string contained in a list of char?

```c
struct cell{
  int info;
  struct cell *next;
};

typedef struct cell *tlist;

void cum_sums(tlist l) {
    if (l != NULL && l->next != NULL) {
        cum_sums(l->next);
        l->info += l->next->info;
    }
}
```

Table 31: Cumulative sums of the elements of a list.

```
struct cell{
  int info;
  struct cell *next;
};

typedef struct cell *tlist;
int average_temperature(tlist l, float *avg_temp) {
    int prev_time;
    float sum_temps;

    if (l == NULL)
        return 0;
    sum_temps = prev_time = 0;
    while (l != NULL) {
        sum_temps += (l->time - prev_time) * l->temp;
        prev_time = l->time;
        l = l->next;
    }
    *avg_temp = sum_temps / prev_time;
    return 1;
}
```

Table 32: Average daily temperature.

```
tlist last(tlist l, int elem) {
    tlist plast;

    if (l == NULL)
        return NULL;
    plast = last(l->next, elem);
    if (plast == NULL && l->info == elem)
        plast = l;
    return plast;
}
```

Table 33: Last occurrence of an element in a list.

where $time_0 = 0$.

Solution. The solution is shown in Table 32. If the provided list is empty, the function returns 0 to denote that the average temperature has not been computed. Otherwise, the function returns 1 and the average is returned via the output parameter `avg_temp`.

5.10 Last occurrence of an element

Write a recursive function that receives as input a list of integers and a number and returns the pointer to the last cell of the list containing that number. If it is not in the list, return `NULL`.

Solution. The solution is shown in Table 33. The base case for the recursion is the empty list: in this case, the element is not present and we return `NULL`. Otherwise, we search for the element in the tail of the list and, if not found, we return the pointer to the head if it contains the searched element.

```
void setzerotail(tlist l){
   if (l) {
      setzerotail(l->next);
      if ((!(l->next) || l->next->info == 0) && l->info < 0)
         l->info = 0;
   }
}
```

Table 34: Set zeros to the tail of a list from the last occurrence of a positive number (excluded) to the end.

5.11 Write zeros at the end of a list

Given a list of int that, by hypothesis, does not contain any zero value implement the function:

```
void setzerotail(tlist l);
```

that sets to zero all the elements from the last positive number to the end of the list. Examples:

- 1-(−4)-6-(−2)-(−5) returns 1-(−4)-6-0-0
- empty list returns empty list
- 6 returns 6
- (−5) − (−3) returns 0-0

Do not use any other auxiliary function or call to the `malloc` function to duplicate the nodes.

Solution. When do we have to change a negative number to zero? When it is the last element of the list or when it is followed by a zero (remember that by hypothesis the list has no zeros). The recursion visit the list from the tail to the head. The solution is shown in Table 34.

5.12 Reverse a list

Write a function that reverses a list of integers and satisfies the following constraints:

- the elements are never read or copied;
- the list must be traversed only once;
- the use of functions for dynamic memory management (`malloc`, `free`, ...) is not allowed.

Solution. In the iterative solution we use three pointers, one to the current node n, one to the node that precedes n and one to the node that follows n. As we traverse the list, we modify the `next` field of n so that it points to the element that precedes it. The code is shown in Table 35.

Let us consider a recursive solution for the same problem. We have two base cases, the empty list and the singleton list, which are both the reverse of themselves. Otherwise, let us consider a list with at least two nodes, n_1 and n_2: we call recursively the function on the sub-list having n_2 as head to reverse it, then update the `next` pointer of n_2, the tail of the reversed sub-list, so that it points to n_1, the new tail of the reversed list. Finally we have to set the `next` pointer of n_1 and propagate the pointer to the head of the reversed list. The solution is shown in Table 36.

```
void reverse(tlist *l) {
    tlist curr, prev, next;

    curr = *l;
    prev = NULL;
    while (curr != NULL) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    *l = prev;
}
```

Table 35: Reverse a linked list: iterative version.

```
void rreverse(tlist *l) {
    if (*l && (*l)->next) {
        tlist pc = (*l)->next;
        rreverse(&pc);
        (*l)->next->next = *l;
        (*l)->next = NULL;
        (*l) = pc;
    }
}
```

Table 36: Reverse a linked list: recursive version.

