Computer Architecture
Preparation for the Midterm Exam
Solutions

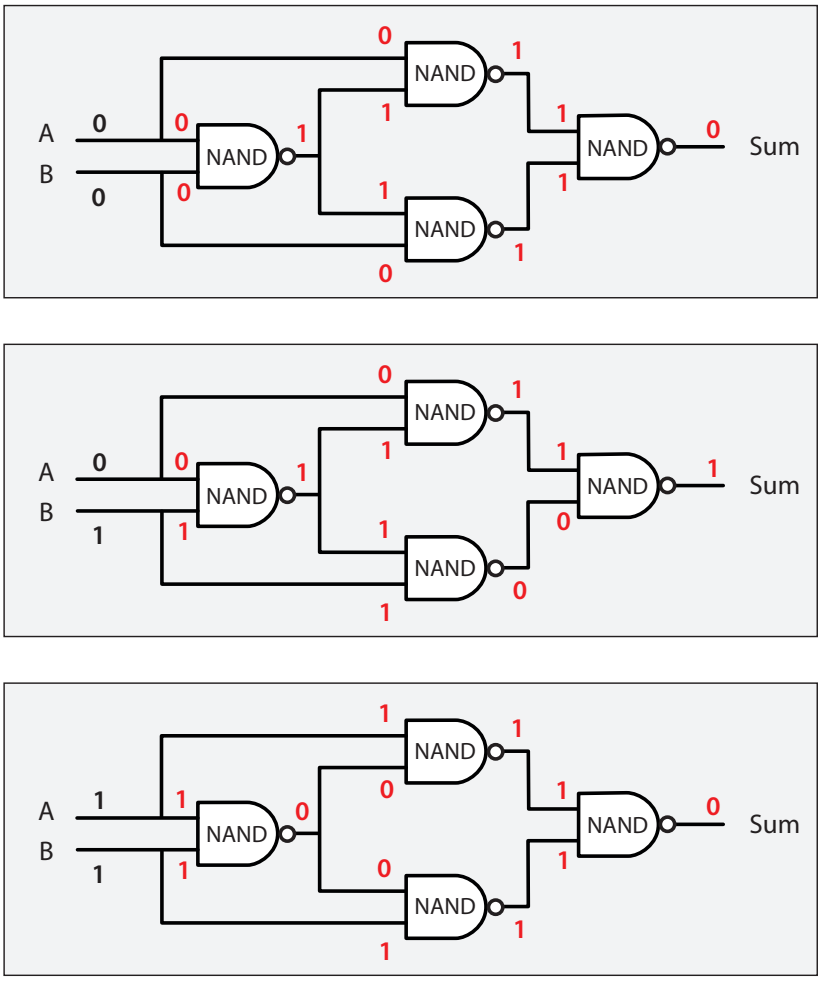
Paul Mellies

Exercice 1

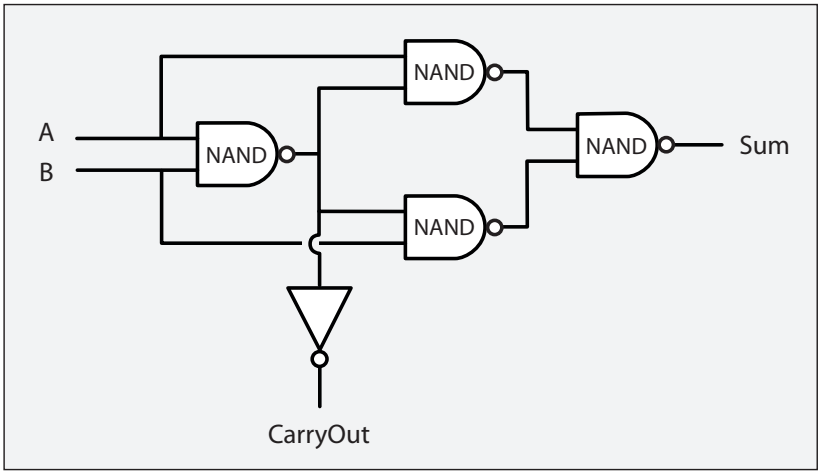
§1a. As explained during the course, the following circuit is the specific part of the half-adder designed to compute the sum of the two inputs A and B. Each of three diagrams below corresponds to one of the four possible inputs

$$A = 0\,,\ B = 0 \qquad A = 0\,,\ B = 1 \qquad A = 1\,,\ B = 1$$

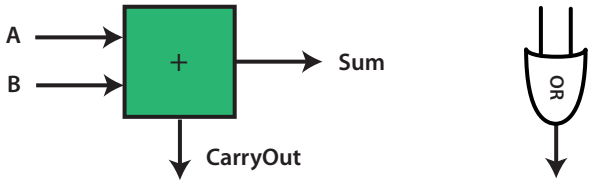
Indicate for each of the three inputs A and B the value of *every wire* in the circuit, and in particular the value of the output Sum:



