Sortieren mit der STL

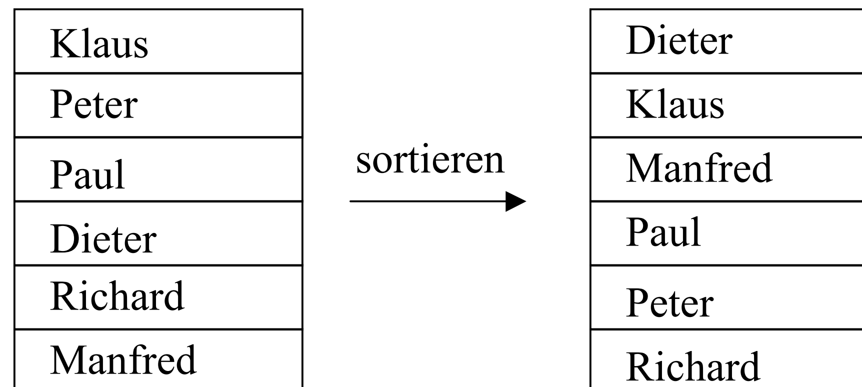
Jan Große

Überblick

- Definition und Motivation
- Sortieren mit **`sort`**
  - QuickSort
  - IntroSort
- Stabiles Sortieren mit **`stable_sort`**
- Mischen sortierter Sequenzen
- Diverse andere Sortiermethoden
- Zusammenfassung

Definition und Motivation

- Gegeben eine Sequenz $S:\{1,...,n\} \rightarrow M$ von n Objekten und Relation $<$
- Gesucht ist Sequenz $S':\{1,...,n\} \rightarrow M$ mit $S'(i) < S'(j)$ für $i < j$
- Beispiel: Liste S von 6 Namen und lexikographischer Vergleich



Definition und Motivation

- Man sortiert Ansammlungen von Objekten
  - Um Dinge mit gleichem oder ähnlichem Schlüssel zusammenzubringen (group by-Anfrage)
  - Um gleiche Dinge, die mehrfach auftreten herauszufinden (Duplikateliminierung)
  - Zur Vorbereitung und Unterstützung von Suchvorgängen (binäre Suche)
  - Um „beste“ oder „schlechteste“ Dinge zu identifizieren
- Sortiermethoden
  - InsertionSort, SelectionSort, BubbleSort
  - QuickSort, MergeSort, HeapSort
  - Verschiedenste Klassifikationen

Sortieren mit `sort`

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class StrictWeakOrdering)
void sort(RandomAccessIterator first, RandomAccessIterator last,
          StrictWeakOrdering comp);
```

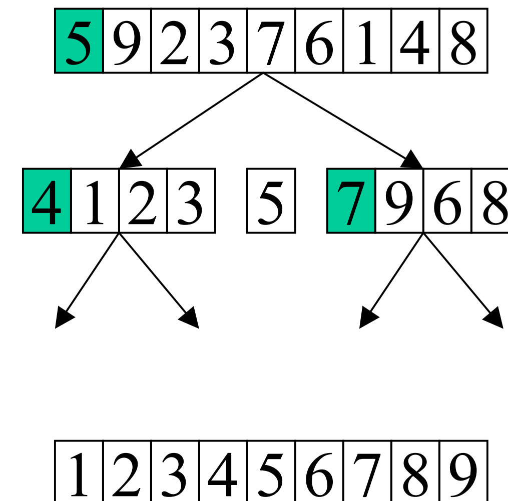
- Sortiert die Elemente in `[first, last)` in aufsteigender Ordnung bezüglich
  - `operator <`
  - `comp`
- Für zwei gültige Iteratoren `i,j` in `[first, last)` mit `i<j` gilt nicht `*j<*i`
- nicht stabil
- anwendbar auf Container mit wahlfreiem Zugriff: `vector, deque, array`
- Laufzeit O(N logN) im Durchschnitt und im worst-case ?

Sortieren mit **`sort`** (Beispiel)

