Zadanie 1 (10 pkt. na pracowni, później 5 pkt.). Napisz program, który wczyta podane przez użytkownika liczbę naturalną *n*, a następnie *n* liczb naturalnych *a_0*, *a_1*, … ,*a_(n-1)*. Program powinien czytelnie informować o tym, co użytkownik ma wprowadzić; jeśli wprowadzone zostaną błędne dane, należy wypisać czytelny komunikat i zakończyć działanie.

Po wczytaniu poprawnych liczb program powinien wydrukować prostokąt znaków (kropek i kratek) o *n* kolumnach i *max {a_i}* wierszach tak, że w *i*-tej kolumnie znajduje się *a_i* kratek położonych możliwie blisko środka kolumny (a pozostałe znaki to kropki). Na przykład kiedy wprowadzone zostaną wartości

```
4
1 5 3 2
```

możliwe odpowiedzi to

```
.*..
.***
****
.**.
.*..
```

oraz (zwróć uwagę na różnicę w ostatniej kolumnie)

```
.*..
.**.
****
.***
.*..
```

Program powinien, przynajmniej co do zasady, działać poprawnie dla dowolnych *n*, w szczególności nie zakładaj żadnego "ręcznego" górnego ograniczenia tej wartości (podobnie dla *a_i*). Nie przejmuj się tym, że dla pewnych wartości efekt będzie "wyglądał dziwnie" (bo np. *n* będzie większe niż aktualna szerokość okna terminala).

Zadanie 2 (10 pkt.). Szyfrem Cezara nazywamy kodowanie, w którym litery tekstu jawnego zamienia się na oddalone od nich o stałą liczbę pozycji alfabetu – nazwijmy ją *k*. Na przykład tekst "Bardzo Tajny Szyfr" dla *k*=3 to "Edugcr Wdmqb Vcbiu". Szyfr ten ignoruje znaki spoza alfabetu (w tym też np. obecne w przykładzie spacje) oraz wielkość liter.

Niedokładnym szyfrem Cezara nazwiemy taką wariację szyfru Cezara, w której również niektóre litery alfabetu są ignorowane. Na przykład: niedokładny szyfr Cezara ignorujący litery "a", "b" oraz "d", także dla *k*=3 z hasła "Bardzo Tajny Szyfr" stworzy "Baudfr Wamqe Vfeiu". Wynika to z tego, że patrzymy na oddalenie o *k* w alfabecie pozbawionym liter "a", "b", "d" zamiast w pełnym (zaś te konkretne litery same pozostają bez zmian).

Napisz program, który w pętli będzie wczytywał wiersze ze standardowego wejścia i wypisywał je **od tyłu** na standardowe wyjście po zakodowaniu niedokładnym szyfrem Cezara. Załóż, że na wejściu będą pojawiać się wiersze długości nie większej niż 80 (nie licząc znaku złamania wiersza) zawierające tylko drukowalne znaki ASCII. **Uwaga**: w wypadku wystąpienia wiersza dłuższego, program może zadziałać niezgodnie z opisem (np. nie wczytać i nie zakodować całości wiersza naraz), natomiast nie powinien wykonywać błędnych operacji (np. czytać lub pisać w nieprzydzielonej pamięci).

Wartość *k* oraz zestaw znaków ignorowanych mają znajdować się w kodzie programu w jasno zaznaczonym (komentarzami) miejscu, a ich zmiana powinna polegać na modyfikacji najwyżej trzech instrukcji programu. Uwzględnij zarówno przypadek, kiedy znaków ignorowanych nie ma w ogóle (a więc uzyskamy zwyczajny szyfr Cezara), jak i kiedy ignorowanych jest wszystkich 26 znaków alfabetu (a więc szyfr będzie identycznością, podobnie jak dla *k*=0). Możesz przyjąć jakieś rozsądne upraszczające założenia (np. że znaki ignorowane muszą być w kodzie podane jako małe, a nie wielkie litery), ale opisz je w komentarzu.

Zadanie 3 (10 pkt.).

