ALGORITMIEK: answers exercise class 7

Problem 1.
See slides 2–4 of lecture 8.

Problem 2.
See slides 4–6 of lecture 8.

Problem 5.
a. Als we twee negatieve (< 0) getallen bij elkaar optellen is het antwoord zeker < 0. Als we twee positieve (> 0) getallen bij elkaar optellen is het antwoord > 0. Beide gevallen leveren dus nooit $= 0$ op. Ergo: van alle paren (i, j) met i en j beide oneven of i en j beide even weten we zeker dat $A[i] + A[j] \neq 0$.
b. Brute force: alle paren (i, j) met $i < j$ aflopen en testen of $A[i] + A[j] = 0$. Met inachtneming van de restrictie dat i en j niet beide even of beide oneven zijn.

```
int teller = 0;
for (i = 0; i < n; i++)
  for (j = i+1; j < n; j+=2)
  // nu worden bij elke even index i alleen oneven indices
  // j > i afgelopen, en analoog voor oneven i
    if (A[i] + A[j] == 0)
      teller++;
return teller;
```

c. We splitsen het array nu (herhaald, want recursie) in twee stukken. We tellen het aantal gevraagde paren in de linkerhelft (recursieve aanroep) en in de rechterhelft (recursieve aanroep). Vervolgens moeten we alleen nog paren indices (i, j) controleren waarbij i in de linkerhelft zit en j in de rechterhelft. Wederom onder de restrictie dat we alleen paren (i, j) bekijken met i even en j oneven, of met i oneven en j even.
Merk op dat het basisgeval wordt gegeven door `rechts` = `links` + 1 als we het stuk A[`links`], ..., A[`rechts`] bekijken. Eerste aanroep: aantal = aantal2(A,0,n-1);

```
int aantal2(int A[], int links, int rechts) {
  int teller = 0;
  int m, i, j;
  if (rechts == links+1) { // 2 elementen
    if (A[links]+A[rechts] == 0)
      return 1;
    else
      return 0;
  } // basisgeval twee elementen
  else {
  // meer dan twee elementen: recursieve aanroepen
    m = (links+rechts)/2; // m is altijd oneven
    // aantal paren links en aantal paren rechts optellen
    teller = aantal2(A, links, m) + aantal2(A, m+1, rechts);
    // en nu linkerhelft met rechterhelft vergelijken
    for (i = links; i < m; i++) { // even i links
      for (j = m+2; j <= rechts; j+=2)  // oneven j rechts
```

```
        if (A[i] + A[j] == 0)
          teller++;
    }
    for (i = links+1; i <= m; i++) { // oneven i links
      for (j = m+1; j < rechts; j+=2)  // even j rechts
        if (A[i] + A[j] == 0)
          teller++;
    }
    return teller;
  } // else meer dan 2 elementen
} // aantal2
```

Problem 6.
For $1201 * 2430$:

$$\begin{aligned} 1201 * 2430 &= (12 * 10^2 + 01) * (24 * 10^2 + 30) \\ &= c_2 * 10^4 + c_1 * 10^2 + c_0 \end{aligned}$$

where

$$\begin{aligned} c_2 &= 12 * 24 \\ c_0 &= 01 * 30 \\ c_1 &= (12 + 01) * (24 + 30) - (c_2 + c_0) = 13 * 54 - (c_2 + c_0) \end{aligned}$$

For $12 * 24$:

$$\begin{aligned} 12 * 24 &= (1 * 10^1 + 2) * (2 * 10^1 + 4) \\ &= c_2 * 10^2 + c_1 * 10^1 + c_0 \end{aligned}$$

where

$$\begin{aligned} c_2 &= 1 * 2 = 2 \\ c_0 &= 2 * 4 = 8 \\ c_1 &= (1 + 2) * (2 + 4) - (c_2 + c_0) = 3 * 6 - (2 + 8) = 18 - 10 = 8 \\ \text{So, } 12 * 24 &= 2 * 10^2 + 8 * 10^1 + 8 = 288 \end{aligned}$$