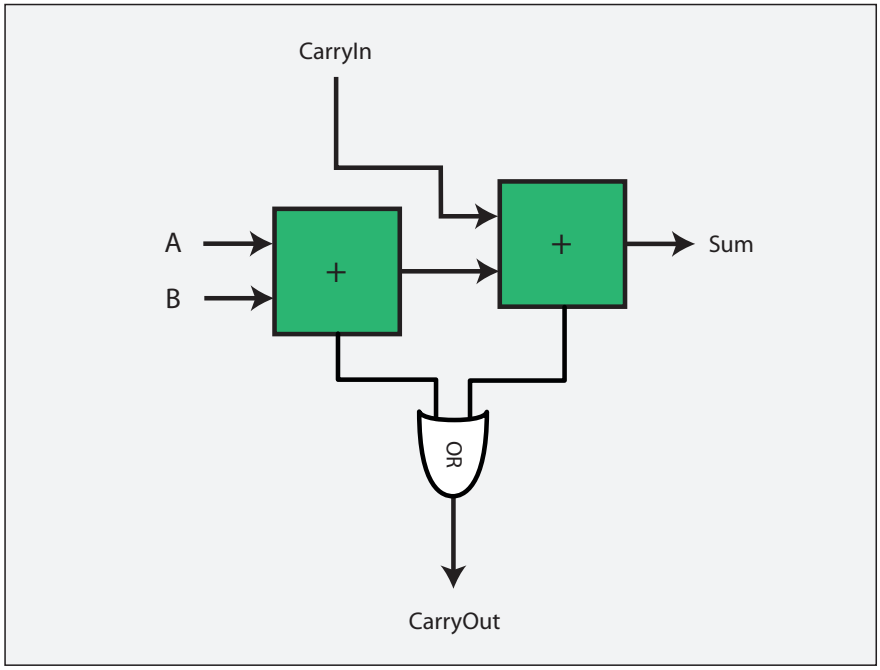
§1b. Complete the circuit below in order to obtain a half-adder:



§1c. Recall that an half-adder and a OR gate are depicted as follows:



Using this notation, explain with a diagram how to construct a full adder using two half-adders together with an OR gate:



Exercice 2

§2. A flying saucer crashes in a Nebraska cornfield. The FBI investigates the wreckage and finds an engineering manual containing an equation in the Martian number

system:

$$325 + 42 = 411$$

If this equation is correct, how many fingers would you expect Martians to have?

▷ The correct answer is 6. Can you explain why?

Exercise 3.

Suppose that you are a collector of old computers and that you bought a machine whose signed and unsigned integer are represented using 8 bits integers.

§3a. What are the maximal and minimal unsigned integers which can be represented as an 8-bit integer of your machine?

▷ The minimal unsigned integer is 0 while the maximal unsigned integer is $2^8 - 1$ which is 255.

§3b. What are the maximal and minimal signed integers (two's complement) which can be represented as an 8-bit integer of the machine?

▷ The minimal unsigned integer is -2^7 which is equal to -128 while the maximal unsigned integer is $2^7 - 1$ which is equal to 127.

§3c. Suppose that you use the signed add instruction of the machine in order to add the signed integer 102 and the signed integer 88. Could you describe what happens? What would happen if you used the unsigned add instruction instead? And what would be the result in that case?

▷ Since $102 + 88 = 190$, adding 102 and 88 produces an integer larger than 127. Adding the two positive integers 102 and 88 using the signed `add` instruction will thus produce an overflow exception. On the other hand, adding the two numbers using the MIPS unsigned `addu` instruction will produce no overflow error. The result of the addition is the number 190 seen as an unsigned number, and the number -66 seen as a signed number.

Exercise 4.

