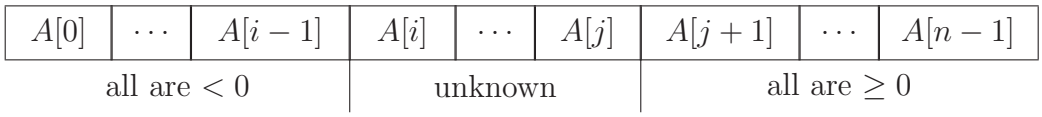
ALGORITMIEK: answers exercise class 7

Problem 1.
The following algorithm uses the partition idea similar to that of quicksort, although is is implemented somewhat differently. Namely, on each iteration, the algorithm maintains three sections (possibly empty) in a given array: all the elements in $A[0..i-1]$ are negative, all the elements in $A[i..j]$ are unknown, and all the elements in A[j+1..n-1] are nonnegative.



On each iteration, the algorithm shrinks the size of the unknown section by one element either from the left or from the right:

```
Algorithm NegBeforePos (A[0..n-1])
// Puts negative elements before positive (and zeros, if any) in an array
// Input: Array A[0..n-1] of real numbers
// Output: Array A[0..n-1] in which all its negative elements precede
//   its nonnegative elements

i = 0; j = n-1;
while (i <= j) {  // i<j would suffice
    if (A[i] < 0)   // shrink the unknown section from the left
        i = i+1;
    else {          // shrink the unknown section from the right
        swap(A[i],A[j]);
        j = j-1;
    }
}
```

An alternative implementation stays more closely to the partition idea:

- increase i until $A[i] \geq 0$
- decrease j until $A[j] < 0$
- if $i < j$, then swap (A[i],A[j])
- repeat these three steps, as long as $i \leq j$

In the first two steps, make sure not to exceed the array bounds.

Problem 2.

```
// invoer: array A[0...n-1] waarin A[i] = 'R', 'W' of 'B'
// uitvoer: array A[0...n-1] waarin eerst alle 'R' komen,
//          dan alle 'W', en ten slotte alle 'B'
r := 0;
w := 0;
```

```
b := n - 1;
while w <= b do
   if A[w] = 'R' then
      wissel( A[r], A[w] );
      r := r + 1;
      w := w + 1;
   else
      if A[w] = 'W' then
         w := w + 1;
      else  // A[w] = 'B'
         wissel( A[w], A[b] );
         b := b-1;
      fi
   fi
od
```

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`], $\ldots, A$[`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+=2) { // 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+=2) { // 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

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

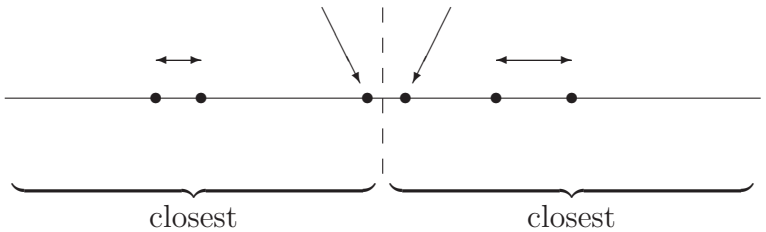
Hence,

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

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);
  }
}
```

Problem 16. We give the solutions for the first digraph.
a. During the DFS traversal, the stack looks as follows, with subscripts indicating the order in which vertices are pushed onto the stack, and the order in which vertices are popped off the stack, respectively:

$$\begin{array}{cccc} & b_{5,2} & & \\ e_{3,1} & g_{4,3} & & \\ b_{2,4} & & c_{6,5} & \\ a_{1,6} & & & d_{7,7} \end{array}$$

When we reverse the order in which the vertices are popped off the stack, we find the following topological ordering: d, a, c, b, g, b, e.
b. During the algorithm, the numbers of (remaining) incoming edges for each of the vertices evolve as follows:

a	b	c	d	e	f	g	action
1	2	2	0	2	3	2	remove d
0	1	1	-	2	2	1	remove a
-	0	0	-	2	2	1	remove b (or c)
-	-	0	-	1	2	0	remove c (or g)
-	-	-	-	1	1	0	remove g
-	-	-	-	0	0	-	remove e (or f)
-	-	-	-	-	0	-	remove f

We now find the following topological ordering: d, a, b, c, g, e, f.

Problem 17. a. Let G be a nonempty DAG, and suppose that G does **not** have any source. Let n be the number of vertices in G. Now choose an arbitrary vertex a_0. Since G does not have any source, in particular a_0 is not a source. It must thus have at least one incoming edge, say from vertex a_1. Since a_1 is not a source, either, it must have at least one incoming edge, say from vertex a_2. When we continue this way, we must encounter a certain vertex for the second time, say a_i, because G only has a finite number n of vertices. Once we encounter this vertex a_i, we have found a cycle: from a_i back to a_i (following edges in reverse direction).
As a DAG is cycle free, this is impossible. Therefore, the assumption that G does not have a source, must be invalid.
b. Suppose that the topological sorting problem has a solution for a certain digraph G. Then we can order the vertices of G in such a way that for each (directed) edge (a, b), vertex a comes before vertex b in the ordering. Then we can never have a cycle, and so G is a DAG.
Suppose that G is a DAG, and let n be the number of vertices in G. We use mathematical induction on n to prove that there exists a topological ordering of the vertices in G. In fact, the proof mimics the source-removal algorithm.
Basis. If $n = 1$, then let a_1 be the only vertex in G. Because G is acyclic, it does not contain an edge from a_1 to a_1 (and in fact, G does not contain any edge). The only possible ordering of the vertices in G is: a_1. Indeed, this is a topological ordering.
Induction step. Let $n_0 \geq 1$, and suppose that for each DAG G with n_0 vertices, there exists a topological ordering of the vertices in G. Now, consider a DAG G with $n = n_0 + 1$ vertices.

By part **a**, G must have a source, say a_1. Because a_1 is a source, it does not have any incoming edges in G. We now remove a_1 from graph G, together with its outgoing edges, leaving a graph G' that is still acyclic, but has only n_0 vertices.
By the induction hypothesis, there exists a topological ordering of the vertices in G', say $a_2, a_3, \ldots, a_{n_0+1}$. Now, the ordering $a_1, a_2, a_3, \ldots, a_{n_0+1}$ is a topological ordering of the vertices in G, because each edge in G

- either is an outgoing edge of a_1, i.e. an edge from a_1 to a later vertex in the ordering,
- or is an edge (a_i, a_j) that also exists in G', which implies that $2 \leq i < j \leq n_0 + 1$, because $a_2, a_3, \ldots, a_{n_0+1}$ is a topological ordering of the vertices in G'.