Coding-Tricks Cheat-Sheet

Container

`std::vector` (“besseres Array”):

```
#include <vector>
vector<int> v(n); // v={0,0,...,0}, v.size()==n
v.clear();        // v={}
v.push_back(5);   // v={5}
int x = v[0];     // x=5
for (size_t i=0; i<v.size(); ++i) // manuelle Schleife
  cout << v[i] << '\n';
for (vector<int>::iterator it=v.begin(); // Iteratoren
     it!=v.end(); ++it)
  cout << *it << '\n';
for (auto it=v.begin(); it!=v.end(); ++it) // auto
  cout << *it << '\n';
for (auto& elem : v) // Range based for
  cout << elem << '\n';
```

`std::map` (Key-Value-Paare, Zugriff nach Key, immer sortiert):

```
#include <map>
std::map<std::string, int> m;
m["key"] = 5;     // m={"key": 5}
// Falls der Key nicht existiert,
// wird er mit Default-Wert eingefügt.
int x = m["new"]; // x=0, m={"key": 5, "new": 0}
for (auto& elem : v) // Range based for
  cout << elem.first << ' ' << elem.second << '\n';
```

Andere Container: `std::set` (wie `std::map`, nur ohne Value), `std::multiset` und `std::multimap` (wie `std::set`/`std::map`, aber doppelte Keys erlaubt), deque (wie `std::vector`, aber mit `push_front`), `std::queue` und `std::stack` (basieren auf deque), `std::priority_queue`.
Bei `std::map`/`std::map` ist immer das kleinste Element vorne und bei /`std::priority_queue` das grösste. Umgekehrt sortieren:

```
map<int, int, greater<int> > m;
set<int, greater<int> > s;
priority_queue<int, vector<int>, greater<int> > pq;
```

Strings

```
#include <string>
string s; // wie ein ganz normaler Container
cin >> s; // kann Eingabe (und Ausgabe)
s = "cystoflagellate"
string t = s.substr(2, 5); // ab 2, Länge 5: t=="stofl"
```

Algorithmen

```
#include <algorithm>
vector<int> v{5,4,3,2,1};
sort(v.begin(), v.end());       // Alles sortieren
sort(v.begin()+1, v.begin()+4); // nur Indices 1,2,3
// eigene Vergleichsfunktion: Ist lhs<rhs?
bool comp(int lhs, int rhs) { return lhs<rhs; }
sort(v.begin(), v.end(), comp);  // mit comp sortieren
// Eigener Kleiner-als-Operator definieren
bool operator<(const mystruct& lhs, const mystruct& rhs) {...}
```

Andere nützliche Algorithmen:

```
vector<int> v{1,2,2,5,7,7,7,8};
// v muss sortiert sein:
bool b = binary_search(v.begin(), v.end(), 5); // b==true
auto it = lower_bound(v.begin(), v.end(), 5); // zeigt auf 5
v.erase(unique(v.begin(), v.end()), v.end()); // v={1,2,5,7,8}
// wenn v nicht sortiert
auto it = find(v.begin(), v.end(), 5); // it zeigt auf 5
reverse(v.begin(), v.end()); // v umkehren
```

Terminal

```
# Navigation
$ cd directory  # Verzeichnis wechseln
$ pwd           # aktuelles Verzeichnis anzeigen
$ ls            # Dateien auflisten
$ ls *.in       # Dateien auflisten, die mit .in enden

# C++ kompilieren (Programm prog.cc)
$ g++ -Wall -Wextra -std=c++11 -g3 -ggdb3 \
  -D_GLIBCXX_DEBUG prog.cc -o prog
# -Wall -Wextra:
#    Warnungen und zusätzliche Warnungen einschalten
# -std=c++11 oder -std=c++14:
#    C++11 bzw. C++14 aktivieren.
# -g3 -ggdb3: Debug-Informationen für gdb
# -D_GLIBCXX_DEBUG:
#   Speicherzugriffe überprüfen bei
#   Containern und Iteratoren

# Testen
$ ./prog <sample01.in # Eingabe von sample01.in
# Eingabe von allen *.in
$ for f in *.in; do echo "-- $f --"; ./prog <$f; done

# gdb benutzen (mit -g kompilieren!)
$ gdb prog
(gdb) run <sample01.in
(gdb) bt  # für backtrace
(gdb) q   # für quit
```

printf-Debugging

```
// Mit Debug-Ausgaben
#define DEB(x) cerr << x << '\n'

// Ohne Debug-Ausgaben
#define DEB(x) // cerr << x << '\n'

for (int i=0; i<10; ++i) {
  DEB("i=" << i << " sum=" << sum);
  sum += i;
}
```

Oder:

```
const bool debug = false; // oder true
#define DEB(x) do { if (debug) cerr << x << '\n'; } while(0)
```

Oder lokal kompilieren mit `-DDEBUG` (auf dem Grader ist das nicht definiert)

```
#ifdef DEBUG
#  define DEB(x) cerr << x << '\n'
#else
#  define DEB(x)
#endif
```

I/O-Optimierungen

```
// Synchronisierung mit scanf/printf ausschalten
ios_base::sync_with_stdio(false);
// Automatisches flush von cout bei Einlesen von cin verhindern
cin.tie(0);
```

Strategie

Laufzeitanalyse: In 1 Sekunde sind 10^7 Schritte möglich.
Problem lösen:

1. Aufgabe genau lesen
2. Limits anschauen, Laufzeit erraten
3. Beispiele durchrechnen
4. Idee entwickeln
5. Korrektheit beweisen/Gegenbeispiele suchen
6. Grobe Implementierung überlegen
7. Programmieren

Wenn Bug im Programm:

1. Gegenbeispiel? Falls ja: printf-Debugging
2. Corner-Cases testen
3. Ist Ansatz wirklich korrekt?
4. Aufgabe nochmal durchlesen
5. Code nochmal durchlesen
6. Andere Aufgaben versuchen
7. Brute-Force-Lösung schreiben und automatisiert Tests generieren