```
...
#include <algorith>
#include <functional>
...

int A[] = {4, 1, 8, 2, 5, 7};
sort(A, A + 6, greater<int>());
copy(A, A + 6, ostream_iterator<int>(cout, “,“));

// Ausgabe: 8,7,5,4,2,1,
```

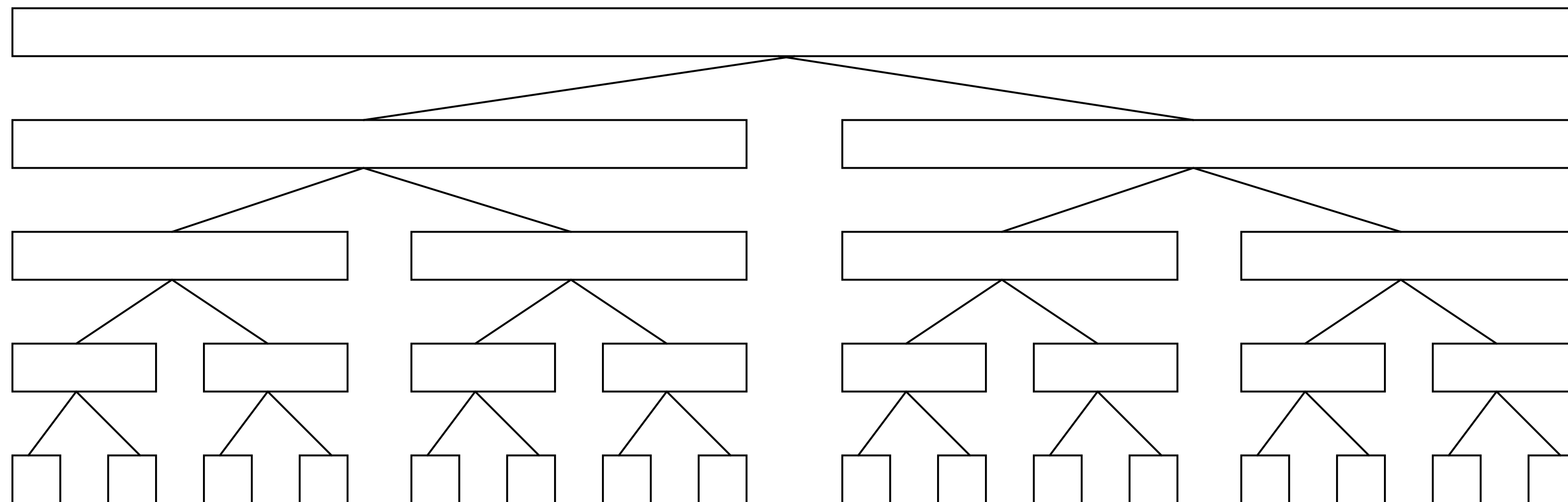
Quicksort

- Grundlage der Implementation von **`sort`**
- Idee von Quicksort:
  - wähle Pivotelement p aus Liste L
  - bilde Teillisten L1 und L2 mit
    - L1 enthält alle Elemente < p
    - L2 enthält alle Elemente > p
  - |L1|+1 ist richtige Position von p !
  - wende Quicksort auf L1 und L2 an



Quicksort (best case)

