Parrots

Yanee is a bird enthusiast. Since reading about *IP over Avian Carriers* (IPoAC), she has spent much of her time training a flock of intelligent parrots to carry messages over long distances.

Yanee's dream is to use her birds to send a message **M** to a land far far away. Her message **M** is a sequence of **N** (not necessarily distinct) integers, each between 0 and 255, inclusive. Yanee keeps **K** specially-trained parrots. All the parrots look the same; Yanee cannot tell them apart. Each bird can remember a single integer between **0** and **R**, *inclusive*.

Early on, she tried a simple scheme: to send a message, Yanee carefully let the birds out of the cage one by one. Before each bird soared into the air, she taught it a number from the message sequence in order. Unfortunately, this scheme did not work. Eventually, all the birds did arrive at the destination, but they did not necessarily arrive in the order in which they left. With this scheme, Yanee could recover all the numbers she sent, but she was unable to put them into the right order.

To realize her dream, Yanee will need a better scheme, and for that she needs your help. Given a message **M**, she plans to let the birds out one by one like before. She needs you to write a program that will perform two separate operations:

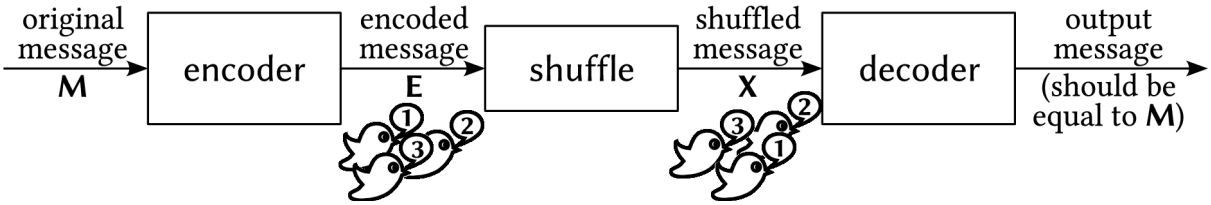
- First, your program should be able to read a message **M** and transform it into a sequence of at most **K** integers between **0** and **R** that she will teach the birds.
- Second, your program should be able to read the list of integers between **0** and **R** received as the birds reach their destination, and then transform it back to the original message **M**.

You may assume that all parrots always arrive at the destination, and that each of them remembers the number it was assigned. Yanee reminds you once again that the parrots may arrive in any order. Note that Yanee only has **K** parrots, so the sequence of integers between **0** and **R** that you produce must contain at most **K** integers.

Your task

Write two separate procedures. One of them will be used by the sender (encoder) and the other by the receiver (decoder).

The overall process is shown in the following figure.



The two procedures you are to write are:

- Procedure **encode(N,M)** that takes the following parameters:
  - **N** – the length of the message.
  - **M** – a one-dimensional array of **N** integers representing the message. You may assume that **0** ≤ **M[i]** ≤ **255** for **0** ≤ **i** < **N**.

This procedure must encode the message **M** into a sequence of integers between **0** and **R**, inclusive, that shall be sent using the parrots. To report this sequence, your procedure **encode** must call the procedure **send(a)** for each integer **a** that you wish to give to one of the birds.

- Procedure **decode(N,L,X)** that takes the following parameters:
  - **N** – the length of the original message.
  - **L** – the length of the message received (the number of birds that were sent).
  - **X** – a one-dimensional array of **L** integers representing the received numbers. The numbers **X[i]** for **0** ≤ **i** < **L** are precisely the numbers that your procedure **encode** produced, but possibly rearranged into a different order.

  This procedure must recover the original message. To report it, your procedure **decode** must call the procedure **output(b)** for each integer **b** in the decoded message, in the correct order.

Note that **R** and **K** are not given as input parameters – please see the subtask descriptions below.

In order to correctly solve a given subtask, your procedures must satisfy the following conditions:

- All integers sent by your procedure **encode** must be in the range specified in the subtask.
- The number of times your procedure **encode** calls the procedure **send** must not exceed the limit **K** specified in the subtask. Please note that **K** depends on the length of the message.
- Procedure **decode** must correctly recover the original message **M** and call the procedure **output(b)** exactly **N** times, with **b** equal to **M[0]**, **M[1]**, ..., **M[N-1]**, respectively.

In the last subtask, your score varies according to the ratio between the lengths of the encoded message and the original message.

