Ludwig-Maximilians-Universität München
Mathematisches Institut
Gregor Kleen

Programmieren II für Studierende der Mathematik

Blatt 5 – Lösungsvorschlag

Aufgabe 5 Eine Permutation ist eine Bijektion zwischen einer Menge (hier $[0, n) \subset \mathbb{N}$) und sich selbst. Die Abbildungsvorschrift einer Permutation auf $[0, n)$ schreiben wir als Tabelle mit zwei Zeilen und n Spalten. Wir betrachten im Folgenden die Permutation π:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 3 & 1 & 6 & 5 & 4 & 7 & 0 & 2 & 9 & 8 \end{pmatrix} \tag{1}$$

Die Schreibweise als Tabelle ist zu verstehen sodass gilt: $\pi(0) = 3, \pi(1) = 1, \pi(2) = 6, \ldots \pi(9) = 8$.

Wir bezeichnen für zwei Permutation σ, ρ auf $[0, n)$ mit $\sigma \circ \rho$ die *Komposition* der Permutationen. Die Definition ist identisch mit der Komposition von Abbildungen: für alle $k \in [0, n)$ gilt $(\sigma \circ \rho)(k) = \sigma(\rho(k))$. Wir verstehen die Komposition $\circ$ rechts-assoziativ. Wir bezeichen mit π^n die n-fache Komposition von π mit sich selbst. D.h. $\pi^0 = \mathrm{id}$, $\pi^1 = \pi$, $\pi^2 = \pi \circ \pi$, $\pi^3 = \pi \circ \pi \circ \pi$, …, $\pi^n = \pi \circ \pi^{n-1}$

Als eine *Transposition* $\tau = (k\ l)$ auf $[0, n)$ bezeichnen wir die Permutation mit $\tau(i) = i$ für alle $i \in [0, n) \setminus \{k, l\}$ und $\tau(k) = l$, $\tau(l) = k$.

Es ist ein bekanntes Resultat, dass sich jede Permutation darstellen lässt als Komposition von Transpositionen. D.h. für jede Permutation π auf $[0, n)$ existiert eine endliche Folge von Transpositionen $\tau_1, \tau_2, \ldots, \tau_m$ auf $[0, n)$ mit $\pi(k) = (\tau_1 \circ \tau_2 \circ \ldots \circ \tau_m)(k)$ für alle $k \in [0, n)$.

Hierfür zerlegen wir die Permutation π auf $[0, n)$ zunächst in *Zykel*. Ein Zykel ist eine Folge der Form $(a\ \pi(a)\ \pi^2(a)\ \ldots\ \pi^{\ell_a - 1}(a))$ für $a \in [0, n)$. $\ell_a \in \mathbb{N} \setminus \{0\}$ bezeichnet hierbei die stets wohldefinierte kleinste Zahl, sodass $\pi^{\ell_a}(a) = a$.
Um π in Zykel zu zerlegen wählen wir zunächst eine beliebige Zahl $a \in [0, n)$ und bestimmen den Zykel in dem sie enthalten ist. Falls nicht alle Zahlen aus $[0, n)$ bereits in diesem ersten Zykel erwähnt wurden, so wählen wir ein $b \in [0, n)$ das noch nicht erwähnt wurde und bestimmen den Zykel in dem b liegt. Durch Iteration des Verfahrens erhalten wir so eine Menge von disjunkten Zykeln. Zykel der Form (a), d.h. $\pi(a) = a$ können wir verwerfen.

Es gilt für eine Permutation π mit Zykeln $(a_0\ a_1\ a_2\ \ldots\ a_m), (b_0\ b_1\ \ldots\ b_l), \ldots$, dass $\pi = (a_0\ a_1) \circ (a_1\ a_2) \circ \ldots \circ (a_{m-1}\ a_m) \circ (b_0\ b_1) \circ \ldots \circ (b_{l-1}\ b_l) \circ \ldots$, also eine Komposition von Transpositionen.

