1. There is a set P = {'a', 'b', 'c', ..., 'z'} of small English letters. The task is to create table T with 8 columns. Each row of T will contain one 8-element subset of P and each cell in the row will contain exactly one letter. All 8-element subsets of P will be listed in T and no subset will be listed twice. Find the size of the table and determine if it is possible for your personal computer to fill it completely in less than 1 second.

I denote $\binom{n}{k}$ by Comb($n$, $k$) in the text, as it is easier to type.

There are 26 letters in P, there are Comb(26, 8) 8-letter subsets of P. Therefore T has
$8 \cdot$ Comb(26, 8) = 8 $\cdot$ 1562275 = 12498200 $\approx 12.5 \cdot 10^6$ entries.
To fill the table in approximately one second each entry has to be generated in constant time.
To verify that a constant time is spent on each entry, formulate an appropriate recusrive algorithm which fills the table.

An example which repeatedly fills an auxiliary array named subset is below. To fill the table, it would be enough to add another global parameter which denotes the table row and in the print part of the code fill the appropriate row with the contents of the auxilliary array.

```
void recksubsets( int k, vector<int> & Set, vector<int> & subset, int iset, int isubset ){
  if( isubset == k ) {
    for( int i = 0; i < k; i++ ) cout << " " << subset[i];
    cout << endl;
    return;
  }

  // either include a particular set element in the subset ...
  if( isubset < k ) { // ... and do not overflow the subset capacity
    subset[isubset] = Set[iset];
    recksubsets(k, Set, subset, iset+1, isubset+1);
  }
  // ... or do not include this element
  if( (Set.size()-iset-1) + isubset >= k ) // a chance to make k-subset?
    recksubsets(k, Set, subset, iset+1, isubset );

}

// wrapper for the recursive call
void ksubsets( int n, int k ) {
  vector<int> Set (n);
  for( int i = 0; i < n; i++ ) Set[i] = i;
  vector<int> subset (k);
  recksubsets( k, Set, subset, 0, 0 );
}
```

The recursive code fills each slot in the auxilliary array in constant time,
transfer of a value from the auxiliary array to the resulting table also happens in constant time.

2. Write a pseudocode of a function which will print out all unempty subsets of the set {0, 1, 2, ..., *n*−1}.

Recirsive function is shorter to construct:
Create an array *Sub* of size *n*.

```
function allSubsets( int posInSub, int setElement ){
   if( setElemen >= n ) print Sub[0 ... posInSub-1] and return
  // else either incluse setElement in Sub or not include it and call recursion in both cases
  /*include*/  Sub[posInSub] = setElement; allSubsets( posInSub+1,  setElement+1);
  /*not include*/  allSubsets( posInSub,  setElement+1);
}
```

Start with posInSub = 0 and setElement = 0.

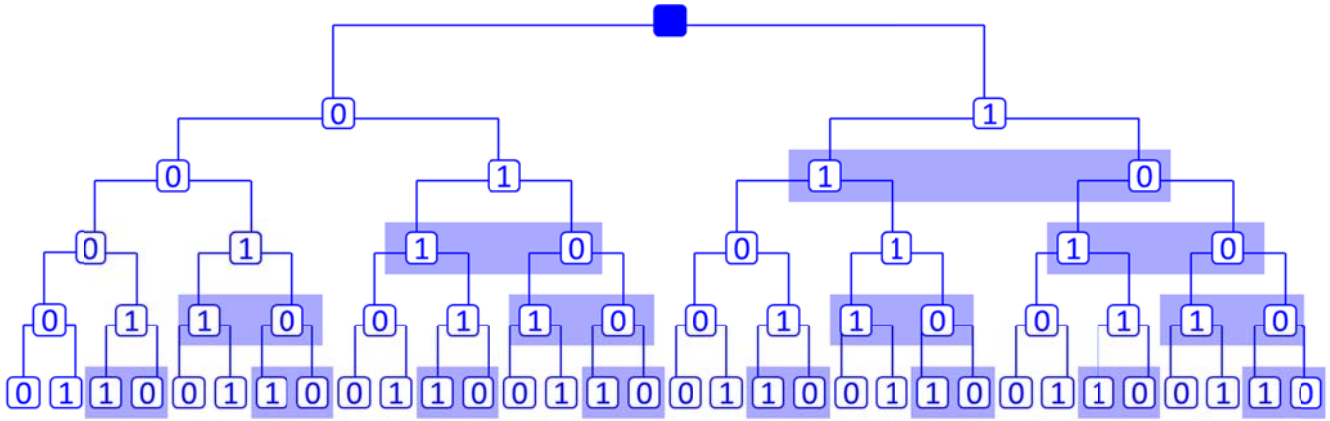
3. Consider permutations of the set M = {1, 2, 3, ..., $n$}, $n > 4$. A permutation $p$ of M is said to be *cheerful* if the following holds: $p(3) \in \{3, n\}$; $p(n) \in \{3, n\}$; $p(1) = 1$; $p(2) = 2$; $p(i) \in \{4, ..., n-1\}$ for $i = 3, 4, ..., n-1$.
Find the number of all *cheerful* permutations of M.

