Course 'Imperative Programming' (IPC031)

Assignment 10: *Order of Complexity Analysis and Heap Sort*

1. Background

In this assignment you analyze the order of run-time complexity of a number of functions. You study and implement an advanced iterative sorting algorithm, *Heap Sort*, and determine its order of run-time complexity. You implement the *Heap Sort* algorithm for *vector-of-integers*.

2. Learning objectives

After doing this assignment you are able to:

- analyze algorithms for their order of run-time complexity;
- apply the *Heap Sort* algorithm for stepwise ordering an array / vector (exam stuff);
- implement the *Heap Sort* algorithm and explain its higher efficiency in comparison to the sorting algorithms insertion sort, selection sort, and bubble sort.

3. Assignment

Part 1: Analyze the order of run-time complexity

Consider the algorithms below. Identify their input and derive the *order* of run-time complexity in terms of the input size. Give a short(!) explanation for your answers.

(a)

```
void easter ( int year, int& day, int& month )
{
    const int a = year % 19 ;
    const int b = year / 100 ;
    const int c = year % 100 ;
    const int d = b / 4 ;
    const int e = b % 4 ;
    const int f = (b + 8) / 25 ;
    const int g = (b - f + 1) / 3 ;
    const int h = (19 * a + b - d - g + 15) % 30 ;
    const int i = c / 4 ;
    const int k = c % 4 ;
    const int L = (32 + 2 * e + 2 * i - h - k) % 7 ;
    const int m = (a + 11 * h + 22 * L) / 451 ;
    month       =  (h + L - 7 * m + 114) / 31 ;
    day         = ((h + L - 7 * m + 114) % 31) + 1 ;
}
```

(b)

```
bool is_prime (int x, int& divisor)
{
    if (x <= 1)
        return false ;
    for (divisor = 2 ; divisor <= sqrt (static_cast<double>(x)); divisor++)
    {
        if (x % divisor == 0)
            return false ;
    }
    return true ;
}
```

Part 2: The Heap Sort algorithm

Heap Sort (1964) is algorithm for in situ sorting of arrays / vectors. It is an elegant but non-trivial algorithm that exploits a tree representation of a partially ordered array, called a *heap*. Below we begin with an explanation of *Heap Sort*. This is followed by a couple of exercises.
Every one-dimensional array can be represented by a *tree*. Here is an example:

```
                                              2
                                             / \
                                           14   30
{2,14,30,29,42,30,11,-5,0,51}       ≅      / \   / \
                                         29  42 30  11
                                         /\  /
                                       -5  0 51
```

i) An array of 10 elements   **ii)** The used *tree* representation of the array in **i)**

The relation between the elements in the tree structure depends on their index (or position) in the array. With each element at index *i* (*i* ≥ 0), we associate two *child* elements at index 2*i*+1 and 2*i*+2.

Conversely, with each element at index *i*, you can find its *parent* element at index (*i*-1)/2. Be aware of the necessity to always check the proper bounds: as always, for an array with *N* elements, the valid index values range from *0* up to and including *N-1*.
An array is a *heap* if for every element *x* in the array, at index *i*, any array element *y* that is reachable from *x* via the child-relation (possibly used repeatedly) does not have a value larger than *x*. A consequence of this property is that the first element of a heap is *maximal (up to equality)* with respect to all other values in the heap. By traversing the array from an element, according to the child-relation, you may encounter values in a non-increasing order.
The *Heap Sort* algorithm of an (unsorted) array *A* of *N* elements consists of two subsequent phases:

Phase 1: Building a heap

Inspect each element of *A*, starting from the first element at index *0* and ending with the last element at index *N-1*. Upon inspecting an element at index *i* you may need to move the corresponding element *A[i]* to a new position in *A[0]...A[i]*, in order to maintain the heap-property. This is done by 'pushing upward' the element at index *i*, by repeatedly swapping it with its parent, as long as its value is larger than the value of its parent. During this operation, the rest of the array is unchanged. We call this algorithm `push_up`. You need to repeatedly apply `push_up` operations until all elements of *A* are inspected. The arising algorithm, that we call `build_heap`, may turn array *A* into a heap.

```
       51
      /  \
    42    30
    / \   / \
   2  30 14  11
  /\  /
-5  0 29
```

iii) A tree representation of **ii)** after `build_heap`

Phase 2: Sorting the array

In a heap *A[0]...A[i]*, for *i* the index of the last element of the heap, the maximum value is at *A[0]*. Swap *A[0]* with *A[i]*. We call this function `swap`. Now the maximum value is in its proper position, and is kept there. Due to the swap, the array *A[0]...A[i-1]* may no longer be a heap because the new value at *A[0]* may not be maximal with respect to the values at *A[0]* ... *A[i-1]*. If so, *A[0]* needs to be 'pushed downwards' by repeatedly swapping it with its largest child as long as its value is smaller than one of its child values. Once no more swapping is possible, *A[0]...A[i-1]* satisfies the condition to be a heap. We call this function `push_down`. Repeated application of `swap` and `push_down` decreases the heap size and increases the number of already sorted elements. Proceed until all elements of *A* are sorted. We call this function `pick_heap`.

```
       -5
      /  \
     0     2
    / \   / \
  11  14 29  30
  /\  /
30 42 51
```

```
≅ {-5,0,2,11,14,29,30,30,42,51}
```

iv) A tree representation of **iii)** after `pick_heap`

v) The resulting array which is sorted

Exercises:

a) Apply the heap sort algorithm to the following array of integers:

`{15, 40, 42, -15, 30, 35, 5}`

Show the steps of phase 1 graphically, by means of tree structures as illustrated in Figure **ii)**. Show the steps of phase 2 by means of arrays as depicted in Figure **i)**. Include your answer as comments in your C++ code.

b) What is the order of run-time complexity of the algorithms `push_up`, `build_heap`, `push_down`, `pick_heap`, and so, of *Heap Sort*, for an array of *N* elements? Motivate your answer.

Part 3: The Heap Sort implementation
Implement the *Heap Sort* algorithm for *vector-of-integers*.

4. Products

As product-to-deliver you only need to upload to Brightspace *"main.cpp"* that you have created with solutions for each part of the assignment.

⇒ **Deadline:** *Thursday, November 22, 2018, 13:30h*.