Exercise 1 (40 points)

In this exercise, you will implement a binary max-heap. A binary max-heap is a binary tree with the following properties:

1. The binary tree is *complete*, i.e. all levels of the tree except possibly the last one are completely filled, and the last level is filled from left to right
2. The value of each node is greater than or equal to the value of its children

Binary heaps are usually implemented using only a variable-size array (e.g. `std::vector`) with all elements of a level l (the root has $l = 0$) being stored at the indexes $[2^l - 1, 2^{l+1} - 1)$. This means that the parent and children of a node can then be found by arithmetic on the array indices. Specifically, given an index i, the indexes of its parent and children can be found as follows:

- Parent: $\lfloor (i-1)/2 \rfloor$ if $i > 0$
- Left Child: $2i + 1$
- Right Child: $2i + 2$

In order to insert an element into a max-heap, one must perform the following operations:

1. Add the element to the bottom level of the heap
2. Compare the added element to its parent; stop if they are in the correct order
3. If not, swap the added element and its parent and return to the previous step

(a) Use the program scaffold contained in the subdirectory for this exercise and implement the `insert` function. Define useful helper functions that can be reused in the following subtasks. Make sure that you use appropriate parameter types (cv-qualification, reference, pointer, etc.). The program scaffold contains test cases that you can use to verify your implementation.

In order to extract an element from a max-heap, one must perform the following operations:

1. Replace the root of the heap with the last element on the last level
2. Compare the new root with its children; stop if they are in the correct order
3. If not, swap the new root with the largest of its children and return to the previous step

(b) Implement the `extract` function. The program scaffold contains test cases that you can use to verify your implementation.

When implementing tree or graph-like data structures, it is often helpful to visually inspect the data structure. For this, the DOT graph description language can be very useful (`http://www.graphviz.org/documentation/`), as it allows for a very easy specification of simple graphs, for example:

```
digraph {
    0 [label="A"];
    1 [label="B"];
    2 [label="C"];
    0 -> 1;
    0 -> 2;
}
```

You can view DOT files with the `xdot` utility which is available on all major Linux distributions.

(c) Implement the `printDot` function which writes a DOT file representing the heap to the specified output stream. That is, your implementation should not write to `std::cout`, but to the the parameter `out` which supports the same operations as `std::cout`.

Note that a heap may contain duplicate values, i.e. the nodes in the DOT file should be identified by their index in the `std::vector`, and have a suitable label like in the example above. Your implementation should first print all nodes in the order in which they are stored in the `std::vector`, followed by the edges in the order they are visited by a preorder depth-first search. You should also inspect the respective test case to determine the exact format that is expected from your output.

Furthermore, complete the `main` function. The program should read unsigned integers from `std::cin` and insert them into a heap until the end of file or an invalid input. Subsequently, the program should write a DOT file representing the heap to the `out` stream which is already defined in the scaffold. Finally, the program should use the `extract` function to print the contents of the heap in sorted order to `std::cout`.

Exercise 2

(40 points)

In this exercise you will implement indirect quicksort and mergesort. Throughout this exercise we assume that there is an immutable array `a` of unsigned integers that we are not allowed to change. In order to sort this array, a second array `b` containing pointers to the individual elements in `a` is created. We then sort the pointers in `b` based on the values of the pointed-to elements in `a`. This allows us to iterate over the pointers in `b` and retrieve the elements in `a` in sorted order by dereferencing these pointers.

(a) Use the program scaffold contained in the subdirectory for this exercise and implement the `quicksort` function which sorts an array of pointers as outlined above. Your implementation should use quicksort to achieve this. Note that the parameters to this function are two pointers, one to the first element in `b` and one to the first element past the end of `b`.

Define suitable helper functions and use appropriate assertions so that your implementation rejects invalid input. The program scaffold contains test cases that you can use to verify your implementation.

(b) Implement the `mergesort` function which sorts an array of pointers as outlined above. Your implementation should use mergesort to achieve this. Note that the parameters to this function are two pointers, one to the first element in `b` and one to the first element past the end of `b`.