There are two possibilities of mapping the set {3, $n$}: (3, $n$) --> (3, $n$) or (3, $n$) --> ( $n$, 3).
There are $(n-4)!$ ways to permute the set {4, ..., $n-1$}.
Therefore, in total, we have $2 \cdot (n-4)!$ *cheerful* permutations.

4. Suppose that every element of Gray code $G^n$ (i.e. *n*-touple of 0's and 1's) is stored in a character array of length *n*. Write a pseudocode of a function which prints out the complete Gray code $G^n$.

Lemma 1 on slide 8 can be used to generate trivially Gray code in two nested loops.

Another approach is to use the recusrive definition and draw a tree in which nodes on each branch from the root to a leaf correspond to one n-tuple in $G^n$ code.

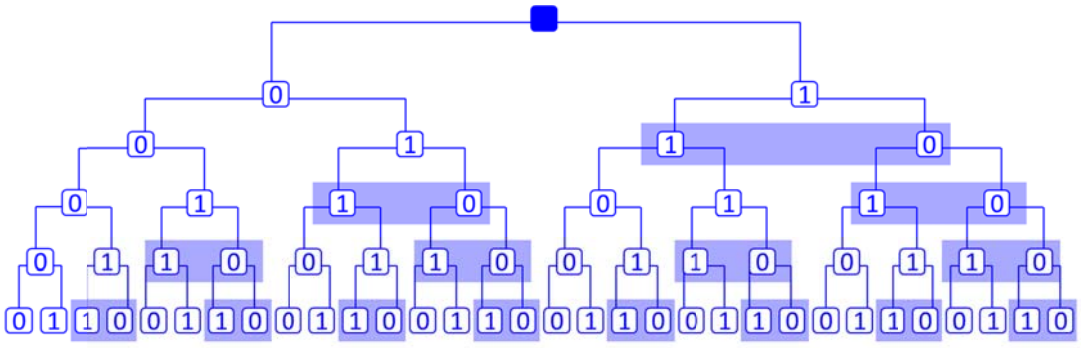



Note that the children of a left child of a particular parent are 0 and 1 while the children of the right child of a particular parent are 1 and 0, the latter pairs are highlighted. This leads to the obvious recursive procedure which which uses only one array of size *n*.

```
void Gray( int n, vector<int> GrayCode, bool firstcall ){
  if ( n == 0 ) {
    for( int i = GrayCode.size()-1; i >= 0; i-- ) cout << " " << GrayCode[i];
    cout << endl;
    return;
  }

  if( firstcall ) {
  GrayCode[n-1] = 0; Gray( n-1, GrayCode, true );
  GrayCode[n-1] = 1; Gray( n-1, GrayCode, false );
  }
  else {
  GrayCode[n-1] = 1; Gray( n-1, GrayCode, true );
  GrayCode[n-1] = 0; Gray( n-1, GrayCode, false );
  }
}

void Gray( int n ) {
  vector<int> GrayCode (n);
  Gray( n, GrayCode, true );
}
```



5. Set M contains 98 elements. Each permutation of M is ranked by an unique integer in the range from 0 to 98!−1. The program represents each permutation by its rank. We know that in any moment the program will store at most 100 permutations of M and therefore it will allocate static memory for just 100 ranks of those permutations.
What is the minimum number of bits needed to store any 100 of ranks?

The maximum rank is 98!−1. The number of its binary digits equals to the upper whole part of
$\lg(98!) = \lg(1) + \lg(2) + \lg(3) + \ldots + \lg(98) = 511.4917804...$
(neither of 98! and 98!−1 is a power of 2)
which is 512.
We will need $100 \cdot 512 = 51\,200$ bits, that is 6 400 bytes.