Permutacja *n*-elementowa to *n*-elementowy ciąg, w którym każda z liczb od 1 do *n* występuje dokładnie raz. Przykładową permutacją 6-elementową jest 5, 3, 6, 2, 1, 4, gdzie liczba 5 jest na 1-szej pozycji, a liczba 4 na 6-tej. **Nałożenie permutacji** na element *a* to po prostu zamiana elementu *a* na wartość, która jest na pozycji *a*. W powyższym przykładzie, element 4 zamieni się na wartość 2, a 1 na 5.

Zauważmy, że permutację można nałożyć na element wielokrotnie – przykładowo, wartość 4 będzie się zamieniała kolejno na elementy 2, 3, 6, 4, 2, 3, 6, ... itd. Co więcej, permutację można też "nałożyć *ujemną* liczbę razy", co oznacza w pewnym sensie "cofnięcie" nałożenia permutacji. Zatem cofając nałożenie permutacji element 4 zamieniłby się na 6, a dla kolejnych "cofnięć" – na 3, 2, 4, 6, 3, 2, ... itd. "Zero-krotne" nałożenie permutacji na element nie zmienia go.

Elementy permutacji *n*-elementowej **tworzą jeden cykl** tylko wtedy, gdy dopiero *n*-krotne nałożenie permutacji na pewien element sprawi, że zamieni się on na samego siebie (ale *k*-krotne nałożenie dla każdego dodatniego $k < n$ nie ma tej własności). Dla przykładu, powyższa permutacja nie tworzy jednego cyklu, za to permutacja 4, 5, 6, 3, 1, 2 już tak.

Zadanie

Napisz program, który wczyta zadaną na wejściu permutację oraz zapytania postaci dwóch liczb *a* i *k*, a dla każdego z zapytań odpowie na jaki element zamieni się *a* po nałożeniu na niego permutacji *k* razy.

Wejście

W pierwszym wierszu wejścia podana jest jedna liczba całkowita dodatnia *n*, oznaczająca liczbę elementów w permutacji. W drugim wierszu wejścia podane jest *n* liczb całkowitych, będących permutacją liczb od 1 do *n*.

W trzecim wierszu podana jest jedna liczba całkowita *q*, oznaczająca liczbę zapytań. W kolejnych *q* wierszach podane są zapytania. Każde z nich składa się z dwóch liczb całkowitych a_i oraz k_i.

Wyjście

Dla każdego zapytania wypisz w kolejnym, *i*-tym wierszu jedną liczbę całkowitą, oznaczającą element otrzymany po k_i-krotnym nałożeniu permutacji z wejścia na element a_i.

Ograniczenia

We wszystkich testach zachodzi $1 \le a_i \le n \le 10^5$, $0 \le q \le 10^5$, oraz $-10^9 \le k_i \le 10^9$ dla wszystkich *i*. Jeśli $n > 1000$, to elementy permutacji tworzą jeden cykl.

- W pierwszych 8 testach zachodzi $n, q \le 1000$.
- W pierwszych 5 testach *k* jest nieujemne.
- W pierwszych 2 testach zachodzi $k \le 1000$.

Przykłady

Przykład A

Wejście

```
5
1 4 2 5 3
5
4 1
1 3
```

3 1
2 2
1 2

Wyjście

5
1
2
5
1

Przykład B

Wejście

6
2 5 4 6 3 1
5
5 10
6 4
5 1
6 0
1 0

Wyjście

1
3
3
6
1

Przykład C

To jedyny z przykładowych zestawów, w którym *k_i* bywają ujemne.

Wejście

10
8 3 4 7 2 6 5 9 1 10
5
5 -39
9 -98
6 73
10 16
8 92

Wyjście

2
1
6
10
1

Uwagi

Przedstawienie wejścia w osobnych wierszach ma znaczenie głównie wizualne; pisząc program nie trzeba przejmować się tym, czy pomiędzy kolejnymi liczbami jest spacja, czy złamanie wiersza – specyfikator konwersji %d w napisie formatującym scanf i tak wczyta wszystkie potrzebne białe znaki poprzedzające kolejną liczbę. Program nie musi też kontrolować poprawności wejścia (ani jego spójności, np. czy podany ciąg faktycznie jest permutacją), bo będzie uruchamiany tylko na poprawnych danych. Konstrukcja scanf("%d", &tablica[indeks]); pozwala na wczytanie wartości do wybranej komórki tablicy.