Erstellen Sie zunächst eine Klasse `Transposition` zur Darstellung einer Transposition. Verwenden Sie hierfür zwei `private` Attribute `n` und `k` vom Typ `int`. Implementieren Sie einen Konstruktor für `Transposition` der zwei Parameter vom Typ `int` akzeptiert und die Attribute `n` und `k` damit initialisiert. Prüfen Sie im Körper des Konstruktors ob $n == k$ und werfen Sie, falls dem so ist, eine geeignete exception.

Transposition

```
class Transposition {
  private:
    int n, k;
```

```
  public:
    Transposition(int n_, int k_): n(n_), k(k_) {
      if (n == k) {
        ostringstream os;
        os << "Transposition(" << n_ << ", " << k_ << ")";
        throw invalid_argument(os.str());
      }
    }

    Transposition Auswertung

    Transposition Ausgabe
};
```

Überladen Sie den Ausgabeoperator für `Transposition` sodass Objekte der Klasse ausgegeben werden können in der Form $(n\ k)$.

Transposition Ausgabe

```
friend ostream& operator<<(ostream& stream, const Transposition& t) {
  return stream << "(" << t.n << " " << t.k << ")";
}
```

Erstellen Sie eine Klasse `Permutation` zur Darstellung von Permutationen durch ein `private` Attribut als Liste von Transpositionen (`list<Transposition>`). Erstellen Sie zunächst Konstruktoren für `Permutation` mit:

- keinen Parametern; es soll die Identitätsfunktion erzeugt werden
- zwei Parametern vom Typ `int`; es soll für Parameter n und k die Transposition $(n\ k)$ erzeugt werden
- einem Parameter vom Typ `Transposition`

Permutation

```
class Permutation {
  private:
    list<Transposition> perm;

  public:
    Permutation(): perm() {}
    Permutation(int n_, int k_): perm({Transposition(n_, k_)}) {}
    Permutation(const Transposition& t): perm({t}) {}
    Permutation Konstruktor

    Permutation Auswertung

    Methode getPerm

    Permutation Multiplikationsoperator

    Permutation Ausgabe
};
```

Implementieren Sie einen Konstruktor für `Permutation` mit einem Parameter vom Typ `map<int, int>`. Die

gegebene endliche Abbildung soll als Permutation aufgefasst werden. Führen Sie im Körper des Konstruktors die Zerlegung des Parameters in Zykel durch, bestimmen Sie die Darstellung der durch die endliche Abbildung dargestellten Permutation als Komposition von Transpositionen und speichern Sie diese als `list<Transposition>` im Attribut des erzeugten Objektes.

Permutation Konstruktor

```
Permutation(const map<int, int>& pi) {
  set<int> seen{};
  for (auto p: pi) {
    if (seen.find(p.first) != seen.end())
      continue;

    list<int> cycle{p.first};
    while (pi.at(cycle.back()) != p.first)
      cycle.push_back(pi.at(cycle.back()));
    seen.insert(cycle.begin(), cycle.end());

    if (cycle.size() <= 1)
      continue;

    list<int>::const_iterator curr = cycle.begin(),
      next = ++(cycle.begin());
    for (; next != cycle.end(); curr++, next++)
      perm.push_back(Transposition{*curr, *next});
  }
}
```

Überladen Sie den Ausgabeoperator für `Permutation` sodass Objekte der Klasse ausgegeben werden können als Folge von Transpositionen in der Form $(n_1\ k_1)\ (n_2\ k_2)\ \ldots$.

Permutation Ausgabe

```
friend ostream& operator<<(ostream& stream, const Permutation& p) {
  list<Transposition>::const_iterator curr = p.perm.begin(),
    next = ++(p.perm.begin());

  for (; curr != p.perm.end(); curr++, next++) {
    stream << *curr;
    if (next != p.perm.end())
      stream << " ";
  }
  return stream;
}
```

Implementieren Sie ein Hauptprogramm in dem Sie n und dann eine Permutation zunächst als Wert vom Typ `map<int, int>` für Indizes $[0, n)$ von der Standardeingabe einlesen. Übergeben Sie die resultierende endliche Abbildung dann an den Konstruktor von `Permutation` und geben Sie das resultierende Objekt auf der Standardausgabe aus. Sie sollten eine Folge von Transpositionen auf der Standardausgabe erhalten. Testen Sie ihr Programm mit der Permutation π aus 1.