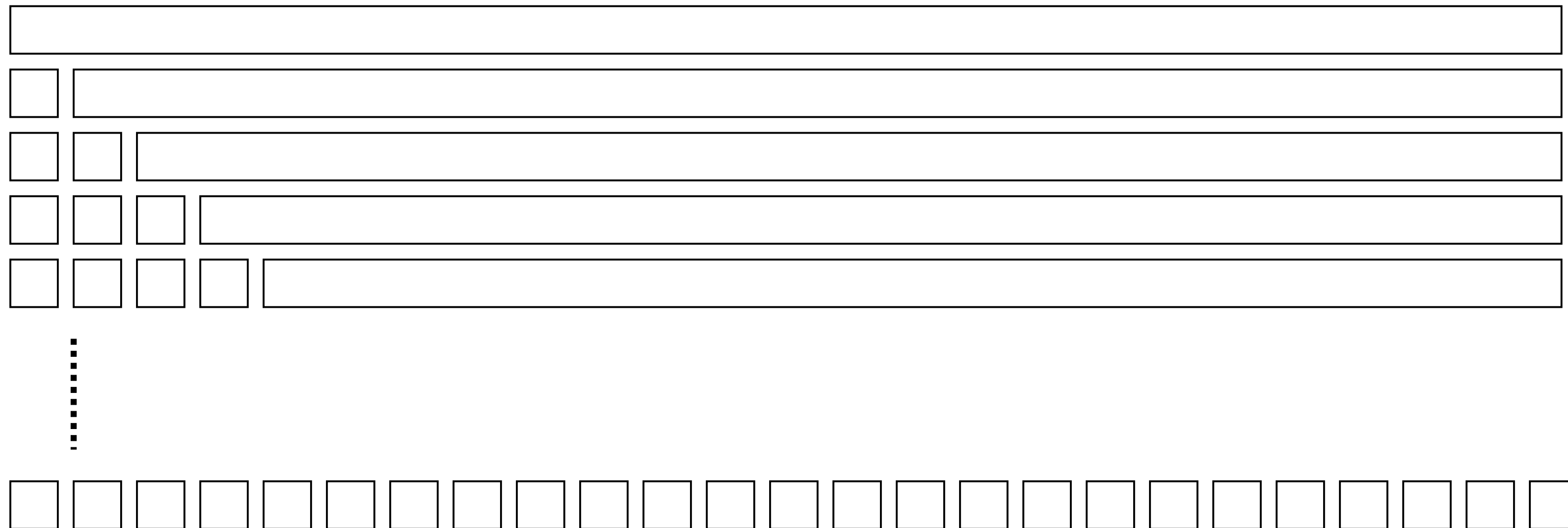
- Wahl von Pivotelement p bestimmt Laufzeit
- Am besten: p = Median der Liste L
- $x \in L$ ist Median $\Leftrightarrow |\{y<x|y \in L\}| \approx |\{y>x|y \in L\}|$



- Tiefe logN $\Rightarrow$ Laufzeit O(N logN)

Quicksort (worst case)

- wählen erstes Element in sortierter Folge



- Tiefe N $\Rightarrow$ Laufzeit O(N^2)

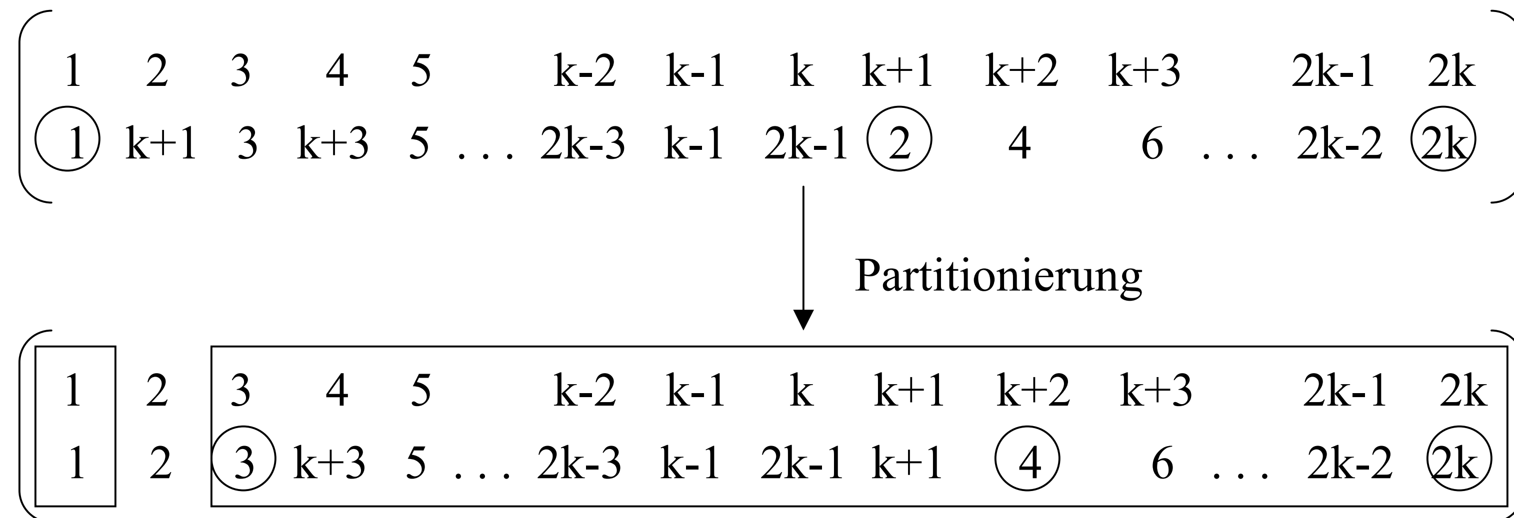
Quicksort

- Verbesserungen: kleine Partitionen mit InsertionSort
- Pivotelement: zufällig, median-of-3
- InsertionSort & median-of-3 in BCC 5.5 (Rogue Wave):

