Duration: 150 minutes.
External Material: you are not allowed to use any written, printed or electronic external material.

Guidelines and Instructions:
- Write your answers only on the exam form, in the intended places. Note that the given place doesn't necessarily indicate the length of the correct answer.
- The even pages of the exam form are empty. You can use them as a draft or to write your answers. Write drafts clearly, so they won't be checked.
- Write your answers tidy, clean and clearly as possible. You're allowed to use pencil and an eraser, but you must fill the title page in pen.
- In all of the questions, you're allowed to define (and implement) your own functions.
- You are not allowed to use global and/or static variables, or pre-compilation commands (include or define).
- You are not allowed to use library functions, or functions implemented in class, without implementing them yourself, unless noted explicitly in the question, excluding input/output functions.
- In every question, you are allowed to use functions defined in previous parts of the same question, even if you didn’t solve these parts, though this is not obligated.
- You don’t need to check input correction, unless explicitly noted in the question.
- You don’t need to implement the main function, and the order of writing the functions is not important.
- the complexity of the solution will not be marked, unless explicitly stated.
- It's recommended to add a written explanation of your algorithm. That explanation will not be marked.
Question 1 (30 points)
Part 1 (15 Points)
Write the following iterative function:

```c
void strangeOrder(int source[], int target[], int size)
```

Its parameters are an integer array `source`, sorted from largest to smallest, and an integer array `target`. Size is the size of the two arrays (it's not necessarily even). The function puts the elements of `source` into `target` in the following way:
The largest element of `source` will be in the beginning of `target`. The second largest in `source` will be in the end of `target`. The third largest will be in the second place of `target`, the fourth in the one before last place of `target`, etc.

Example: for the following `source`:

| 99 | 54 | 45 | 44 | 30 | 23 | 21 | 18 | 15 | 3 | 1 |

`target` will be:

| 99 | 45 | 30 | 21 | 15 | 1 | 3 | 18 | 23 | 44 | 54 |

You are not allowed to use recursion at all.

```c
int* s = source;
int* t = target;

while(size > 0) {
    t[0] = s[0];
    if(size > 1) {
        t[size - 1] = s[1];
    }
    t += 1;
    s += 2;
    size -= 2;
}
```
Part 1 (15 Points)
Write the following recursive function:

```c
void strangeOrderRec(int source[], int target[], int size)
```

Its parameters are an integer array `source`, sorted from largest to smallest, and an integer array `target`. `Size` is the size of the two arrays (it's not necessarily even). The function puts the elements of `source` into `target` in the following way:
The largest element of `source` will be in the beginning of `target`. The second largest in `source` will be in the end of `target`. The third largest will be in the second place of `target`, the fourth in the one before last place of `target`, etc.

**Example:** for the following `source`:

<table>
<thead>
<tr>
<th>99</th>
<th>54</th>
<th>45</th>
<th>44</th>
<th>30</th>
<th>23</th>
<th>21</th>
<th>18</th>
<th>15</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
</table>

target will be:

<table>
<thead>
<tr>
<th>99</th>
<th>45</th>
<th>30</th>
<th>21</th>
<th>15</th>
<th>1</th>
<th>3</th>
<th>18</th>
<th>23</th>
<th>44</th>
<th>54</th>
</tr>
</thead>
</table>

You are not allowed to use loops at all.

```c
void strangeOrderRec(int source[], int target[], int size){

if(size <= 0) {
    return;
}

    target[0] = source[0];
    if(size > 1) {
        target[size - 1] = source[1];
        strangeOrderRec(source + 2, target + 1, size - 2);
    }
}
```
Question 2 (20 points)
The office of measurements performs measurements in many different locations in Israel. The measurement always begins in a place higher than sea level, and ends in a place lower than sea level. The results of each measurement are saved in an array of doubles. Notice that the values in the array are not a decreasing series (notice the examples). Also, it's guaranteed that the value in the first cell (index 0) is always positive, and the value in the last cell is always negative. It's also guaranteed that the value 0.0 is not in the array.

There is a single index, k, in the array, that has $a[k]>0$, $a[k+1]<0$.

Write the following function:

```c
int SeaLevel(double ar[], int len)
```

Its parameters are an array of doubles, ar, and its length, len. The function returns the index k, defined above.

Examples:
For the following array (results of the measurement between Jerusalem and Ein-Gedi), and its length, 12, the function will return 2.

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[i]</td>
<td>888.8</td>
<td>503.8</td>
<td><strong>200.0</strong></td>
<td>-20.8</td>
<td>-51.0</td>
<td>-34.6</td>
<td>-123.4</td>
<td>-78.1</td>
<td>-51.4</td>
<td>-267.5</td>
<td>-100.1</td>
<td>-398.3</td>
</tr>
</tbody>
</table>

For the following array (results of the measurement between Mount Tabor and Tiberias), and its length, 11, the function will return 6.