Example

Consider the case where **N** = **3**, and

M= 10
30
20

Procedure **encode(N,M)**, using some strange method, may encode the message as the sequence of numbers **(7, 3, 2, 70, 15, 20, 3)**. To report this sequence, it should call the procedure **send** as follows:

send(7)
send(3)
send(2)
send(70)
send(15)
send(20)
send(3)

Once all parrots reach their destination, assume we obtain the following list of transcribed numbers: **(3, 20, 70, 15, 2, 3, 7)**. The procedure **decode** will then be called with **N**=**3**, **L**=**7**, and

```
    3
    20
    70
X=  15
    2
    3
    7
```

The procedure **decode** must produce the original message **(10, 30, 20)**. It reports the result by calling the procedure **output** as follows.

```
output(10)
output(30)
output(20)
```

Subtasks

Subtask 1 (17 points)

- **N** = **8**, and each integer in the array **M** is either 0 or 1.
- Each encoded integer must be in the range from **0** to **R**=**65535**, inclusive.
- The number of times you can call the procedure **send** is at most **K**=**10**×**N**.

Subtask 2 (17 points)

- **1** ≤ **N** ≤ **16**.
- Each encoded integer must be in the range from **0** to **R**=**65535**, inclusive.
- The number of times you can call the procedure **send** is at most **K**=**10**×**N**.

Subtask 3 (18 points)

- **1** ≤ **N** ≤ **16**.
- Each encoded integer must be in the range from **0** to **R**=**255**, inclusive.
- The number of times you can call the procedure **send** is at most **K**=**10**×**N**.

Subtask 4 (29 points)

- **1** ≤ **N** ≤ **32**.
- Each encoded integer must be in the range from **0** to **R**=**255**, inclusive.
- The number of times you can call the procedure **send** is at most **K**=**10**×**N**.

Subtask 5 (up to 19 points)

- $16 \le N \le 64$.
- Each encoded integer must be in the range from **0** to **R=255**, inclusive.
- The number of times you can call the procedure **send** is at most $K=15 \times N$.
- **Important**: the score for this subtask depends on the ratio between the length of the encoded message and that of the original message.

  For a given test case **t** in this subtask, let $P_t=L_t/N_t$ be the ratio between the length L_t of the encoded message and the length N_t of the original message. Let **P** be the maximum of all P_t. Your score for this subtask will be determined using the following rules:

  - If $P \le 5$, you get the full score of **19** points.
  - If $5 < P \le 6$, you get **18** points.
  - If $6 < P \le 7$, you get **17** points.
  - If $7 < P \le 15$, your score is $1 + 2 \times (15 - P)$, rounded down to the nearest integer.
  - If $P > 15$ or *any* of your outputs is incorrect, your score is **0**.

- **Important**: Any valid solution for subtasks 1 to 4 will also solve all preceding subtasks. However, due to the larger bound on **K**, a valid solution to subtask 5 might not be able to solve subtasks 1 to 4. It is possible to solve all subtasks using the same solution.

Implementation details

Limits

- Grading Environment: In the real grading environment, your submissions will be compiled into two programs **e** and **d** to be executed separately. Both your encoder and decoder modules will be linked to each executable, but **e** only calls **encode** and **d** only calls **decode**.
- CPU time limit: Program **e** will make 50 calls to procedure **encode** and it should run in 2 seconds. Program **d** will make 50 calls to procedure **decode** and it should run in 2 seconds.
- Memory limit: 256 MB
  Note: There is no explicit limit for the size of stack memory. Stack memory counts towards the total memory usage.

Interface (API)

- Implementation folder: parrots/
- To be implemented by contestant:
  - encoder.c or encoder.cpp or encoder.pas
  - decoder.c or decoder.cpp or decoder.pas

  Note for C/C++ programmers: both in the sample grader and in the real grader, encoder.c[pp] and decoder.c[pp] are linked together with the grader. Therefore, you should declare all global variables inside each file as **static** to prevent them from interfering with variables from other files.

- Contestant interface:
  - encoder.h or encoder.pas
  - decoder.h or decoder.pas
- Grader interface:
  - encoderlib.h or encoderlib.pas
  - decoderlib.h or decoderlib.pas
- Sample grader: grader.c or grader.cpp or grader.pas

  The sample grader executes two separate rounds. In each round, it first calls **encode** with the given data, and then it calls **decode** with the output your procedure **encode** produced. In the first round the grader does not change the order of the integers in the encoded message. In the second round the sample grader swaps the integers on odd and even positions. The real grader will apply various kinds of permutations to the encoded messages. You can change how the sample grader shuffles the data by modifying its procedure **shuffle** (in C/C++) or **Shuffle** (in Pascal).

  The sample grader also checks for both range and length of the encoded data. By default, it checks that the encoded data is in the range between **0** and **65535**, inclusive, and that the length is at most **10×N**. You can change this by adjusting the constants **CHANNEL_RANGE** (from 65535 to 255, for example) and **MAX_EXPANSION** (from 10 to 15 or 7, for example).

- Sample grader input: grader.in.1, grader.in.2, ...

  Note: The sample grader reads the input in the following format:

  - Line 1: **N**
  - Line 2: a list of **N** numbers: **M[0]**, **M[1]**, ..., **M[N-1]**

- Expected output for sample grader input: grader.expect.1, grader.expect.2, ...
  For this task, each one of these files should contain precisely the text "**Correct.**"