```
const int __stl_threshold = 16;

Template <class RandomAccessIterator, class T>
void __quick_sort_loop_aux(RandomAccessIterator first,
                           RandomAccessIterator last, T*)
{
    while(last - first > __stl_threshold)
    {
        RandomAccessIterator cut = __unguarded_partition
        (first, last, T(__median(*first, *(first + (last - first)/2),
                               *(last-1))));
        ...
```

Quicksort (median-of-3 killer sequence)

- median-of-3 liefert gute Partitionen auch für sortierte und fast sortierte Sequenzen
- aber quadratische Laufzeit für folgende Sequenz K_{2k} :

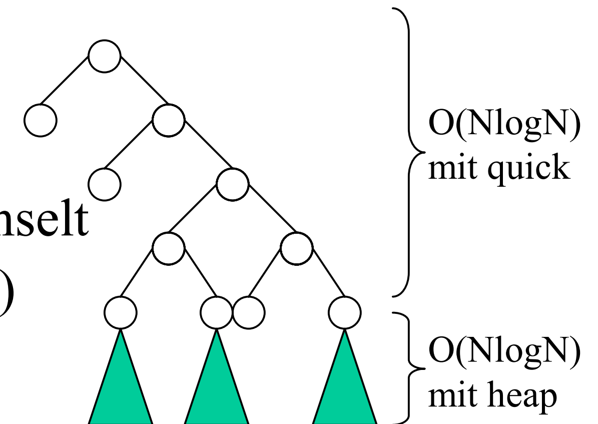


Quicksort (median-of-3 killer sequence)

- K_{2k} läßt sich k/2 mal partitionieren
- Für K_N mit N | 4 erzeugt median-of-3-quicksort mindestens Partitionstiefe N/4
- Damit Laufzeit O(N^2)
- K_N weniger wahrscheinlich als (fast) sortierte Sequenzen
- K_N wahrscheinlicher als zufällige Sequenz unter Annahme einer universellen Verteilung statt Gleichverteilung
- Universelle Verteilung: Sequenzen, die durch kurze Programme erzeugt werden, haben höhere Wahrsch.

Introsort

- als Variante von Quicksort von David Musser entwickelt um auch im worst case O(NlogN) zu gewährleisten
- warum nicht gleich Heapsort? Im Durchschnitt 2 bis 5 mal langsamer als Quicksort
- Idee:
  - Algorithmus merkt sich Rekursionstiefe
  - ab bestimmtem Limit wird zu Heapsort gewechselt
  - das Limit ist in O(logN) (empirisch gut: 2logN)



- Algorithmus beobachtet sich selbst: introspective sort
- In SGI-Implementation: ->

Introsort (Implementation)