5.13 Count the word frequency

Given a vector of words we want to create a function that return a list of the minimal length such that each cell contains a *copy* of a different word contained in the array and the number of its occurrences in the array. Each cell of the list must contain a different word. The order of the cells in the list is arbitrary.
We want to solve the exercise following the steps:

A Define the types for the list;

B Define function `void prepend(tlist *l, char * str, int val)` the prepend a cell to the list `*l` which will contain a copy of the string `str` and a number of occurrences `val`. We can assume that all the `malloc` calls will allocate the dynamic memory successfully. The space allocated for the strings must be as small as necessary;

C Define the *recursive* function `tlist is_present(tlistl, char *mystr)` that returns a pointer to the cell containing a copy of `mystr` if present, otherwise it returns `NULL`;

D Define the function `void create_list(tlist *l, char *vetstr[], int size)` that given the vector of strings `vetstr` return the list according the requirements of the exercise;

E Define the function `void destroy_list(tlist *l)` that frees the heap memory containing the list and set the pointer `*l` to `NULL`;

Solution. The solution to the exercise is just a combination of simple functions. We need to pay attention to the fact that the list must contain copies of those contained in the vector and hence we must allocate dynamically in the heap. Moreover the comparison cannot be done by comparing the address of the first char, but by using the function `strcmp` from the standard library. Moreover, when the list is destroyed we must pay attention to free the memory allocated for the string and for the cell in the correct order. The solution is shown in Table 37.

5.14 Delete a list from the last repetition

Given a list of integers, write the following function:

```
void remove_elements(tlist *l);
```

that deletes from the dynamic memory all the list cell from the last two consecutive numbers (not included) to the end of the list. Examples:

- $4 - 4 \Rightarrow$ empty list
- $5 - 7 - 9 - 9 - 9 - 3 - 4 - 4 - 4 - 4 - 5 - 6 - 3 \Rightarrow 5 - 7 - 9 - 9 - 9 - 3 - 4 - 4 - 4 - 4$
- $6 \Rightarrow$ empty list

Solution. We propose two solutions of the problem. One is conceptually simple but requires to remove and recreate a cell since at the moment of the deletion we do not know if previously there is another cell containing the same information. The second solution is more complicated but also mote elegant. The solutions are shown in Table 38.

5.15 Alternating merge of two lists

Given two lists `l1` and `l2`, write the following function:

```
tlist merge(tlist l1, tlist l2);
```

that returns a list with the elements of `l1` and `l2` alternating. If one of the two lists is longer than the other, then all its exceeding elements will be added consecutively.

Solution. If we accept to destroy the input list `l1` and `l2`, then the solution becomes trivial as shown in Table 39. A little bit longer is the solution that duplicates the lists but it is conceptually identical.

5.16 Binary numbers in lists

In this exercise we consider lists of integers whose values can be 1 or 0 (more compact representations could be possible, but for the sake of simplicity we maintain this assumption). The data structure is defined as:

```
struct cell{
   int info;
   struct cell *next;
};

typedef struct cell *tlist;
```

We address the following problems:

```c
/*Bullet A*/
struct cell{
   int rep;
   char* str;
   struct cell *next;
};

typedef struct cell *tlist;


/*Bullet B*/
void prepend(tlist *l, char *str, int val) {
   tlist newcell = (tlist) malloc(sizeof(struct cell));

   newcell->next = *l;
   *l = newcell;
   newcell->str = (char*)malloc(sizeof(char)*(strlen(str)+1));
   strcpy(newcell->str, str);
   newcell->rep = val;
}

/*Bullet C*/
tlist is_present(tlist l, char *mystr) {
   if (!l)
      return NULL;
   else {
      if (!strcmp(l->str, mystr))
         return l;
      else
         return is_present(l->next, mystr);
   }
}

/*Bullet D*/
void create_list(tlist *l, char *vetstr[], int size) {
   int i;
   tlist where;
   *l = NULL;
   for (i=size-1; i>=0; i--) {
      where = is_present(*l, vetstr[i]);
      if (!where)
         prepend(l, vetstr[i], 1);
      else
         (where->rep)++;
   }
}

/*Bullet E*/
void destroy_list(tlist *l) {
   if (*l) {
      destroy_list(&(*l)->next);
      free((*l)->str);
      free(*l);
      *l = NULL;
   }
}
```

