Module IN2002—Data Structures and Algorithms
Sample Answers to Exercise Sheet 2

1. The function power presented in the lecture is rather inefficient. Use the equations:

$$x^{2n} = x^n \cdot x^n$$
$$x^{2n+1} = x \cdot x^n \cdot x^n$$

to write a more efficient **recursive** algorithm. How much time does the new version take?

```
/**
* Calculate x^n more efficiently.
* @param x The base.
* @param n The exponent (must not be negative).
*/
double power(double x, int n) {
     if (n == 0)
          return 1.0;
     double p = power(x, n/2); // p=x^(n/2) int division!
     p *= p; // p = x^(n-n%2)
     if (n%2 != 0) // odd n needs one more factor
          p *= x; // p = x^n
     return p;
}
```

- The call tree for this function looks like this:

```
power(x, n)
     power(x, n/2)
          power(x, n/4)
               ...
```

and this will have depth log n. As the function without the recursive call takes constant time, the whole function takes O(log n) time. To achieve this, it is crucial that we call `power(x, n/2)` only once and store its value in `p` to use the result twice. In other words: using `power(n, n/2) * power(n, n/2)` makes it O(*n*) again.

2. Consider the function:

```
int sumOfSquares(int n) {
     if (n == 0)
          return 0;
     return n*n + sumOfSquares(n-1);
}
```

Is it tail recursive? Can you convert it to iterative? If so, how? If not, why not?

- This function computes the sum $n^2 + ... + 2^2 + 1^2$ and takes time O(n).

- The last call the function makes is n*n + sumOfSquares(n-1). The function is not tail recursive because the last call it makes is to operator '+' rather than the recursive call "sumOfSquares".

- The same function can be expressed as tail recursive by creating an auxiliary function:

```
int sumOfSquares(int n) {
     return sos(n, 0);
}
int sos(int n, int sum) {
     if (n==0)
          return sum;
     return sos(n-1, n*n + sum);
}
```

- This version can be easily converted to iterative:

```
int sumOfSquares(int n) {
     int sum = 0;
     for ( ; n>0; n-=1)
          sum = n*n + sum;
     return sum;
}
```

- A skilled programmer can of course convert the original version to iterative, too, using a stack. But for a tail-recursive function it works easily without a stack.

3. The array priority queue implementation provided in the lecture notes assumed a sorted array, where the method *add* contains a call to a method *insert*. Provide the algorithm for the method *insert* (pseudocode or a programming language are fine).

```
void insert(array data, int size, int element) {
     int position ← 1;
     // find the position in the array where the new element
     // needs to be added
     WHILE (position < size AND data[position] < element)
          position++;
     // move all larger existing elements up one position
     int inPlace ← size
     WHILE (inPlace > position)
          data[inPlace] ← data[inPlace-1];
          inPlace--;
     // insert the new element in its correct place
     data[position] ← element;
}
```

- Note that finding the position here has been done using sequential search. It could also have been done using binary search. That would have improved the time complexity of the first loop, but not of the second. Hence, the overall complexity would remain unchanged.

4. Had the priority queue been implemented using an unsorted array, there would be some differences to the implementation of its basic operations (*isEmpty*, *add*, *extractMax*). Discuss which of these would need to be changed and why, while providing new algorithms for those requiring changes (again, pseudocode or a programming language are fine).

- isEmpty remains the same, as what really matters is simply whether there are any elements in the array or not.
- add would be faster and changed, as we would simply add the new element to the end of the existing array.

```
void add(int elt) {
     count++;
     data[count] ← elt;
}
```

- extractMax would be slower and changed, as we would need to traverse the whole array keeping track of the largest value found so far and its position. Then remove that element and reshuffle all other elements that were in the array in positions after it, actually removing the largest element.

```
int extractMax() {
     int largest ← data[1]
     int position ← 1
     int i ← 1
     // find largest value and its position
     WHILE i < count
          i++
          IF largest < data[i]
               largest ← data[i]
               position ← i
     // reshuffle other elements
     i ← position
     WHILE i < count
          data[i] ← data[i+1]
          i++
     count--
     Return largest
}
```

And a bit of programming. You are not expected to tackle this during the tutorial slot, but at some later time, at your own convenience. Note that answers to this will be released much later than for the other questions, giving you time to experiment with it.

5. A cocktail shaker sort designed by Donald Knuth is a modification of bubble sort in which the direction of bubbling changes in each iteration: In one iteration, the smallest element is bubbled up; in the next, the largest is bubbled down; in the next, the second smallest is bubbled up; and so forth. Implement the new algorithm and discuss its complexity.

And a bit of programming. Answers to this will be released much later, giving you time to experiment with it.

Refresher Powers and Logs (justify your answers)
// TO BE USED FOR REVISION OUTSIDE THE TUTORIAL

6. What is 2^3 (2 power 3)?
   a) 9
      - ➢ No, $9 = 3*3 = 3^2$
   b) 6
      - ➢ No, 6=2*3
   c) 8
      - ➢ Yes, $8 = 2*2*2 = 2^3$
   d) 5
      - ➢ No, 5=2+3

7. What is the $\log_2 16$ (logarithm of 16 base 2)?
   a) 1
      - ➢ No, $2^1 = 2$, $1 = \log_2 2$ (or log 2 base 2)
   b) 2
      - ➢ No, $2^2 = 2*2 = 4$, $2 = \log_2 4$ (or log 4 base 2)
   c) 3
      - ➢ No, $2^3 = 2*2*2 = 8$, $3 = \log_2 8$ (or log 8 base 2)
   d) 4
      - ➢ Yes, $2^4 = 2*2*2*2 = 16$, $4 = \log_2 16$ (or log 16 base 2)

8. What is the $\log_4 16$ (logarithm of 16 base 4)?
   a) 1
      - ➢ No, $4^1=4$, $1 = \log_4 4$ (or log 4 base 4)
   b) 2
      - ➢ Yes, $4^2 = 4*4 = 16$, $2 = \log_4 16$ (or log 16 base 4)
   c) 3
      - ➢ No, $4^3 = 4*4*4 = 64$, $3 = \log_4 64$ (or log 64 base 4)
   d) 4
      - ➢ No, $4^4 = 4*4*4*4 = 256$, $4 = \log_4 256$ (or log 256 base 4)