6. A sequence P = (000, 001, 011, 010, 110, 111, 101, 100) represents a Gray Code $G^3$.
Two finite sequences are said to be equivalent if:

1. A reversed is equal to B, or
2. Left or right rotation of A by any number of positions is equal to B, or
3. There exists sequence C equivalent to A and B.

Find an 8-element sequence Q which is a Gray code and which is not equivalent to P. Here we define Gray code to be any a binary system where each two neighbour codes differ in only one bit.

Take a 3-dim cube of unit volume with edges parallel to axes and which one corner sits in (0,0,0) and the oposite corner sits in (1,1,1) .
There are 8 vertices of the cube and their coordinates represent all possible triples of 0's and 1's.
A Gray code is obtained when we start in any vertext and then walk around the cube along its edges
visiting each node exactly once. (Here you may pause and let the the students figure out the rest.)
The code depends on sequence of nodes we choose.
When we start with (0,0,0) --> (0,0,1) there are 4 possibilities:
(0,0,0) --> (0,0,1) --> (0,1,1) --> (0,1,0) --> (1,1,0) --> (1,1,1) --> (1,0,1) --> (1,0,0)
(0,0,0) --> (0,0,1) --> (0,1,1) --> (1,1,1) --> (1,0,1) --> (1,0,0) --> (1,1,0) --> (0,1,0)
(0,0,0) --> (0,0,1) --> (1,0,1) --> (1,0,0) --> (1,1,0) --> (1,1,1) --> (0,1,1) --> (0,1,0)
(0,0,0) --> (0,0,1) --> (1,0,1) --> (1,1,1) --> (0,1,1) --> (0,1,0) --> (1,1,0) --> (1,0,0)
We can also start with with (0,0,0) --> (0,1,0) or with with (0,0,0) --> (1,0,0). Each of these will bring
analogously four possibilities. When we take all these $4 \cdot 3 = 12$ paths, there will be each path present twice:
in forward and backward direction. As we count a reversed sequence to be equal to the original one there are
6 distinct sequences -- 6 distinct codes equivalent to the Gray code, one of them the Gray code itself, so there are another 5 sequences equivalent to the Gray code $G^3$.

Take a 3-dim cube of unit volume with edges parallel to axes and which one corner sits in (0,0,0) and the oposite corner sits in (1,1,1) .
A Gray code is obtained when we start in any vertext and then walk around the cube along its edges visiting each node exactly once.
When we start with (0,0,0) --> (0,0,1) there are 4 possibilities:

(0,0,0) --> (0,0,1) --> (0,1,1) --> (0,1,0) --> (1,1,0) --> (1,1,1) --> (1,0,1) --> (1,0,0)
(0,0,0) --> (0,0,1) --> (0,1,1) --> (1,1,1) --> (1,0,1) --> (1,0,0) --> (1,1,0) --> (0,1,0)
(0,0,0) --> (0,0,1) --> (1,0,1) --> (1,0,0) --> (1,1,0) --> (1,1,1) --> (0,1,1) --> (0,1,0)
(0,0,0) --> (0,0,1) --> (1,0,1) --> (1,1,1) --> (0,1,1) --> (0,1,0) --> (1,1,0) --> (1,0,0)

We can also start with with (0,0,0) --> (0,1,0) or with with (0,0,0) --> (1,0,0). Each of these will bring analogously four possibilities.
When we take all these $4 \cdot 3 = 12$ paths, there will be each path present twice: in forward and backward direction.
As we count a reversed sequence to be equal to the original one there are 6 distinct sequences -- 6 distinct codes equivalent to the Gray code, one of them the Gray code itself, so there are another 5 sequences equivalent to the Gray code $G^3$.

7. Consider all *k*-element subsets of set M = {1, 2, 3, ..., *n*}, $1 \leq k \leq n$. There exists an algorithm which transforms a list of elements of one subset into a list of elements of the next subset in the lexicographical ordering of all *k*-element subsets of M. Your task is to design a reverse algorithm, i.e. an algorithm which which transforms a list of elements of one subset into a list of elements of the previous subset in the lexicographical ordering of all *k*-element subsets of M. Will the asymptotic complexity of both transformations be the same?

Consider the elements of the subset stored in the array in always in ascending order.
The code of the algorithm "k-subset--> next lexicographically bigger k-subset" is below, on the left.
A possible but not effective approach to the solution would be:

1. Reverse the order of elements in the subset.
2. Run the method which yields the next lexicographically *bigger* subset.
3. Perform 1. to obtain appropriate order of elements in the subset.