§4a. Translate the following decimal numbers into binary numbers of length 8 bits:

15	82	97	114
▷ 00001111	01010010	01100001	01101010

§4b. Translate the same decimal numbers into hexadecimal numbers

15	82	97	114
▷ 0F	52	61	6A

§4c. Translate the decimal numbers into signed binary numbers (two's complement) of length 8 bits:

−1	15	− 82	− 97	− 114
▷ 11111111	11110001	10101110	10011111	10010110

§4d. Describe a simple recipe (or algorithm) to translate a positive number represented as a signed binary number into its opposite number, also represented as a signed binary number on 8 bits.

▷ Start by computing the one's complement by replacing every 0 by a 1 and every 1 by a 0 in the original binary number, and then increment the resulting binary number with 1.

§4e. Indicate whether the same recipe (or algorithm) works in order to translate a negative number (typically -5) into its absolute value (typically 5).

▷ If we start from 5 represented as the binary number 00000101, then the signed number −5 is represented in two's complement as the binary number 11111011. Then, if one applies the algorithm to the binary number 11111011, one obtains 00000101 and thus recovers the original positive number 5.

§4f. Explain what one means by « sign extension » from 8 bits to 16 bits.

▷ Sign extension is described in slides 46 and 47 of Lesson 3. The idea is that one extends a signed number of 8 bits into a signed number of 15 bits without changing its value. To that purpose, one needs to extend the original number with 8 bits of value 0 when the number is positive, and with 8 bits of value 1 when the number is negative. The fact that the original number of eight bits is positive or negative can be detected by looking that its most significant bit, which is the bit number 7.

Exercise 5.

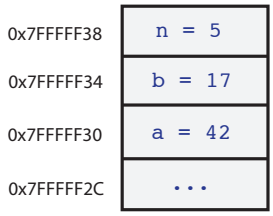
§5a. Consider the C program below

```
1.      void swap(int x, int y){
2.          int tmp;
3.          tmp = y;
4.          y = x;
5.          x = tmp;
6.      }
7.
8.      int main()
9.      {
10.         int a = 42;
11.         int b = 17;
12.         int n = 5;
13.
14.         printf("a=%d b=%d \n", a , b);
15.         swap(a,b);
16.         printf("a=%d b=%d \n", a , b);
17.     }
```

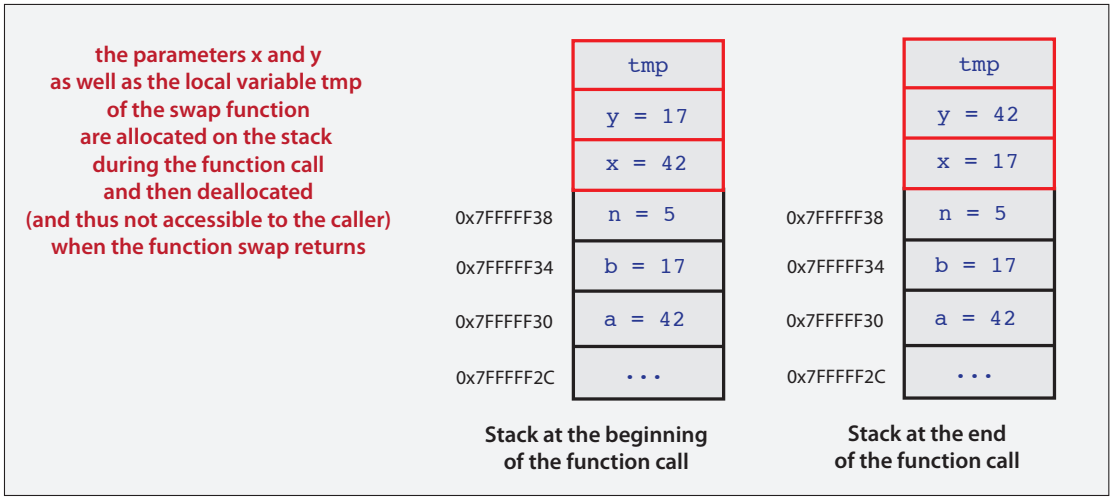
and indicate what the `main` program prints as output

```
a = 42   b = 17
a = 42   b = 17
```

§5b. [optional] Suppose that the stack has the following shape when the function `swap` is called by the `main` function:



Explain how many local variables are allocated during the function call, and describe the shape of the stack at the beginning and at the end of the execution of the `swap` function, before it returns.



