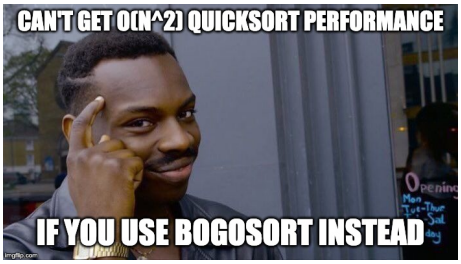
Quicksort



Quicksort ist ein einfacher Sortieralgorithmus, der in der Praxis gut funktioniert, aber aus theoretischer Sicht nicht ganz so quick ist. Damit wollen wir uns in dieser Aufgabe beschäftigen. Der Algorithmus funktioniert rekursiv: Es wird ein Pivotelement p ausgewählt, die restlichen Elemente nach kleiner und größer p partitioniert und links bzw. rechts von p angeordnet. Die kleineren Listen können dann rekursiv sortiert werden. Die maximale Tiefe der Rekursion, die dabei erreicht wird, ist ein Maß für die Laufzeit des Algorithmus. Wenn wir n Elemente sortieren und p immer gut wählen, sind die Partitionen ungefähr gleich groß, sodass man nur eine Tiefe von $\mathcal{O}(\log n)$ erreicht. Wird ungünstig partitioniert, kann die Tiefe bis zu n werden und die Laufzeit zu $\mathcal{O}(n^2)$ degenerieren. In dieser Aufgabe geht es darum Testfälle zu konstruieren, wo der Algorithmus *genau* eine bestimmte Rekursionstiefe k erreicht.

Um Quicksort in C++ zu implementieren, bietet sich eine Standard-Library-Funktion namens `std::stable_partition` an. Diese erledigt die Aufgabe, ein Array in einen linken und einen rechten Teil zu partitionieren, ohne dabei die Elemente, die auf derselben Seite landen, im Vergleich zueinander umzuordnen („stable"). Der Aufruf ist etwas kompliziert, aber die Details sind für diese Aufgabe nicht wichtig:

```
auto mid = std::stable_partition(bgn, end, [p](int x) {
    return x < p;
});
```

Hier wird der Bereich von `bgn` bis exklusive `end` so umarrangiert, dass alle Elemente, die `< p` sind links stehen, und alle anderen rechts. Nach dem Aufruf zeigt der Rückgabewert `mid` auf das erste Element, das `>= p` ist.

Wir betrachten folgende C++ Quicksort Implementierung:

```
#include <algorithm>

template<class Iterator>
int quicksort(Iterator bgn, Iterator end) {
    if (bgn == end)
        return 0;
    int p = *bgn;
    auto mid = std::stable_partition(bgn, end, [p](int x) {
        return x < p;
    });
    return 1 + std::max(quicksort(bgn, mid),
                        quicksort(mid+1, end));
}
```

Die Funktion bekommt zwei Iteratoren[1] auf ein Range von Integers und sortiert dieses. Zurückgegeben wird die maximal erreichte Rekursionstiefe. Zwei Details der Implementierung

[1] z.B. `a.begin()` und `a.end()` eines `std::vector<int> a`.

sind interessant: Erstens wird immer das erste Element als Pivot gewählt. Zweitens landet das Pivot-Element p immer an der Position `mid`, da es ja vor dem Partitionieren an der ersten Stelle steht und `stable_partition` die Reihenfolge nicht unnötig verändert.

Kannst du den Fakt, dass immer das erste Element als Pivotelement ausgewählt wird, ausnutzen und Eingaben konstruieren, die *genau* Rekursionstiefe k erzeugen? Dir sind zwei Zahlen n und k gegeben. Finde eine beliebige Permutation von $1, 2, \ldots n$, sodass `quicksort` darauf Rückgabewert k hat. Falls keine solche Permutation existiert, gib stattdessen -1 aus.

Eingabe

Eine Zeile mit n und k.

Ausgabe

Eine Permutation von $1, 2, \ldots n$ oder -1, falls es keine Lösung gibt. Wenn es mehrere Lösungen gibt, kannst du irgendeine davon ausgeben.

Beispiele

Eingabe	Ausgabe	Anmerkungen
`5 5`	`1 2 3 4 5`	Hier kommen immer alle Element in die rechte Hälfte.

Eingabe	Ausgabe	Anmerkungen
`4 3`	`2 1 3 4`	Alternative Lösungen sind z.B. 2 1 4 3 oder 3 1 2 4.

Eingabe	Ausgabe	Anmerkungen
`2 1`	`-1`	Sowohl 1 2 als auch 2 1 haben eine maximale Rekursionstiefe von 2.

Subtasks

Allgemein gilt: $1 \leq k \leq n \leq 10^5$.

Subtask 1 (15 Punkte): $n \leq 10$

Subtask 2 (10 Punkte): $n \leq 20$

Subtask 3 (10 Punkte): $n \leq 100$

Subtask 4 (15 Punkte): $n \leq 500$

Subtask 5 (10 Punkte): $n \leq 3000$

Subtask 6 (10 Punkte): $k = n$ (worst case)

Subtask 7 (10 Punkte): $k = \lceil \log_2 n \rceil$ (best case)

Subtask 8 (20 Punkte): Keine Einschränkungen

Limits

Zeitlimit: 1 s **Speicherlimit:** 256 MB