For $01 * 30$:

$$\begin{aligned} 01 * 30 &= (0 * 10^1 + 1) * (3 * 10^1 + 0) \\ &= c_2 * 10^2 + c_1 * 10^1 + c_0 \end{aligned}$$

where

$$\begin{aligned} c_2 &= 0 * 3 = 0 \\ c_0 &= 1 * 0 = 0 \\ c_1 &= (0 + 1) * (3 + 0) - (c_2 + c_0) = 1 * 3 - (0 + 0) = 3 - 0 = 3 \\ \text{So, } 01 * 30 &= 0 * 10^2 + 3 * 10^1 + 0 = 30 \end{aligned}$$

For $13 * 54$,

$$\begin{aligned} 13 * 54 &= (1 * 10^1 + 3) * (5 * 10^1 + 4) \\ &= c_2 * 10^2 + c_1 * 10^1 + c_0 \end{aligned}$$

where

$$\begin{aligned} c_2 &= 1 * 5 = 5 \\ c_0 &= 3 * 4 = 12 \\ c_1 &= (1+3) * (5+4) - (c_2 + c_0) = 4 * 9 - (5 + 12) = 36 - 17 = 19 \\ \text{So, } 13 * 54 &= 5 * 10^2 + 19 * 10^1 + 12 = 702 \end{aligned}$$

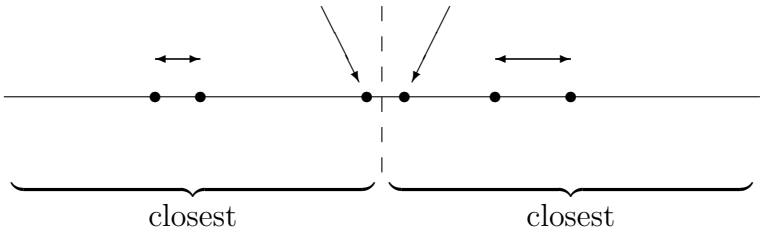
Hence,

$$1201 * 2430 = c_2 * 10^4 + c_1 * 10^2 + c_0$$

where

$$\begin{aligned} c_2 &= 12 * 24 = 288 \\ c_0 &= 01 * 30 = 30 \\ c_1 &= 13 * 54 - (c_2 + c_0) = 702 - (288 + 30) = 702 - 318 = 384 \\ \text{So, } 1201 * 2430 &= 288 * 10^4 + 384 * 10^2 + 30 = 2918430 \end{aligned}$$

Problem 7.
Both in part **a.** and part **b.**, we assume that the numbers have been sorted already (e.g., by mergesort).
a. Assuming that the points are sorted in increasing order, we can find the closest pair (or, for simplicity, just the distance between two closest points) by comparing three distances: the distance beetween the two closest points in the first half of the sorted list, the distance beetween the two closest points in its second half, and the distance between the rightmost point the first half and the leftmost point in the second half.



Therefore, for a given array $P[0 \ldots n-1]$ of real numbers, sorted in nondecreasing order, we can call `ClosestNumbers (P, 0, n-1)`, where

```
double ClosestNumbers (double P[], int left, int right)
// A divide-and-conquer alg. for the one-dimensional closest-pair problem
// Input: A subarray P[left...right] (left <= right) of a given array
//        P[0...n-1] of real numbers sorted in nondecreasing order
// Output: The distance between the closest pair of numbers
```

```
{ if (left==right)  // only one number, hence no distance
    return ‘infinity’;
  else
  { mid = (left+right)/2;
    return  min { ClosestNumbers (P, left, mid),
                  ClosestNumbers (P, mid+1, right),
                  P[mid+1] - P[mid] };
  }
}
```

b. A non-recursive algorithm may simply iterate over all pairs of numbers in the sorted list:

```
if (n<=1)  // at most one number, hence no distance
  return ‘infinity’;
else
{ tempmin = P[1]-P[0];
  for (i=1; i<n-1; i++)
    if (P[i+1]-P[i] < tempmin)
      tempmin = P[i+1]-P[i];

  return tempmin;
}
```