```
template <class _RandomAccessIter>
inline void sort(_RandomAccessIter __first, _RandomAccessIter __last)
{
   if(__first != __last) {
        __introsort_loop(__first, __last, __VALUE_TYPE(__first),
                         __lg(__last - __first) * 2);
   ...
}
```

```
template <class _RandomAccessIter, class _Tp, class _Size>
void __introsort_loop(_RandomAccessIter __first,
                      _RandomAccessIter __last, _Tp*,
                      _Size __depth_limit)
{
  while (__last - __first > __stl_threshold) {
    if (__depth_limit == 0) {
      partial_sort(__first, __last, __last);
      return;
    }
    --__depth_limit; ...
```

Hier verbirgt heapsort

Performancevergleich
(median-of-3 KillerSequenzen)

Größe in 1000	Algorithmus	Vergleiche	:=	Iterator Operationen	Distanz Operationen	Operationen total
1	Introsort Quicksort Heapsort	28.8 199.1 10.4	17.1 6.4 15.6	175.6 421.3 136.9	153.6 3.6 159.4	375.1 630.4 322.3
16	Introsort Quicksort Heapsort	662.0 48334.2 231.1	353.1 132.5 321.8	3918.2 97103.0 2966.6	3446.0 57.3 3454.1	8379.3 145627.1 6973.6
64	Introsort Quicksort Heapsort	3035.4 769724.0 1052.6	1574.7 609.4 1447.3	17748.3 1541328.4 13403.2	15602.5 229.4 15610.8	37961.0 2311891.2 31513.8

Performancevergleich
(zufällige Sequenzen)

Größe in 1000	Algorithmus	Vergleiche	:=	Iterator Operationen	Distanz Operationen	Operationen total
1	Introsort Quicksort Heapsort	11.9 11.9 10.3	9.4 9.4 15.5	52.9 53.3 136.1	1.2 1.2 159.1	75.4 75.7 320.9
64	Introsort Quicksort Heapsort	1235.1 1235.1 1044.7	934.6 934.6 1435.6	5125.7 5152.3 13316.9	73.6 73.5 69419.7	7369.0 7395.5 31359.7
1024	Introsort Quicksort Heapsort	24945.6 24945.6 20812.4	17805.8 17805.8 27065.6	100946.7 101374.4 262222.8	1177.1 1176.4 306349.8	144875.1 145302.2 616450.6

Stabiles Sortieren mit `stable_sort`

```
template <class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class StrictWeakOrdering>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                 StrictWeakOrdering comp);
```

- stabil: für zwei Elemente **`x,y`** aus **`[first,last)`** mit **`x`** ist Vorgänger von **`y`** und beide sind äquivalent (es gilt nicht: **`x<y`** oder **`y<x`**) ist nach Aufruf von **`stable_sort`** weiterhin **`x`** der Vorgänger von **`y`**

Stabiles Sortieren mit `stable_sort`

- Anwendungsfall Telefonbuch: Personen sollen nach Nachnamen und dann nach Vornamen sortiert werden

```
struct Person {
    string vname;
    string nname;

    Person() : vname(""), nname("") { }
    Person(string v, string n) : vname(v), nname(n) { }
};
```

Datenstruktur für eine Person

```
void printPerson(const Person& p) {
    cout << p.nname << ", " << p.vname << endl;
}
```

Ausgabe der Personendaten

```
bool compVName(const Person& p1, const Person& p2) {
    return (p1.vname < p2.vname);
}

bool compNName(const Person& p1, const Person& p2) {
    return (p1.nname < p2.nname);
}
```

Funktionale Objekte für Namensvergleich

Stabiles Sortieren mit `stable_sort`

```
int main()
{
    vector<Person> v(8);

    v[0] = Person("Herbert", "Meier");
    v[1] = Person("Bernard", "Meier");
    v[2] = Person("Frank", "Mueller");
    v[3] = Person("Bernd", "Schmidt");
    v[4] = Person("Peter", "Mueller");
    v[5] = Person("Gerhard", "Meier");
    v[6] = Person("Werner", "Krause");
    v[7] = Person("Manfred", "Becker");

    sort(v.begin(), v.end(), compVName);
    stable_sort(v.begin(), v.end(), compNName);

    for_each(v.begin(), v.end(), printPerson);
}
```

```
Ausgabe:

Becker, Manfred
Krause, Werner
Meier, Bernard
Meier, Gerhard
Meier, Herbert
Mueller, Frank
Mueller, Peter
Schmidt, Bernd
```

Stabiles Sortieren mit `stable_sort`

- Realisierung durch Mergesort (SGI, Rogue Wave)
- Laufzeiten: best-case O(NlogN), worst-case O(N(logN)^2)
- Adaptiver Algorithmus: versucht temporären Puffer anzulegen mit dessen Hilfe Ordnung äquivalenter Elemente erhalten wird
- Laufzeit abhängig vom verfügbaren Speicher:

```
...
_Temporary_buffer<_RandomAccessIter, _Tp> buf(__first, __last);
 if (buf.begin() == 0)
   __inplace_stable_sort(__first, __last, __comp);
 else
   __stable_sort_adaptive(__first, __last, buf.begin(),
                          _Distance(buf.size()),
                          __comp);
...
```

Mischen mit `merge`

- Verschmelzen zweier sortierter Sequenzen zu einer

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator merge(InputIterator1 first1, InputIterator1, last1,
                     InputIterator2 first2, InputIterator2, last2,
                     OutputIterator result);
```

- Kopiert Elemente von `[first1, last1)` und `[first2, last2)` nach `[result, result+(last1-first1)+(last2-first2))`
- Voraussetzungen:
  - Eingabeintervalle sind gültig und sortiert
  - Ausgabeintervall ist gültig und überlappt Eingabeintervalle nicht
- Komplexität: O(N) Vergleiche

Mischen mit **merge**

```
#include <iostream>
#include <algorith>
#include <vector>
#include <list>

using namespace std;

int main()
{
    vector<int> V(5);
    int A[] = {1, 3, 5, 7, 9};
    list<int> L(10);

    V[0] = 0;
    V[1] = 2;
    V[2] = 4;
    V[3] = 6;
    V[4] = 8;

    merge(V.begin(), V.end(), A, A + 5, L.begin());

    copy(L.begin(), L.end(), ostream_iterator<int>(cout, ", "));
    // Ausgabe: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
}
```

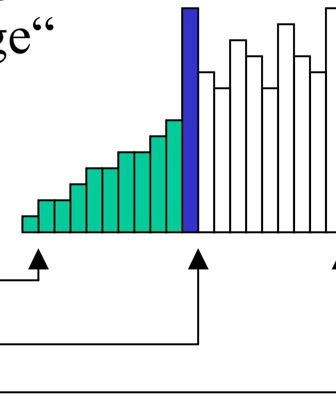
Mischen mit `inplace_merge`

```
template <class BidirectionalIterator>
inline void inplace_merge(BidirectionalIterator first,
                          BidirectionalIterator middle,
                          BidirectionalIterator last);
```

- Verschmilzt zwei aufeinanderfolgende sortierte Bereiche `[first, middle)` und `[middle, last)` zu einem sortierten Bereich `[first, last)`
- adaptives Mischen mit Laufzeit abhängig von Verfügbarkeit eines Zwischenspeichers:
  - Best-case: O(N), wenn genügend großer Puffer verfügbar (kopiert ersten Teil in `buf` und ruft `merge(buf.begin(),buf.end(),middle, last, first)`
  - Middle-case: Splitten in Teillisten bis eine in den Puffer paßt
  - Worst-case: O(NlogN), wenn kein Puffer verfügbar (rekursiv: in zwei Teillisten splitten)

Sortieren mit `partial_sort`

- manchmal keine vollständige Sortierung notwendig: „ermittle die 10 billigsten Waschmaschinen in sortierter Reihenfolge“
- lediglich partielle Sortierung



```
template <class RandomAccessIterator>
void partial_sort(RandomAccessIterator first,
                  RandomAccessIterator middle,
                  RandomAccessIterator last);
```