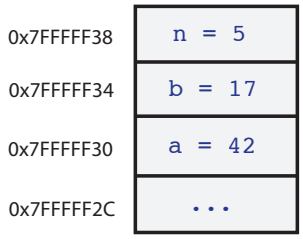
§5c. Consider now the C program below where the `swap` function has been replaced by the `swapbyptr` function:

```
1.    void swapbyptr(int *xptr, int *yptr){
2.        int tmp;
3.        tmp = *yptr;
4.        *yptr = *xptr;
5.        *xptr = tmp;
6.    }
7.
8.    int main()
9.    {
10.       int a = 42;
11.       int b = 17;
12.       int n = 5;
13.
14.       printf("a=%d b=%d \n", a , b);
15.       swapbyptr(&a,&b);
16.       printf("a=%d b=%d \n", a , b);
17.   }
```

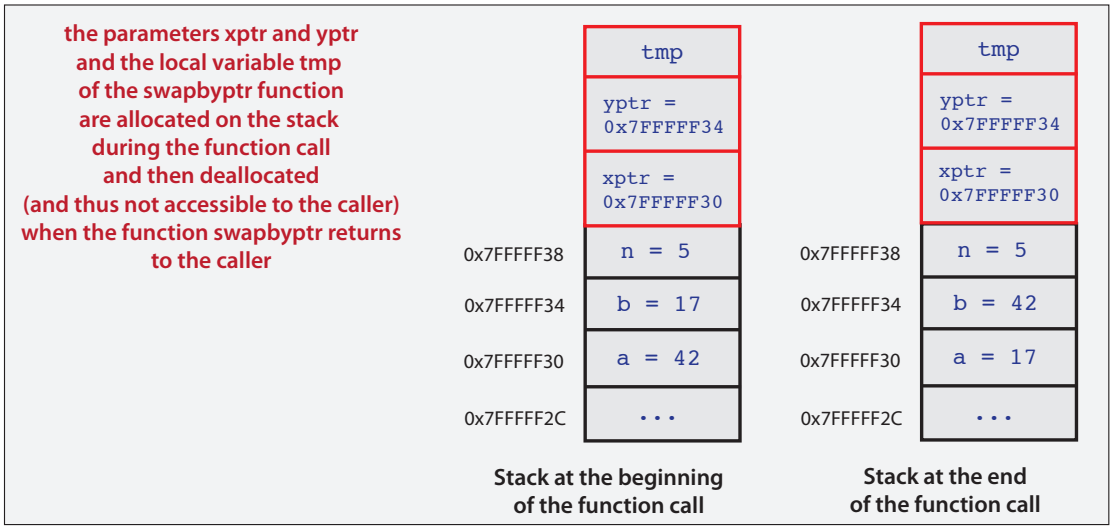
Indicate below what the `main` program prints as output in that case

a = 42 b = 17
a = 17 b = 42

§5d. [optional] Suppose that the stack has the same shape as previously:



when the function `swapbyptr` is called by the `main` function. Explain how many local variables are allocated during the function call, and describe below the shape of the stack at the beginning and at the end of the execution of the `swapbyptr` function:

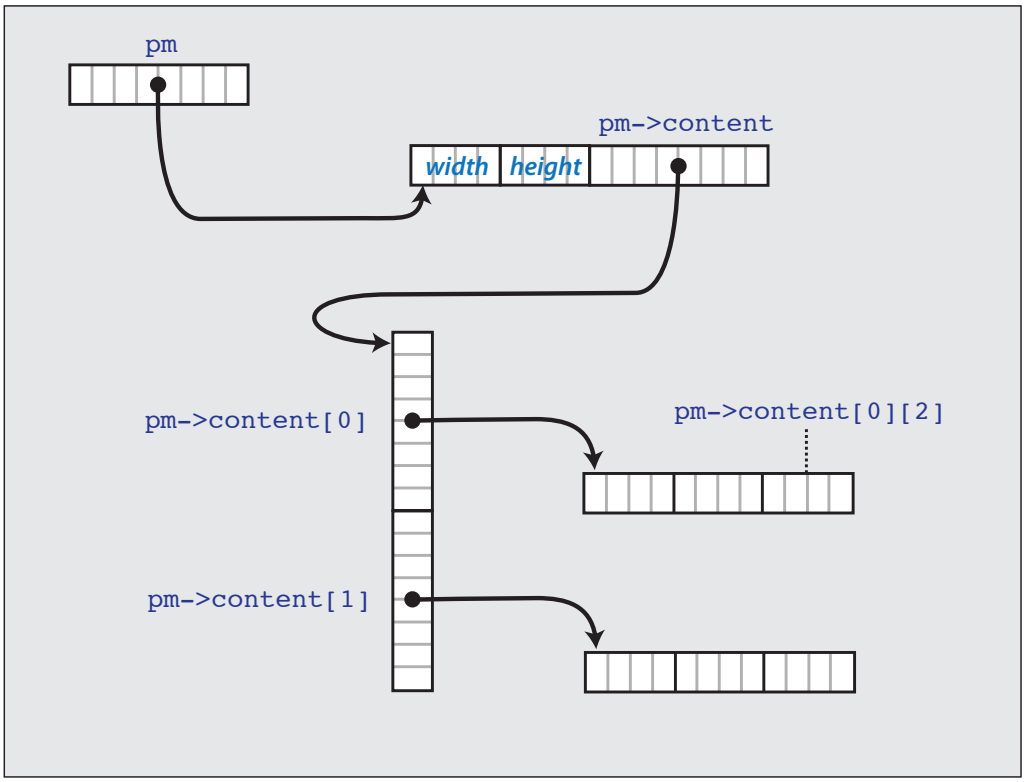


Exercise 6.

As explained during the course, one declares the following struct type `matrix` in order to implement Java-like matrices in memory.

```
struct matrix {
    int width;
    int height;
    int **content;
};
```

§6a. Can you draw a picture explaining how a matrix of width 3 and height 2 would be represented in memory.



§6b. Suppose that the struct `matrix` is of size 16 bytes. Give a simple formula expressing the size of allocated memory for a matrix with width w and height h.

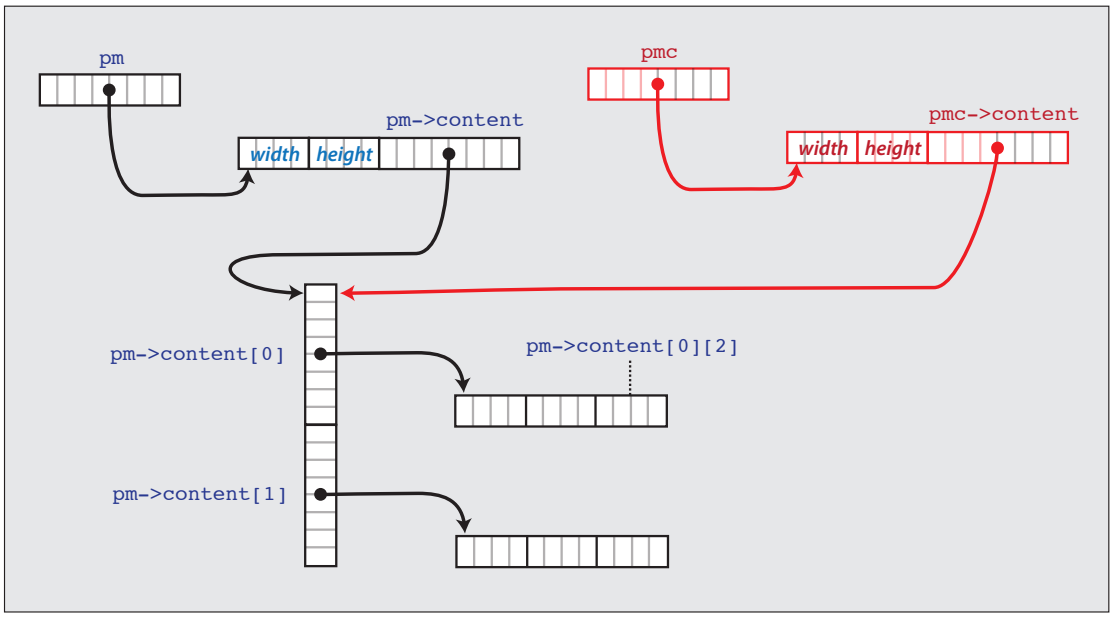
▷ The formula describing the size of the allocated memory is $16 + 8 \times h + 4 \times h \times w$

§6c. Explain why the C function `clone_matrix` below whose purpose is to clone a Java-like matrix and to return a pointer to its clone matrix will not work as expected.

```
1.    struct matrix *clone_matrix (struct matrix *pm){
2.      struct matrix *pmc;
3.      int i, j;
4.
5.      // (1) allocate the structure of the clone
6.      pmc = malloc (sizeof(struct matrix));
7.      // (2) copies the content of the fields
8.      pmc -> width = pm -> width;
9.      pmc -> height = pm -> height;
10.     pmc -> content = pm -> content;
11.
12.   return pmc; // return the address of the clone matrix
13.   }
```