Problem 10.
We use a global array `char B[0...n+1]`, with `B[n] = '\0'` (the end-of-string marker). We then call `BitstringsRec (n)`, where

```
void BitstringsRec (int i)
// Generates recursively all the bit strings of a given length
// Input: A nonnegative integer i
// Output: All bit strings of length i as contents of a global char array B
// Pre: B already contains end-of-string marker
{ if (i==0)
    cout << B << endl;
  else
  { B[i-1] = '0';
    BitstringsRec (i-1);
    B[i-1] = '1';
    BitstringsRec (i-1);
  }
}
```

Problem 11.
b.

0 :	0000
1 is binary 0001:	0001
2 is binary 0010:	0011
3 is binary 0011:	0010
4 is binary 0100:	0110
5 is binary 0101:	0111
6 is binary 0110:	0101
7 is binary 0111:	0100
8 is binary 1000:	1100
9 is binary 1001:	1101
10 is binary 1010:	1111
11 is binary 1011:	1110
12 is binary 1100:	1010
13 is binary 1101:	1011
14 is binary 1110:	1001
15 is binary 1111:	1000

This is exactly the same Gray code as the one you would get at part **a.**. This time, however, non-recursively.

Problem 12.
a. Swap $A[2]$ and $A[n-1]$, swap $A[4]$ and $A[n-3]$, etcetera, until the first index (i in the code below) has reached the middle of the array:

```
for (i=2; i<=n/2; i+=2)
  swap (A[i], A[n-i+1]);
```

This algorithm performs $n/4$ swaps if $n/2$ is even, and $(n-2)/4$ swaps if $n/2$ is odd. In both cases: $\lfloor n/4 \rfloor$ swaps.
b. The idea is as follows: swapping $A[2]$ and $A[n-1]$ reduces the problem for indices $1, \ldots, n$ to the same problem for indices $3, \ldots, n-2$. That is, four array elements less. We call `hussel (1, n)`, where

```
void hussel (int i, int j)
{
  if (j-i+1 >= 4)  // at least 4 elements
  { swap (A[i+1], A[j-1]);
    hussel (i+2, j-2);
  }
  // if 2 or 0 elements: do nothing, it is OK already
}
```

Problem 14.
The idea is as follows: let m be the middle of the (complete) array $A[1 \ldots n]$: $m = (n+1)/2$ (rounded down, if applicable). We first check if $A[m] < A[m+1]$.
If so, then the first half of the array ($A[1 \ldots m]$) must be increasing: $A[1] < A[2] < \cdots < A[m]$. The reason is, that we cannot have a decrease first, followed by an increase between m and $m+1$. This implies that the peak p must be in the second half of the array ($A[m+1 \ldots n]$).
If, on the other hand, $A[m] > A[m+1]$, then we find in an analogous way that the second half of the array must be decreasing: $A[m+1] > A[m+2] > \cdots > A[n]$. This implies that the peak p must be in the first half of the array.
Remark 1: In general, we do not consider the complete array, but a subarray `A[left...right]`
Remark 2: To determine the middle m and to compare $A[m]$ with $A[m+1]$, we need at least two elements. We therefore consider the case that we have one element separately.
We thus have a function `int search (int A[], int left, int right)`, with first call `search (A, 1, n)`, as follows:

```
int search (int A[], int left, int right)
// Pre: left <= right
{ int m;

  if (left==right)  // one element
    return left;
  else
  { m = (left+right)/2;
    if (A[m]<A[m+1)
      return search (A, m+1, right);
    else
      return search (A, left, m);
  }
}
```