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[i]</td>
<td>555.5</td>
<td>345.6</td>
<td>499.9</td>
<td>111.1</td>
<td>240.7</td>
<td>123.9</td>
<td><strong>3.5</strong></td>
<td>-1.2</td>
<td>-98.4</td>
<td>-150.4</td>
<td>-212.1</td>
</tr>
</tbody>
</table>

Note that a solution that uses a serial (linear) search of the array will only get a small part of the points, because the arrays the function will work on are very large.

```c
int SeaLevel(double ar[], int len) {
```
A recursive solution:
   if (len == 2) return 0; /* cannot be < 2 */
   mid = len/2;
   if (ar[mid] < 0) return SeaLevel(ar, mid + 1);
   return mid + SeaLevel(ar + mid, len - mid);

An iterative solution:
   int l = 0, r = len - 1, mid = (l + r)/2;
   while (mid > l) {
      if (ar[mid] > 0) {
         l = mid;
      } else {
         r = mid;
      }
      mid = (l + r)/2;
   }
   return l; /* Same as return mid */

A sequential solution like the following lost 13 points:

   int i = 0;
   while (ar[i] > 0) {
      ++i;
   }
   return i - 1;
**Question 3 (20 Points)**

Write the following function:

```c
void sortDirtyString(char* str)
```

Its parameter is a string, `str`, that contains only lowercase letters and digits. The function sorts the letters by lexicographical order from smallest to largest. All the digits remain in their original locations, and in the original order. You're not allowed to use extra arrays.

**Examples:**
- For `str="gr4ea5t"` the function will change `str` to "ae48gr5t"
- For `str="8dc95ga9"` the function will change `str` to "8ac95dg9"

```c
void sortDirtyString(char* str){
    int i, j, temp, min;
    for (i = 0; str[i+1] != '\0'; i++)
    {
        if(str[i]>='a' && str[i] <= 'z')
        {
            min = i;
            for (j = i+1; str[j] != '\0'; j++)
            {
                if(str[j]>='a' && str[j] <= 'z' && str[j] <str [min])
                {
                    min = j;
                }
            }
        }
        temp = str[i];
        str[i] = str[min];
        str[min] = temp;
    }
}
```
**Question 4 (30 points)**

A "pyramidal" number is a positive number. Its digits are **ascending** (<, not ≤) until the largest digit, and from it its digits are **decreasing** (> , not ≥). The largest digit is called the "peak of the pyramid", and there is only one peak for every "pyramidal" number.

**Examples:**
- The following numbers are pyramidal: 258763 (the peak is 8), 1 (the peak is 1), 123 (the peak is 3), 321 (the peak is 3), 454 (the peak is 5).
- The following numbers are not pyramidal: 569881 (8 appears twice), 12330 (3 appears twice), 124372 (there are two peaks), 987689 (the digits are decreasing, then ascending).

**Part 1 (15 points)**

Write the following function:

```
int topOfPyramid(unsigned int num){

int current, next, max;

    current = num%10;
    next = (num/10)%10;
/* Going over the part of num that "goes up"*/
    while(current < next){
        num/=10;
        current = num%10;
        next = (num/10)%10;
    }
/* current should be the maximum if this is indeed a pyramid*/
    max = current;
/* Going over the part of num that "goes down"*/
    while(num){
        if (current <= next) return -1;
        num/=10;
        current = num%10;
        next = (num/10)%10;
    }

    return max;
}
```
Part 2 (15 points)
A "pyramidal ridge" is a matrix of positive integers, where all the numbers in it are pyramidal, and the largest peak of every row is the same for all the rows.

You are not allowed to change the array. You can use part 1 even if you didn't solve it.

Examples: the following matrix is a pyramidal ridge:

\[
\begin{bmatrix}
12432 & 1232 & 4821 \\
128765 & 4565 & 676 \\
\end{bmatrix}
\]
The largest peak in the first row is 8, and the largest peak in the second row is also 8. Therefore it is a pyramidal ridge.

The following matrix is not a pyramidal ridge:

\[
\begin{bmatrix}
12432 & 1232 & 4821 \\
129765 & 4565 & 676 \\
343 & 454 & 565 \\
\end{bmatrix}
\]
The largest peak in the first row is 0. The largest peak in the second row is 9, so the largest peak is different in the first two rows; therefore it is not a pyramidal ridge.

The following matrix is not a pyramidal ridge:

\[
\begin{bmatrix}
12323 & 4821 \\
4565 & 676 \\
\end{bmatrix}
\]
Because the number in cell [0][0] is not pyramidal.

Write the following function:

```c
int pyramidRidge(unsigned int a[M][N]) {
```

Its parameter is a matrix of positive integers \( a \), of size \( M \times N \) (defined in a define statement). The function returns 1 if \( a \) is a pyramidal ridge, and 0 if it's not.
```c
int pyramidRidge(unsigned int a[M][N])
{
    /* SOLUTION1*/
    unsigned int top0;
    unsigned int curr_top, next;
    int i,j;

    top0 = 0;
    for (j=0;j<N;++j) {
        next = topOfPyramid(a[0][j]);
        if (next == -1) return 0;
        if (top0 < next) top0 = next;
    }
    for (i=1;i<M;++i) {
        curr_top = 0;
        for (j=0;j<N;++j) {
            next = topOfPyramid(a[i][j]);
            if (next == -1) return 0;
            if (curr_top < next) curr_top = next;
        }
        if (curr_top != top0) return 0;
    }
    return 1;
    /* end of SOLUTION1 */
}
```
/ * SOLUTION2 */

int topOfRow(unsigned int a[M][N], int i) {
    unsigned int curr_top;
    unsigned int next;
    int j;

    curr_top = 0;
    for (j=0; j<N; ++j) {
        next = topOfPyramid(a[i][j]);
        if (next == -1) return -1;
        if (curr_top < next) curr_top = next;
    }
    return curr_top;
}

int pyramidRidge(unsigned int a[M][N]) {
    unsigned int top0;
    top0 = topOfRow(a, 0);
    if (top0 == -1) return 0;
    for (i=1; i<M; ++i) {
        if (topOfRow(a, i) != top0) return 0;
    }
    return 1;
}

/*end of solution 2*/