1. Consider sorting n numbers in array A by first finding the smallest element of A and putting it in the first entry of another array B. Then find the second smallest element of A and put it in the second entry of B. Continue in this manner for the n elements of A. Write pseudocode for this algorithm, which is known as **selection sort**. Give the best-case and worst-case running times of selection sort in Θ-notation.

2. Consider linear search again (see Exercise1.1-3). How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about the worst case? What are the average-case and worst-case running times of linear search in Θ-notation. Justify your answer.

Suppose that cost to lookup an item in the list is a constant c. On average, supposing that the key we're looking for is in the list and assuming a uniform distribution, we would expect to inspect half the keys in the list before finding the one we're looking for. The expected lookup time is therefore $T(n) = cn/2$. Since $c/2$ is a constant, this can be expressed in Θ notation as $\Theta(n)$.

In the worst case, the key we are looking for is not in the list, or it is the last item in the list. In either case, we need to inspect all n items before terminating. The time spent searching the list is therefore $T(n) = cn$. Since c is a constant, this can be expressed in Θ notation as $\Theta(n)$.

The expected lookup time for both the average and the worst case is therefore $\Theta(n)$.

3. Consider the problem of determining whether an arbitrary sequence $(x_1, x_2, ..., x_n)$ of n numbers contains repeated occurrences of some number. Show that this can be done in $\Theta(n \lg n)$ time, where $\lg n$ stands for $\log_2 n$.

The number sequence can be sorted in $\Theta(n \lg n)$ time using a recursive algorithm like merge sort or quicksort, and then the resulting list can be searched in $\Theta(n)$ linear time to see whether two adjacent elements have the same value. The total running time would be $T(n) = \Theta(n \lg n)$.

4. Consider the problem of evaluating a polynomial at a point. Given n coefficients $a_0, a_1, ..., a_n$ and a real number x, we wish to compute $\sum_{ii=o}^{n-1} a_i x_i$. Describe a straightforward $\Theta(n^2)$-time algorithm for this problem. Describe a $\Theta(n)$-time algorithm that uses the following method (called Horner's rule) for rewriting the polynomial:

$$\sum_{i=0}^{n-1} a_i x^i = (...(a_{n-1}x + a_{n_2})x + ... + a_1)x + a_0$$

Computation of polynomials requires multiplication and addition. Let us assume, for our model of computation, that each multiplication and addition costs a constant amount of time, c_m and c_a respectively.

The computation of x^i is bounded from above by $\Theta(n)$ (i.e., a maximum of n multiplications, since $i < n$). There are a total n such terms in the expression to be added together. Hence we should be able to evaluate the expression in $\Theta(n^2)$ time, in the worst case.

An example $\Theta(n^2)$ evaluation is as follows. Coefficients are sorted in descending order of x^i, that is, the list $(1, 2, 3)$ would represent the coefficients in the expression $P(x) = x^2 + 2x + 3$:

```
1  import java.util.List;
2
3  public class Polynomial {
4      //! Coefficients are sorted in descending order of x^i
5      private List<Integer> coefficients;
6      public Polynomial(List<Integer> coefficients) {
7          this.coefficients = coefficients;
8      }
9
10     //! Simple O(n^2) evaluation of polynomial at point 'x'.
11     public Double evaluate(Double x) {
12         Double result = 0.0;
13         int n = this.coefficients.size();
14         for (int i=0; i<n; ++i) {
15             result += this.coefficients.get(i) * exp(x, n-i-1);
16         }
17         return result;
18     }
19
20     //! Exponential function. O(n) performance.
21     private static Double exp(Double base, Integer power) {
22         Double result = 1.0;
23         for (int i=0; i<power; ++i) {
24             result *= base;
```

```
        }
        return result;
    }
}
```

An example $\Theta(n)$ Horner evaluation is as follows – as before, the coefficients are sorted in descending order of x^i:

```
import java.util.List;

public class Polynomial {
    //! Coefficients are sorted in descending order of x^i.
    private List<Integer> coefficients;
    public Polynomial(List<Integer> coefficients) {
        this.coefficients = coefficients;
    }

    //! Horner O(n) evaluation of polynomial at point 'x'.
    public Double horner(Double x) {
        Double result = 0.0;
        if (this.coefficients.size() != 0) {
            result = new Double(this.coefficients.get(0));
            int n = this.coefficients.size();
            for (int i=1; i<n; ++i) {
                result = (result*x + this.coefficients.get(i));
            }
        }
        return result;
    }
}
```

Note that the Horner evaluation does not require a separate evaluation of the exponential function.

5. Express the function $n^3/1000 - 100n^2 - 100n + 3$ in terms of Θ-notation.

For a polynomial function f in n, only the highest order term is relevant when considering order of growth statistics. More precisely, suppose that

$$f(n) = \sum_{i=0}^{k} a_i n^i = a_k n^k + a_{k-1} n^{k-1} + ... + a_0$$

$$\frac{1}{n^k} f(n) = a_k + \frac{1}{n} a_{k-1} + ... + \frac{1}{n^k} a_0$$

so that

$$\lim_{n\to\infty} \frac{1}{n^k} f(n) = a_k$$

Hence for large n we can write

$$f(n) \approx a_k n^k$$

The constant coefficient a_k can be ignored, giving $\Theta(f(n)) = n^k$.

In the example above, where $f(n) = n^3/1000 - 100n^2 - 100n + 3$, even though the quadratic and linear terms have large negative coefficients and even though the coefficient on the n^3 term is a small fraction, we can still write $\Theta(f(n)) = n^3$.

6. How can we modify almost any algorithm to have a good best-case running time?

For any algorithm, we can "hard-code" the right answer for known inputs.

For example, a good recursive sorting algorithm normally runs in $\Theta(n \lg n)$ time. Insertion sort, studied in Section 1, runs in $\Theta(n^2)$ time. However, we can take a known input array, say $< 31, 41, 59, 26, 41, 58 >$, and "hard-code" the correct response – in this case $< 26, 31, 41, 41, 58, 59 >$.

This still requires a $\Theta(n)$ operation to compare the input array with the hard-coded answer, to see if they are equal. If they are, we can return the hard-coded answer. If not, we can sort the array using a standard sorting algorithm. But $\Theta(n)$ is still a big improvement over $\Theta(n^2)$ or $\Theta(n \lg n)$.

Similarly, the exponential function normally requires n multiplications, where n is the power the base is being raising to. However, we could hard-code the response to return the correct answer in the case of specific inputs. Suppose we hard-code $\exp(base, power)$ to return 81 when invoked with arguments $base = 3$ and $power = 4$. This would make $\exp(3, 4)$ a $\Theta(1)$ operation. Invoking the procedure with arbitrary inputs would still be a $\Theta(n)$ operation, where n is the size of the *power* argument.