Table 37: Count the occurrences of a word in a vector.

```
void remove_elements(tlist *l) {
   if (*l) {
      if ((*l)->next) {
         int copy = (*l)->next->info;
         remove_elements(&(*l)->next);
         if ((*l)->next == NULL) {
            if ((*l)->info == copy) {
               (*l)->next = (tlist)malloc(sizeof(struct cell));
               (*l)->next->next = NULL;
               (*l)->next->info = copy;
            }
            else {
               free(*l);
               *l=NULL;
            }
         }
      }
      else {
         free(*l);
         (*l) = NULL;
      }
   }
}
```

```
void remove_elements(tlist *l) {
     if (*l) {
      if (!(*l)->next) { /*List with one element*/
         free(*l);
         *l = NULL;
      }
      else { /*List with at least two elements*/
         int foundpair = 0;
         tlist *pc = l;
         tlist prev = *l;
         tlist cur = prev->next;
         while (cur && prev->info == cur->info) {
            foundpair = 1;
            pc = &prev->next;
            prev = cur;
            cur = cur->next;
         }
         remove_elements(&prev->next);
         if (!prev->next && !foundpair) {
            free(prev);
            *pc = NULL;
         }
      }
   }
}
```

Table 38: Two solutions for the problem of deleting the cells of a list from the last two equal elements to the end of the list.

```
tlist mix(tlist l1, tlist l2) {
   if (!l1)
      return l2;
   else {
      l1->next = mix(l2, l1->next);
      return l1;
   }
}
```

Table 39: Alternating lists

1. Compress a list. Implement function `void compress(tlist l, int **vect, int *size)` that encodes a list in a dynamic vector as follows: a value n in the i-th position means that the i-th sequence of identical contiguous bits has length $|n|$. If $n > 0$ then it is a sequence of 1s, otherwise 0s. For example, for a list whose elements are:

   - 0-1-1-0-0-0-1-0

   The output is a vector of size 5 whose entries are $[-1, 2, -3, 1, -1]$.

2. Convert the list into an integer. If the list represents a binary number whose less significant bit is in the head of the list, write the *recursive* function `unsigned int convert(tlist l);` that converts the list into an equivalent integer. The empty list encodes the value 0.

3. Maximum contiguous sequence. Implement the function `tlist maxseq(tlist l);` that returns the point to the first cell of the longest sequence of identical contiguous bits. In the case of sequences with identical length, we choose the sequence which is closest to the head of the list. For empty list, we return `NULL`.

In the whole exercise you can assume that the dynamic memory allocation will not fail.

Solution. For the first exercise we divide the solution algorithm in two phases: first we count the number of sequences with identical bits. Then, after the allocation of the vector in dynamic memory, we fill the vector with the correct values. We use variable `lastvisited` to store if the sequence we are visiting consists of 1s o 0s. The second point can be solved in an identical manner than that proposed for Exercise 5.1. The third point is solved by storing the length and the starting position of the maximum sequence found up to a certain position in the list in the variables `maxsize` and `longest`, respectively. The solution is shown in Table 41.

5.17 Erase contiguous identical cells in a list

Write the recursive function:

```
void erase(tlist *l);
```

that, given a list of integers pointed by `l`, modifies the list according to the following rules:

- If the list contains a sequence of identical cells with even length then the sequence is completely erased
- If the list contains a sequence of identical cells with odd length then the sequence is replaced by a single element whose value is the value of the elements of the eliminated sequence.

```
struct cell{
   int info;
   struct cell *next;
};

typedef struct cell *tlist;

void compress(tlist l, int **vect, int *size) {
   tlist pc = l;
   int lastvisited = -1;
   int length = 0;
   int seqn = 0;

   /*compute the size of the vector*/
   *size = 0;
   while (pc) {
      if (pc->info != lastvisited) {
         *size = *size +1;
         lastvisited = pc->info;
      }
      pc = pc->next;
   }

   *vect = (int*) malloc ((*size) * sizeof(int));

   /*fill the vector*/

   lastvisited = -1;
   while (l) {
      lastvisited = l->info;
      length = 1;
      l = l->next;
      while (l && l->info == lastvisited) {
         length++;
         l=l->next;
      }
      (*vect)[seqn] = length * ((lastvisited == 1) ? 1 : -1);
      seqn++;
      length = 0;
   }
}


int convert(tlist l) {
   if (l)
      return convert(l->next)*2 + l->info;
   else
      return 0;
}
```

Table 40: Binary lists /A.

```
tlist maxseq(tlist l) {
   tlist longest = l;
   tlist cur;
   int maxsize = 0;
   int cursize = 0;
   while (l) {
      cur = l;
      cursize = 1;
      while (l && l->next && l->info == l->next->info) {
         cursize++;
         l = l->next;
      }
      if (cursize > maxsize) {
         maxsize = cursize;
         longest = cur;
      }

      if (l) {
         l = l->next;
      }
   }
   return longest;
}


int main() {
   tlist l = NULL;
   tlist longest = NULL;
   int *vet;
   int size;
   int i;

   /*initialize the list*/

   compress(l, &vet, &size);

   for (i=0; i<size; i++)
   {
      printf ("%d\n", vet[i]);
   }

   printf("size: %d\n", size);

   printf("Conversion in decimal: %d\n", convert(l));

   longest = maxseq(l);

   printf("\n Longest first cell %d \n ", longest->info);

   free(vet);

   return 0;
}
```

Table 41: Binary lists /B

```
void erase(tlist *l) {
   if (*l && (*l)->next) {
      if ((*l)->info == (*l)->next->info) {
         tlist supp = (*l)->next->next;
         free ((*l)->next);
         free ((*l));
         *l = supp;
         erase(l);
      }
      else
         erase(&(*l)->next);
   }
}
```

Table 42: Erase sequences in a list of integers.

```
struct nodeDL {
    int info;
    struct nodeDL *prev;
    struct nodeDL *next;
};
typedef struct nodeDL *tlistdl;
```

Table 43: Type definition for a doubly linked list of integers.

For instance, the following list:

$$10, 10, 10, 10, 16, 14, 12, 12, 12, 12, 12, 10$$

becomes:

$$16, 14, 12, 10$$

Solution. The idea is to erase pairs of contiguous elements with the same value. The solution is shown in Table 42.

6 Doubly linked lists

In this section we propose exercises on doubly linked lists of integer numbers whose type definition is given in Table 43. Each node of the list contains a field `info` (the stored information) and two pointers, one to the previous node and one to the next node in the list.

6.1 Append an element to a list

Write a function that, given a doubly linked list and an integer, appends to the tail of the list a new node containing the provided number. The function returns 1 if the allocation in the heap of the new node succeeds, 0 otherwise.

Solution. The solution is pretty straightforward: we just need to distinguish between the case of the empty list, where we have to set the pointer to the head, and the other cases, where we have to update the `next` field of the tail so that it points to the new node. The solution is shown in Table 44.

```
int append(tlistdl *head, tlistdl *tail, int info) {
    tlistdl node = malloc(sizeof(struct nodeDL));
    if (node == NULL) {
        return 0;
    }
    node->info = info;
    node->next = NULL;
    node->prev = *tail;
    if (*head == NULL) {
        *head = node;
    } else {
        (*tail)->next = node;
    }
    *tail = node;
    return 1;
}
```

Table 44: Append a node to a doubly linked list.

6.2 Deletion of an element from the list

Write a function that removes from a doubly linked list the first occurrence of the number provided as input to the function.

Solution. First we traverse the list, starting from the head, searching for the node n to be removed, if any. If n is the head of the list, we need to modify the pointer provided by the user, otherwise we must change the `next` pointer of the element that precedes n so that it points to the element that follows n. A similar reasoning applies if n is the tail of the list. The solution is shown in Table 45.