The complexity of this approach would be O(*k*). Note, that the *amortized* complexity of the original method is O(1).
The example code on the right represents the solution which runs also in O(1) amortized complexity.

```
def nextSubset( subset,  n ):
  j = len(subset)-1
  maxval = n
  while True:
    if j < 0: return False
    if subset[j] == maxval:
      # move left
      maxval -= 1
      j -= 1
    else:
      subset[j] += 1
      for j2 in range(j+1, len(subset)):
        subset[j2] = subset[j2-1]+1
      return True
```

```
def prevSubset( subset, n ):
  j = len(subset)-1
  while True:  # try to move right
    if j == 0:
      if subset[j] == 1: return False
      else: break
    else:
      if subset[j-1] + 1 < subset[j]: break
      else: j -= 1
  subset[j] -= 1
  # set the values to the right of j
  # to be  ..., n-2, n-1, n
  val = n
  for j2 in range(len(subset)-1, j, -1):
    subset[j2] = val
    val -= 1
  return True
```

8A. Consider permutations of set $M = \{1, 2, 3, ..., n\}$. We define a cycle of length $k$ in permutation $p$ to be a set $A = \{a_1, a_2, ..., a_k\} \subseteq M$, for which holds:

$p(a_j) = a_{j+1}$ for $1 \leq j < k$; $p(a_k) = a_1$.

Determine the number of such permutations of M which contain exactly two cycles and moreover the length of one cycle is 4 and the length of the other cycle is $n-4$.

First, let us compute the number of cycles of length $k$ in case when the particular sets of elements in the cycle is fixed. Take the smallest element in the set as the start point of a cycle. There are only $k-1$ posibilities left how to choose the next element in the cycle. After that, choose the third element in the cycle, for which there are only $k-1$ possibilities. And so on. This will result in $(k-1)!$ possibilities. A set of four elements can be chosen in Comb($n$, 4) ways from M. So, there are Comb($n$, 4) · 3! cycles of length 4 in M. The remaining set of $n-4$ elements can be ordered in a cycle in $(n-5)!$ ways. Put together, the nuber of permutations with the given property is equal to

Comb($n$, 4) · 3! · $(n-5)!$

The formula can be simplified using the definition of factorial nad binomial coefficient to

$n!/(4 \cdot (n-4))$.

8B. Consider permutations of set $M = \{1, 2, 3, ..., n\}$. We define an ordered cycle of length $k$ in permutation $p$ to be an ordered set $A = \{a_1, a_2, ..., a_k\} \subseteq M$, for which holds:

$1 \leq a_1 < a_2 < ... < a_k \leq n$; $p(a_j) = a_{j+1}$ for $1 \leq j < k$; $p(a_k) = a_1$.

Determine the number of such permutations of M which contain exactly two ordered cycles and moreover the length of one ordered cycle is 4 and the length of the other ordered cycle is $n-4$.

Each four-element subset $\{a1, a2, a3, a4\} \subseteq M$ defines one ordered cycle. When this cycle is specified, the remaining n−4 elements (the complement of the set $\{a1, a2, a3, a4\}$ in M) also specify one ordered cycle. Thus, whenever we choose $\{a1, a2, a3, a4\}$, the whole permutation is completely specified. The number of permutations we are asked to find is equal to the number of different subsets $\{a1, a2, a3, a4\}$ in M, and that is clearly Comb(n, 4).

9. Rank of a permutation $\pi$ of set N = {0, 1, 2, ..., $n$−1} is the index of $\pi$ in the list of all permutations of N. The list is sorted in increasing lexicographical order and is indexed from 0. Write a pseudocode of a function which will print out the permutation which has rank $n!/2$ and will do it in time proportional to $n$. Suppose $n \geq 2$.

Denote the given permutation by P.
P is the first permutation in the second half of the list of all permutation.
There are two cases: $n$ is even and $n$ is odd. When n is even, the second half of the list starts with the permutation
($n/2$, 0, 1, ..., $n/2$−1, $n/2$+1, $n/2$+2, ... $n$−1) (With the exception for $n$ = 2, where the permutation is (1, 0) ).
When n is odd, the first entry in P is $(n-1)/2$. Now, P is the first permutation in the second half of the sub-list of all permutations which first entry is also $(n-1)/2$. (If necessary, draw yourself an example with $n$ = 5 or $n$ = 7).
In this sub-list, the second entry runs from 0 to $n$−1, omitting $(n-1)/2$ which is unavailable. The second half of the sub-list thus starts with permutation P which has $(n-1)/2 + 1 = (n+1)/2$ as a second entry. P is thus
( $(n-1)/2$, $(n+1)/2$, 0, 1, 2, ..., $(n-1)/2 - 1$, $(n+1)/2+1$, ..., $n/2$).
I do not present a pseudocode for generating sequences
($n/2$, 0, 1, ..., $n/2$−1, $n/2$+1, $n/2$+2, ..., $n$−1) and
( $(n-1)/2$, $(n+1)/2$, 0, 1, 2, ..., $(n-1)/2 - 1$, $(n+1)/2+1$, ..., $n/2$), it should be a trivial task for a student of master degree.

