Assignment 1: Accompanying pseudocode for 3SUM and 4SUM

1 The 3SUM problem

Let us recall the 3SUM problem. The problem statement[1] is as follows: Given a list of n numbers, are there any three numbers a, b, c in the list such that $a + b + c = 0$?

There are three basic solutions. The first one is to try all triplets (a, b, c) in the list, for example, by three nested for loops, and compute their sum. This solution is obviously $\Theta(n^3)$ in the number of elements. This solution is given in pseudocode in Algorithm 1.

Algorithm 1 The cubic algorithm for 3SUM.

```
1: Input: a list of n integers x
2: Output: a triple a, b, c of integers in the list x such that a + b + c = 0 if
   such a triple exists, or none otherwise
3: function ThreeSumCubic(x)
4:     for i ← 1, 2, ..., n do
5:         a ← x[i]
6:         for j ← i + 1, i + 2, ..., n do
7:             b ← x[j]
8:             for k ← j + 1, j + 2, ..., n do
9:                 c ← x[k]
10:                if a + b + c = 0 then
11:                    return (a, b, c)
12:                end if
13:            end for
14:        end for
15:    end for
16:    return none
17: end function
```

A less obvious solution works by reducing the problem to the related 2SUM problem: iterate over all elements a of the list, and check if there exists a pair of elements b, c in the remainder that sum up to $b + c = -a$.

[1]Note that there are several variants of the problem. Here we allow the solution to contain the same number multiple times, provided the list originally contained the number multiple times.

The trick is that if we first sort the list, once the target value is known, we can perform a linear scan of the remainder: start by setting b next value from a, and c as the last value; since $a \leq b \leq c$, if $a + b + c < 0$, we know that we have to increase b to get closer to 0, and if $a + b + c > 0$, we need to decrease c, and if $c \leq b$, we know that there are no two values that sum up to $-a$. The complexity of this solution is: $\Theta(n \log n + n^2) = \Theta(n^2)$: when n is large, the sorting does not factor into play, as the time is dominated by the quadratic search. This solution is given in pseudocode in Algorithm 2.

Algorithm 2 The quadratic algorithm for 3SUM.

```
 1: Input: a list of n integers x
 2: Output: a triple a, b, c of integers in the list x such that a + b + c = 0 if
    such a triple exists, or none otherwise
 3: function ThreeSumQuadratic(x)
 4:     Sort(x)
 5:     for i ← 1, 2, ..., n do
 6:         a ← x[i]
 7:         ℓ ← i + 1
 8:         r ← n
 9:         while ℓ < r do
10:             b ← x[ℓ]
11:             c ← x[r]
12:             if a + b + c = 0 then
13:                 return (a, b, c)
14:             else if a + b + c < 0 then
15:                 ℓ ← ℓ + 1
16:             else
17:                 r ← r − 1
18:             end if
19:         end while
20:     end for
21:     return none
22: end function
```

Another less obvious basic solution is to store all elements of the list in a hash map along with their indices in the original list. Then, iterate over each pair of elements (a, b) and query the hash map to see if $-a - b$ is in the list. Assuming the elements of the hash table can be accessed in $O(1)$ time, this solution is also $\Theta(n^2)$. This solution is given in pseudocode 3.

Observe that there is an extra check on line 13 of Algorithm 3 that the index j is strictly less than the index k returned from the hash map. Why

is that? Hint: consider what happens if you have a list of two elements a, b satisfying $a + a + b = 0$. Why do we not need to check that $i < j$?

Algorithm 3 The hash map algorithm for 3SUM.

```
1: Input: a list of n integers x
2: Output: a triple a, b, c of integers in the list x such that a + b + c = 0 if
   such a triple exists, or none otherwise
3: function ThreeSumHashMap(x)
4:     Initialize hash map H
5:     for i ← 1, 2, ..., n do
6:         H[x[i]] ← i
7:     end for
8:     for i ← 1, 2, ..., n do
9:         a ← x[i]
10:        for j ← i + 1, i + 2, ..., n do
11:            b ← x[j]
12:            k ← H[−a − b]          ▷ Assume the hash map returns 0 if the
   element is not present
13:            if k ≠ 0 and j < k then
14:                return (a, b, x[k])
15:            end if
16:        end for
17:    end for
18:    return none
19: end function
```

2 Algorithms for 4SUM

The 4SUM problem is defined as follows: given a list of n integers, do there exist four integers in the list a, b, c, and d, such that $a + b + c + d = 0$?

Algorithms 4, 5, and 6 provide pseudocode for three different algorithms that mimic those of 3SUM, but adapted for the 4SUM problem. Note that when implementing the code, particularly that of Algorithm 6, some care must be taken, as the hash map maps integers to pairs of integers; you will probably need to store arrays in the Java `HashMap`.

Algorithm 4 The quartic algorithm for 4SUM.

```
Input: a list of n integers x
Output: a quadruple a, b, c, d of integers in the list x such that a + b + c + d = 0 if such a triple exists, or none otherwise
function FourSumQuartic(x)
    for i ← 1, 2, ..., n do
        a ← x[i]
        for j ← i + 1, i + 2, ..., n do
            b ← x[j]
            for k ← j + 1, j + 2, ..., n do
                c ← x[k]
                for ℓ ← k + 1, k + 2, ..., n do
                    d ← x[ℓ]
                    if a + b + c + d = 0 then
                        return (a, b, c, d)
                    end if
                end for
            end for
        end for
    end for
    return none
end function
```

Algorithm 5 The cubic algorithm for 4SUM.

```
1: Input: a list of n integers x
2: Output: a quadruple a, b, c, d of integers in the list x such that a + b +
   c + d = 0 if such a triple exists, or none otherwise
3: function FourSumQuartic(x)
4:     Sort(x)
5:     for i ← 1, 2, ..., n do
6:         a ← x[i]
7:         for j ← i + 1, i + 2, ..., n do
8:             b ← x[j]
9:             ℓ ← j + 1
10:            r ← n
11:            while ℓ < r do
12:                c ← x[ℓ]
13:                d ← x[r]
14:                if a + b + c + d = 0 then
15:                    return (a, b, c, d)
16:                else if a + b + c + d < 0 then
17:                    ℓ ← ℓ + 1
18:                else
19:                    r ← r − 1
20:                end if
21:            end while
22:        end for
23:    end for
24:    return none
25: end function
```

Algorithm 6 The hash map algorithm for 4SUM.

Input: a list of n integers x
Output: a quadruple a, b, c, d of integers in the list x such that $a + b + c + d = 0$ if such a triple exists, or *none* otherwise
function FourSumHashMap(x)
    Initialize hash map H
    for $i \leftarrow 1, 2, \ldots, n$ **do**
        for $j \leftarrow i + 1, i + 2, \ldots, n$ **do**
            $H[x[i] + x[j]] \leftarrow (i, j)$
        end for
    end for
    for $i \leftarrow 1, 2, \ldots, n$ **do**
        $a \leftarrow x[i]$
        for $j \leftarrow i + 1, i + 2, \ldots, n$ **do**
            $b \leftarrow x[j]$
            $(k, \ell) \leftarrow H[-a - b]$ ▷ Assume the hash map returns a pair of integers, and $(0, 0)$ if the element is not present
            if $(k, \ell) \neq (0, 0)$ **and** $j < k$ **then**
                $c \leftarrow x[k]$
                $d \leftarrow x[\ell]$
                return (a, b, c, d)
            end if
        end for
    end for
    return *none*
end function