6.3 Check if the list is palindrome

Write a function that checks whether a doubly linked list is palindrome or not.

Solution. We iterate over the list using both pointers `head` and `tail` and stop as soon as one of the following condition occurs:

- `head == tail` occurs when the list is either empty or contains an odd number of elements and it is palindrome;
- `head->prev == tail` holds when the list contains an even number of elements and is palindrome;
- `head->info != tail->info` is verified when the list is not palindrome.

The solution is shown in Table 46.

7 Exercises on binary trees

7.1 Delete a tree

Write a function that removes from the heap a binary tree of integer.

```
void remove(tlistdl *head, tlistdl *tail, int info) {
    tlistdl iter = *head;
    while (iter != NULL && iter->info != info) {
        iter = iter->next;
    }
    if (iter != NULL) {
        if (iter == *head) {
            *head = iter->next;
        } else {
            iter->prev->next = iter->next;
        }
        if (iter == *tail) {
            *tail = iter->prev;
        } else {
            iter->next->prev = iter->prev;
        }
        free(iter);
    }
}
```

Table 45: Remove a node from a doubly linked list.

```
int palindrome(tlistdl head, tlistdl tail) {
    while (head != tail && head->prev != tail && head->info == tail->info) {
        head = head->next;
        tail = tail->prev;
    }
    return head == tail || head->prev == tail;
}
```

Table 46: Check whether a doubly linked list is palindrome.

Solution. We propose the recursive solution to the problem in Table 47. Notice that, similarly to the library function `free`, the pointer specified as actual parameter is not changed since it is passed by value.

8 Tests

8.1 Questions

1. Given the following declaration: `int *a;` what can be said about `&a`?

   (a) It is an expression of type `int`

   (b) It is a variable of type `int`

   (c) It is an expression of type `int**`

   (d) It is a variable of type `int**`

2. Given the following function:

   ```
   void swap(int *a, int *b) {
      int c = *a;
      *a = *b;
      *b = c;
   }
   ```

   and the following call:

   ```
   int *k1, *k2;
   swap(k1, k2);
   ```

   Which is the correct choice?

   (a) The listing does not compile

   (b) The listing compiles but the usage of pointers is wrong

   (c) The listing exchanges the values of pointers `k1` e `k2`

   (d) The listing exchanges the values of the variables pointed by `k1` e `k2`

3. Given the following declarations `int a; int *b;` the writing `*(b+a)` denotes:

   (a) It is a variable with type `int`

   (b) It is not correct

   (c) It is an expression with type `int*` but not a variable

   (d) It is a variable with type `int*`

4. Given the declarations `int a; int *b;` the writing `(*b)+a` denotes:

   (a) A variable of type `int`

   (b) An expression of type `int` but not a variable

   (c) An expression with type `int*` but not a variable

   (d) A variable with type `int*`

5. Given the following declaration `int myvect[100];`, then the writing `myvect` in an assignment is:

   (a) A variable with type `int*`

   (b) An expression with type `int*` equivalent to `&myvect[0]`

   (c) A variable of type `int` equivalent to `myvect[0]`

   (d) Is wrong

6. Given the following listing:

```
char* mystring() {
   char news[100];
   news[0]='\0';
   return news;
}
```

   Which of he following is correct?

   (a) It does not compile

   (b) It compiles the it does not work properly because it accesses to position 0 of an array

   (c) It compiles but the instruction `return` causes a type casting

   (d) It compiles but the instruction `return` gives the address of a memory location that will be freed when the execution of the `mystring` terminates

7. Given the following function:

```
int foo(int a, int b) {
   if (a == b)
      return a;
   else
      return (a>b) ? foo(a-b, b) : foo(a, b-a);
 }
```

   What is the value of the expression `foo(18, 60)`?

   (a) 6

   (b) 60

   (c) 18

   (d) The recursion does not terminate

8. Given the following function:

```
int* mine(int a) {
   return &a;
}
```

   and the following call:

```
int k1=8;
int *pt;
pt = mine(k1);
```

   Which of the following propositions is correct?