10 . Consider all permutations of set $M = \{1, 2, 3, ..., n\}$, $1 \leq k \leq n$. There exists an algorithm which transforms one permutation into the permutation which is next the lexicographical ordering of all permutations of M. Your task is to design a reverse algorithm, i.e. an algorithm which transforms one permutation into the permutation which is previous the lexicographical ordering of all permutations of M. Will the asymptotic complexity of both transformations be the same?

An obvious, but not effective, modification would be:
1. substitute each element x in permutation by $n - x$.
2. run the method which yields the next lexicographically *bigger* permutation.
3. perform 1. to obtain original values in the permutation.
The complexity of this approach would be $O(k)$. Note, that the *amortized* complexity of the original method is O(1).
The example code on the right represents the solution which runs also in O(1) amortized complexity.

To work effectively, consider the original algorithm and reverse those inequality signs which compare the values of two elements in the permutation. In the code on the next page, there are just two such inequalities, no other changes in the code are needed.

```python
def nextPerm( perm ):
    n = len( perm )
    j = n-1 # last index

    # 1. find shortest non-descending suffix
    # of length >= 2
    while True:
        if j == 0: return False # no next permutation
        if perm[j-1] < perm[j]: break
        j -= 1
    j -= 1

    # 2. swap perm[j] with minimum bigger
    # than perm[j] in the suffix after j
    j2 = n-1
    while perm[j] > perm[j2]: j2 -= 1
    perm[j], perm[j2] = perm[j2], perm[j]  # the swap

    # 3. arrange suffix after j into ascending order
    j += 1
    j2 = n-1
    while j < j2:
        perm[j], perm[j2] = perm[j2], perm[j]
        j += 1
        j2 -= 1
    return True
```

```python
def prevPerm( perm ):
    n = len( perm )
    j = n-1 # last index

    # 1. find shortest non-ascending suffix
    # of length  >= 2
    while True:
        if j == 0: return False # no next permutation
        if perm[j-1] > perm[j]: break
        j -= 1
    j -= 1

    # 2. swap perm[j] with maximum smaller
    # than perm[j] in the suffix after j
    j2 = n-1
    while perm[j] < perm[j2]: j2 -= 1
    perm[j], perm[j2] = perm[j2], perm[j]  # the swap

    # 3. arrange suffix after j into descending order
    j += 1
    j2 = n-1
    while j < j2:
        perm[j], perm[j2] = perm[j2], perm[j]
        j += 1
        j2 -= 1
    return True
```

11. Permutation $p$ of set M is called derangement, if for each $k \in M$ holds
$1 \le k \le n \Rightarrow p(k) \neq k$. Write down all derangements of the set {1, 2, 3, 4}.
Find 1000000-th element in the lexicographical ordering of derangements of set {1, 2, 3, ..., 20}.

The derangements of the set {1, 2, 3, 4} are:
(2 1, 4, 3), (2, 3, 4, 1), (2, 4, 1, 3), (3, 1, 4, 2), (3, 4, 1, 2), (3, 4, 2, 1), (4, 1, 2, 3), (4, 3, 1, 2), (4, 3, 2, 1).

The second question demands either some more advanced approach or just a brute force implementation.
The brute force approach is trivial, generate all permutations of {1, 2, 3, ..., 20} in lexicographic order,
count each permutation which is a derangement and stop when the 1000000-th derangement is found.
(2, 1, 4, 3, 6, 5, 8, 7, 10, 9, 18, 17, 20, 16, 14, 12, 15, 19, 11, 13)
The generation shoud start at very the first derangement
( 2, 1, 4, 3, 6, 5, 8, 7, 10, 9, 12, 11, 14, 13, 16, 15, 18, 17, 20, 19 ),
to avoid generation of at least 19! = 121 645 100 408 832 000 permutations which have 1 at the beginning and therefore are not derangements.
Utilize, for example, the code which generates the next permutation in the previous example.