Hinweis (Kontrollergebnis).

$$\pi = (0\ 3)\ (3\ 5)\ (5\ 7)\ (7\ 2)\ (2\ 6)\ (8\ 9)$$

transpositions.cpp

```cpp
#include <iostream>
#include <set>
#include <map>
#include <list>
#include <algorithm>
#include <stdexcept>
#include <sstream>

using namespace std;

Transposition

Permutation

Funktion ausgeben

int main() {
  int n;
  cout << "n: "; cin >> n;
  map<int, int> pi;
  for (int i = 0; i < n; i++) {
    cout << "pi[" << i << "]: ";
    cin >> pi[i];
  }

  Permutation p{pi};
  cout << p << endl;
  Auswertung Kontrollausgabe

  Kontrollausgabe multipliziert

  return 0;
}
```

```
n: 10
pi[0]: 3
pi[1]: 1
pi[2]: 6
pi[3]: 5
pi[4]: 4
pi[5]: 7
pi[6]: 0
pi[7]: 2
pi[8]: 9
pi[9]: 8
(0 3) (3 5) (5 7) (7 2) (2 6) (8 9)
p[0]: 3  p[1]: 1  p[2]: 6  p[3]: 5  p[4]: 4  p[5]: 7  p[6]: 0  p[7]: 2  p[8]: 9  p[9]: 8
q[0]: 3  q[1]: 1  q[2]: 6  q[3]: 5  q[4]: 4  q[5]: 7  q[6]: 0  q[7]: 2  q[8]: 9  q[9]: 8
```

Überladen Sie die Funktionsauswertungsoperatoren für die Klassen `Transposition` und `Permutation` mathe-

matisch sinnvoll.

Transposition Auswertung

```
int operator()(int i) const {
  if (i == k)
    return n;
  else if (i == n)
    return k;
  else
    return i;
}
```

Permutation Auswertung

```
int operator()(int i) const {
  for (list<Transposition>::const_reverse_iterator it = perm.rbegin();
    it != perm.rend();
    it++)
    i = (*it)(i);
  return i;
}
```

Erweitern Sie Ihr Hauptprogramm sodass dieses den Funktionsauswertungsoperator von `Permutation` verwendet um für die Werte $k \in [0, n)$ jeweils $\pi(k)$ auszugeben wobei π die von der Standardeingabe eingelesene Permutation bezeichne.

Funktion ausgeben

```
void ausgeben(ostream& stream, const string& c, const Permutation& p, int n) {
  for (int i = 0; i < n; i++) {
    cout << c << "[" << i << "]: " << p(i);
    if (i != n - 1)
      cout << "  ";
  }
  cout << endl;
}
```

Auswertung Kontrollausgabe

```
ausgeben(cout, "p", p, n);
```

Implementieren Sie eine Methode `getPerm` für `Permutation` die Zugriff auf das `private` Attribut, welches die Liste von Transpositionen enthält, als konstante Referenz ermöglicht.

Methode getPerm

```
const list<Transposition>& getPerm() const {
  return perm;
}
```

Überladen Sie den Multiplikationsoperator für `Permutation` sodass dieser die Komposition $\circ$ zweier Permutation berechnet.

Permutation Multiplikationsoperator

```
Permutation& operator*=(Permutation o) {
  perm.splice(perm.begin(), o.perm);
  return *this;
}
friend Permutation operator*(Permutation a, const Permutation& b) {
  return a *= b;
}
```

Erweitern Sie ihr Hauptprogramm sodass dieses die Methode `getPerm` verwendet um Zugriff auf die Transpositionen zu erhalten in die sich π zerlegen lässt. Berechnen Sie eine neue Permutation π' indem Sie die Transpositionen jeweils als Permutation auffassen (mit dem geeigneten Konstruktor von `Permutation`) und geeignet aufmultiplizieren. Geben Sie zum Vergleich mit π auch für π' für die Werte $k \in [0, n)$ jeweils $\pi'(k)$ aus.

Kontrollausgabe multipliziert

```
Permutation q{};
for (Transposition t: p.getPerm())
  q = Permutation{t} * q;
ausgeben(cout, "q", q, n);
```