(a) `pt` points to variable `k1`

(b) `pt` points to a memory location which is not associated with a variable

(c) The listing does not compile because of a type error

(d) The value of `pt` is 8

9. Given the following declarations `float *a; float *b;` the writing `b-a` denotes:

   (a) An expression with type `float*` but not a variable

   (b) An expression with type `float` but not a variable

   (c) An expression with type `int` but not a variable

   (d) A variable with type `float*`

10. The C language supports the parameter passing:

    (a) by value

    (b) by reference

    (c) both by value and by reference

    (d) not by value nor by reference

11. Given the following declaration `double* myvect[100];`, how can you interpret this data structure?

    (a) A vector with 100 cells of type `double`

    (b) A vector with 100 pointers to `double` memory locations

    (c) A pointer to a vector of cells with type `double`

    (d) A pointer to a vector of pointers to `double` memory locations

12. Given the following declarations `float a[100]; int b=20;`, the writing `*(a+b)` is equivalent to:

    (a) It is not correct

    (b) `&a[0]+b`

    (c) `*a[b]`

    (d) `a[b]`

13. Given the following code:

```
int i, a = 1;
for (i=0; i<10; i++)
  a = a*2;
```

    Which is the value of variable `a` after the cycle termination?

    (a) 512

    (b) 20

    (c) 1024

(d) 2

14. Given the following function:

```
void foo(int a, int *b) {
  *b = a;
}
```

and the follwoing function call:

```
int k1=10, k2=20;
foo(k1, &k2);
```

Which is the value of variables `k1` and k2?

(a) 10 and 20

(b) 10 and 10

(c) 20 and 10

(d) The code is wrong because pointers are used uncorrectly

15. Given the following declarations: `float a; float *b; float **c=&b;`. We want to assign value 10.2 to variable `a` with the following instruction `**c = 10.2`. Before doing this, which other instruction must be executed?

(a) `*a = b;`

(b) `b = a;`

(c) `&b = a;`

(d) `b = &a;`

16. Given the array declaration `int a[100];` which of the following writings is equivalent to `a[3]`?

(a) `a+3`

(b) `&(a+3)`

(c) `*(a+3)`

(d) `**(a+3)`

17. Given the following code:

```
float* myfunction(float x){
  float var = 12.0 + x;
  return &var;
}
```

Which of the following propositions is true?

(a) The code does not compile due to an error on types

(b) The code does not compile due to a syntax error

(c) The code compiles but the return statement returns the address of a local variable that will be deallocated just after the termination of the function call

(d) The function assigns the value 12.0 to the argument

18. Let `tlist` be the type of a pointer to a struct defined for a list of integers and consider the following function:

```
int foo(tlist l) {
    if (!l)
        return 0;
    return (l->info % 2 == 0) ? l->info + foo(l->next) : foo(l->next);
}
```

Given the list $3-5-2-0-1-4$ in input, what is the value computed by the function?

(a) 3

(b) 4

(c) 6

(d) None of the previous ones

19. Let `tlist` be a pointer to a struct defined for a list of integers and consider the following function:

```
void remove(tlist l) {
    if (l) {
        tlist l1 = l->next;
        free(l);
        l = l1;
    }
}
```

What is the outcome of the function on the list specified as actual parameter?

(a) It removes the first cell of a non-empty list

(b) The list specified as actual parameters will have a dangling pointer

(c) It creates garbage in the heap, but the list is not modified

(d) It contains a type error

20. In C, the evalation of expressions occurs at...

(a) compile time

(b) execution time

(c) linking time

(d) expressions can never be evaluated

21. Let `head` be a variable of type `tlist` that points to the cell of a list containing at least two elements. Why can't we say that `head + 1` is the address of the cell following the one addressed by `head`?

(a) Because `head + 1` is not a valid expression

(b) Because `head + 1`, although being a valid expression, is not of type `tlist`

(c) Because the cells of the list can be allocated non contiguously in main memory

(d) Because `head + 1` increases the value stored in the cell pointed by `head`

22. Consider the following snippet of code:

```
int *a, *b;
int vett[10];
a = &vett[1];
b = vett + 2;
```

What is the type and the value of the expression `b - a`?