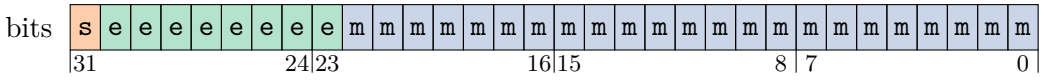
Your implementation is allowed to use additional storage proportional to the size of the input (i.e. you do *not* have to implement in-place mergesort). Nevertheless, you should make sure not to copy data around unnecessarily. Define suitable helper functions and use appropriate assertions so that your implementation rejects invalid input. The program scaffold contains test cases that you can use to verify your implementation.

Exercise 3

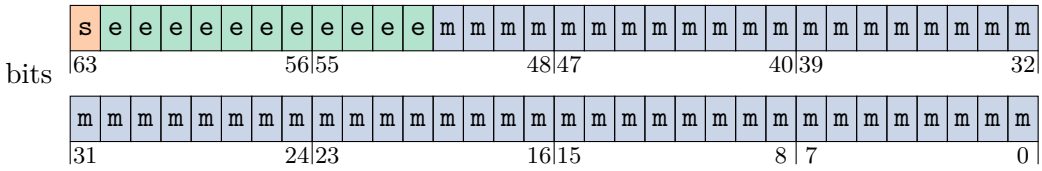
(20 points)

While it is rather easy to access and manipulate the binary representation of integral types in C++ by using bit-wise operators, matters become a little bit more tricky for floating point types. Special care has to be taken to avoid undefined behavior in this case.

In most architectures floating point numbers are represented as specified by IEEE 754. This standard specifies that a `float` is stored as follows:



The most significant bit (bit 31) stores the *sign*, bits 30 to 23 the *exponent* (all bits marked with `e`), and bits 22 to 0 the *mantissa* (all bits marked with `m`). For `double`s the layout of bits is similar:



Here, the most significant bit (bit 63) is also the sign, bits 62 to 52 are the exponent, and bits 51 to 0 are the mantissa.

For both `float` and `double` the meaning of the sign, exponent, and mantissa are the same. When the sign bit is zero, the represented number is positive otherwise it is negative. The exponent and mantissa are unsigned integers that are used to calculate the actual floating point value.

(a) Use the program scaffold contained in the subdirectory for this exercise, and implement the `printBinary` functions. These functions take a single `float`/`double` number as parameter, and should print the sign, exponent, and mantissa to `std::cout` in binary format. Note that you should not use any floating point operations (comparisons, arithmetic, etc.) but instead inspect the *object representation* of the parameter. Furthermore, you should not use `std::bitset` or `std::vector<bool>`.

The output for the `float` number 42.125 for which the sign is 0, the exponent is 132, and the mantissa is 2654208, should look like this:

```
sign: 0
exponent: 10000100
mantissa: 01010001000000000000000
```

The program scaffold contains test cases that you can use to verify your implementation. Make sure that your implementation does not contain undefined behavior.

Sometimes, it is required to store heterogeneous or opaque data in a contiguous block of memory. This can be achieved by storing the object representation of the data, and possibly some type information to distinguish between the different data types. In the following, you will implement a simple stack that can store both `float` and `double` values on top of an `std::vector<unsigned char>`.

For pushing a value onto the stack, your implementation should first push the value itself onto the stack, followed by a single byte containing the number of bytes of its object representation, which we will call n.

To pop a value off the stack, your implementation should first check that the type of the next value on the stack matches the expected output type by inspecting n. If this check succeeds, your implementation should read the value and pop the corresponding bytes off the stack. If the check fails, no changes should be made to the stack.

(b) Implement the `push` functions. These functions take a reference to the stack and a `float`/`double` value as parameters. The stack is represented by an `std::vector` of individual bytes, with the top of the stack being at the end of the `std::vector`. They should push the value onto the stack as outlined above.

Also implement the `pop` functions, which take a reference to the stack and a reference to a `float`/`double` value as parameters. They first need to check if the type of the next value on the stack matches the expected output type. If not, they should return `false`. Otherwise, they should pop the value off the stack, store it in the value parameter, and return `true`. If the stack is empty, these functions should also return `false`.

The program scaffold contains test cases that you can use to verify your implementation. Make sure that your implementation does not contain undefined behavior.