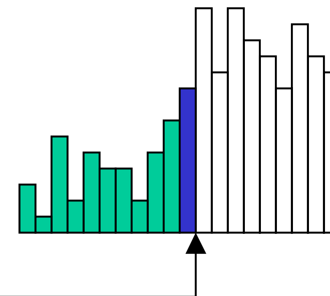
- rearrangiert den Bereich `[first,last)` derart, daß die kleinsten `middle-last` Elemente in aufsteigender Ordnung in `[first,middle)` platziert werden
- `[middle,last)` hat danach keine spezifische Ordnung
- Laufzeit: O(NlogM) durch Heapsort, nicht stabil

Sortieren mit `partial_sort`

```
struct Waschmaschine {
    int preis;
    int verbrauch;
    ...
};

bool compByPreis(const Waschmaschine& w1, const Waschmaschine& w2) {
    return (w1.preis < w2.preis);
}

bool compByVerbrauch(...);

vector<Waschmaschine> wmaschinen(100);

partial_sort(wmaschinen.begin(),
             wmaschinen.begin()+10,
             wmaschinen.end(),
             compByPreis);
```

Sortieren mit `nth_element`

- Weitere Abschwächung der Anfrage: „ermittle die 10 günstigsten Waschmaschinen“



```
template <class RandomAccessIterator>
void nth_element(RandomAccessIterator first,
                 RandomAccessIterator nth,
                 RandomAccessIterator last);
```

- Ähnlich wie **`partial_sort`** enthält **`[first, nth)`** die **`nth-first`** kleinsten Elemente aber in <u>beliebiger</u> Reihenfolge
- Iterator **`nth`** zeigt auf Element, welches auch in sortierter Sequenz dort stehen würde
- Komplexität: O(N) im Durchschnitt

Sortieren mit `nth_element`

- Ermittlung des Medians (Waschmaschine mittleren Preises):

```
median = wmaschinen.begin() + wmaschinen.size() / 2;
nth_element(wmaschinen.begin(),
            median,
            wmaschinen.end(),
            compByPreis);
```

- Ermittlung eines Elements mit prozentualem am Sortierattribut (Waschmaschine mit 20% höherem Verbrauch als Waschmaschine mit bestem Verbrauch)

```
offset = 0.2 * wmaschinen.size();
nth_element(wmaschinen.begin(),
            wmaschinen.begin() + offset,
            wmaschinen.end(),
            compByVerbrauch);
```

Diverses

- Überprüfen, ob Intervall bereits sortiert:

```
template <class ForwardIterator>
bool is_sorted(ForwardIterator first, ForwardIterator last);
```

- Partitionieren:

```
template <class ForwardIterator, class Predicate>
ForwardIterator partition(ForwardIterator first,
                          ForwardIterator last,
                          Predicate pred);
```

- Alle Elemente, die **`pred`** erfüllen, stehen vor denen, die **`pred`** nicht erfüllen (**`stable_partition`**)

Performancevergleich

- Matthew Austern: Microsoft VC++ 7.0, P3 500:

700000 double

Methode	sec
Sort	0.971
Stable_sort	1.402
Heap_sort	1.282
List_sort	1.993
Set_Sort	3.194

42731 Zeilen

Methode	sec
Sort	0.431
Stable_sort	1.322
Heap_sort	0.751
List_sort	0.25
Set_Sort	0.43

- Sequenzen von großen Objekten: sortiere nicht Objekte, sondern Zeiger auf diese
- Indirekter Vergleich: `*i<*j`

Zusammenfassung

- Idee hinter **`sort`** und damit verbundenen Laufzeiten
- „Know your sorting options“
  - **`partition`**
  - **`nth_element`**
  - **`partial_sort`**
  - **`sort`**
  - **`stable_sort`**

  ↓ Laufzeit, Sortiertheit

- Behandlung von Sequenzen komplexer Objekte