(a) The type is `int` and the value is the size of an integer

(b) It is not a valid expression

(c) The type is `int*` and the value is the difference of the pointers

(d) The type is `int` and the value is 1

23. Consider the following snippet of code:

```
int *a, *b;
int vett[10];
a = &vett[1];
b = vett + 2;
```

What is the type of the expression `a + b`?

(a) `int`

(b) It is not a valid expression

(c) `int*`

(d) `void`

24. Let `tlist` be the type of a pointer to a cell of a list of `int`. Given the following function:

```
int foo(tlist l) {
   return (l && l->info > 0) ? 1+foo(l->next): 0;
}
```

what does it return when applied to the list $3-5-2-0-1-0$?

(a) 3

(b) 4

(c) 6

(d) None of the above

25. Which of the following is true for the C language?

(a) The type of the expressions is decided during the execution

(b) The type of the expressions is decided statically

(c) The value of the expressions is decided statically

(d) Each expression is associated with an identifier

26. Give the declarations `int v1[10]; int v2[10];`, what is the value of the expression `v1 == v2`?

(a) The encoding of `true`

(b) The encoding of `false`

(c) It depends on the values contained in the arrays

(d) It depends on the two values contained in the first cells of the arrays

27. Given the declaration `int **p`, what is the type of `*p`?

(a) `int***`

(b) `int**`

(c) `int*`

(d) `int`

28. Given the declarations `int *a, *b` which of the following expressions is not valid?

(a) `a-b`

(b) `*a + *b`

(c) `(&a) + (*b)`

(d) `a+b`

29. Given the following snippet of code:

```
int i,j;
int acc = 0;
for (i = 0; i < 100; i++)
   for (j=0; j < 4; j++)
      if ( (i+j)%2 == 0) acc++;

printf(''%d'', acc);
```

what does it print?

(a) 100

(b) None of the other answers

(c) 200

(d) 400

30. Given the following snippet of code:

```
int foo(tlist l) {
   return (l && l->next && l->info < l->next->info) ? 1 + foo(l->next) : 0;
}
```

which is the value return for the call on the list 3-5-9-0-1-0?

(a) 2

(b) 3

(c) 5

(d) None of the other values

31. In the C language the actual parameters are always:

(a) Memory addresses

(b) Variables

(c) Ambients

(d) Expressions

32. Given the following declaration `char *v1[10]` which of the following expressions is equivalent to `v1[2][7]`?

    (a) *(v1+2+7)

    (b) *(*(v1+2)+7)

    (c) *(v1+2)+7

    (d) *v1 + 7

33. Given the following function:

```
void foo(int **vet) {
   int i;
   *vet = (int*) malloc(100*sizeof(int));
   for (i=0; i<100; i++)
      (*vet)[i] = 0;
}
```

    and the main:

```
int main() {
   int **vet;
   foo(vet);
   ...
}
```

    Which of the following statements is correct?

    (a) The function call causes a type mistmatch error at compiling time

    (b) The function call causes a type mistmatch error at execution time

    (c) The code is correct and in the `main` it will be possible to use the memory allocated and initialised by `foo`

    (d) There are not error for the types the after the execution of function `foo` the program creates garbage in the heap memory

8.2 Solutions

1. c
2. b - function `swap` uses the operator * on pointers `a, b` which contains the same values of `k1, k2` which are not initialized.
3. a
4. b
5. b
6. d - local parameters are automatically deallocated when the function execution terminates
7. a - the function computes the greatest common divisor of two positive integers

8. b - in C the parameters are passed by copy
9. c
10. a
11. b
12. d
13. c
14. b
15. d
16. c
17. c
18. c
19. b
20. b
21. c
22. d
23. b
24. a
25. b
26. b
27. c
28. d
29. c
30. a
31. d
32. b
33. d

```
struct bintree{
  int info;
  struct bintree *left;
  struct bintree *right;
};

typedef struct bintree *Bintree;

void delete_tree(Bintree bt) {
  if (bt) {
    delete_tree(bt->left);
    delete_tree(bt->right);
    free(bt);
  }
}
```

Table 47: Free the memory occupied by a binary tree.