▷ This is the diagram which describes what one obtains in memory when the `clone_matrix` function returns. Note that the cloning is not performed properly since the content of the original matrix `pm` has not been copied in memory. In particular, if the value of one element of the original matrix `pm` is altered, the same will be

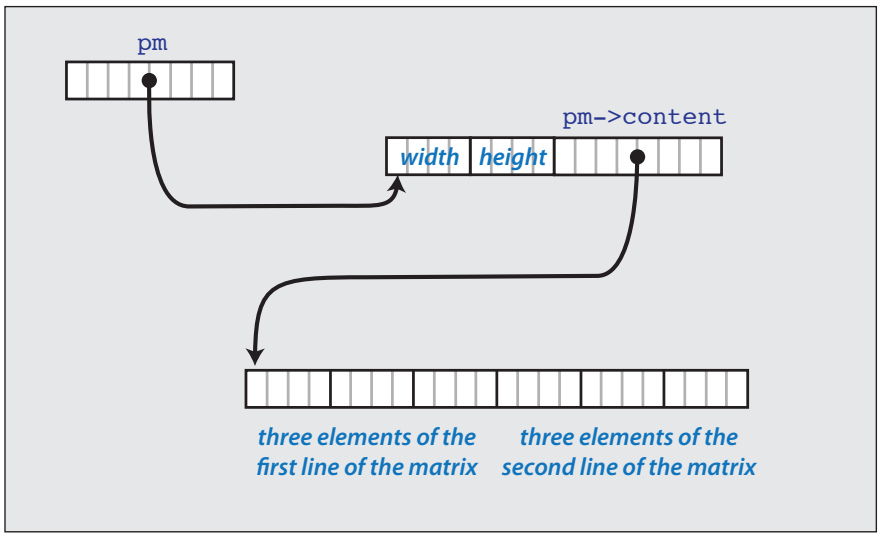
true for the matrix `pmc`. This is not what is expected of a function cloning a matrix `pm` into a matrix `pmc`.



§6d. Describe with a drawing the other possible representation of matrices in memory (also discussed during the course) with struct `matrix` defined in that case as

```
struct matrix {
    int width;
    int height;
    int *content;
};
```

Suppose as in §6b that the struct `matrix` is of size 16 bytes. Give a simple formula expressing the size of allocated memory for a matrix with width w and height h in that representation.



▷ The formula describing the size of the allocated memory is $16 + 4 \times h \times w$

§6e. Explain the advantages and disadvantages of representing matrices of integers as Java-like matrices instead of the alternative representation discussed in §6d.

▷ The representation as a Java-like matrix enables one to manipulate large matrices (with large height and width) while this is difficult with the naive representation. The reason is that it is much easier to allocate h arrays of length $4 \times w$ than a single array of length $4 \times w \times h$. On the other hand, the access to the elements of the matrix is faster in the naive representation. The naive representation is thus appropriate to manipulate small matrices in a fast way.

Exercise 7.

§7a. Explain in a few words the difference between an architecture and a microarchitecture.

▷ The architecture is another name for the machine language and instruction set of the microprocessor. It is what the programmer can see, while the microarchitecture is the electronic device which implements the language. In the case of a Princeton architecture, the microarchitecture is typically executing the code in a parallel way, while the programmer believes that it is executed in a sequential way.

§7b. Explain in a few words the notion of « memory leak » encountered in our study of dynamic allocation in the programming language C.

▷ Memory leaks typically happens when the code forgets to free the regions of memory which were allocated on the heap using the `malloc` function. These regions of memory remain allocated but are not accessible any more to the program. They are thus useless, and one can thus think of them as memory leaks. If there are too many of such memory regions allocated but not accessible any more, then the system may not find at some point any memory left for the program to use.

§7c. [optional] Explain in a few words why the `free` function does not need to take as parameter the size in memory of the deallocated block, but only its pointer.

▷ This is explained in slide 33 of the optional supplementary lesson 8. The reason is that the `malloc` function stores the size of the allocated region of memory at the same time as it allocates that region of memory in the heap. In that way, the `free` function can find this piece of information and deallocate the appropriate amount of memory.