tion in der Lage sein, aus einem Bereich solcher Buchwerte nur die Titel auszuwählen. Das folgende Beispiel veranschaulicht die Verwendung dieser Funktion:

```
struct book
{
   int id;
   std::string title;
   std::string author;
};

std::vector<book> books{
   {101, "The C++ Programming Language", "Bjarne Stroustrup"},
   {203, "Effective Modern C++", "Scott Meyers"},
   {404, "The Modern C++ Programming Cookbook", "Marius Bancila"}};

auto titles = select(books, [](book const & b) {return b.title; });
```

57. Sortieralgorithmus

Schreiben Sie eine Funktion, die ein Paar von Iteratoren für den wahlfreien Zugriff entgegennimmt, um ihre obere und untere Grenze zu bestimmen, und die Elemente in dem Bereich mit dem Quicksort-Algorithmus sortiert. Stellen Sie zwei überladene Versionen der Sortierfunktion bereit, von denen die eine den Operator < heranzieht, um die Elemente in dem Bereich zu vergleichen und in eine aufsteigende Reihenfolge zu bringen, und die andere für den Vergleich eine benutzerdefinierte binäre Vergleichsfunktion verwendet.

(...)

(...)

57. Sortieralgorithmus

Quicksort ist ein Sortieralgorithmus für Elemente eines Arrays, für die eine totale Ordnung definiert ist. Bei guter Implementierung ist er erheblich schneller als *Mergesort* und *Heapsort*.

Schlimmstenfalls führt der Algorithmus $O(n^2)$ Vergleiche durch (wenn der Bereich bereits sortiert ist), doch im Durchschnitt beträgt die Komplexität nur $O(n \times log(n))$. Quicksort geht nach dem Prinzip »teile und herrsche« vor: Er partitioniert einen großen Bereich in kleinere und sortiert diese rekursiv. Dafür gibt es unterschiedliche Partitionierungsverfahren. In der hier gezeigten Implementierung verwenden wir das ursprüngliche, von *Tony Hoare* entwickelte Verfahren. In Pseudocode sieht der Algorithmus bei der Verwendung dieses Verfahrens wie folgt aus:

```
algorithm quicksort(A, lo, hi) is
   if lo < hi then
      p := partition(A, lo, hi)
      quicksort(A, lo, p)
      quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
   pivot := A[lo]
   i := lo - 1
   j := hi + 1
```

```
loop forever
   do
      i := i + 1
   while A[i] < pivot

   do
      j := j - 1
   while A[j] > pivot

   if i >= j then
      return j

   swap A[i] with A[j]
```

Eine Allzweckimplementierung dieses Algorithmus sollte Iteratoren statt Arrays oder Indizes verwenden. In der folgenden Implementierung müssen die Iteratoren einen wahlfreien Zugriff gewähren (damit sie in konstanter Zeit zu einem beliebigen Element verschoben werden können):

```
template <class RandomIt>
RandomIt partition(RandomIt first, RandomIt last)
{
   auto pivot = *first;
   auto i = first + 1;
   auto j = last - 1;
   while (i <= j)
   {
      while (i <= j && *i <= pivot) i++;
      while (i <= j && *j > pivot) j--;
      if (i < j) std::iter_swap(i, j);
   }

   std::iter_swap(i - 1, first);

   return i - 1;
}

template <class RandomIt>
void quicksort(RandomIt first, RandomIt last)
{
   if (first < last)
   {
      auto p = partition(first, last);
      quicksort(first, p);
      quicksort(p + 1, last);
   }
}
```

Mit der im Folgenden gezeigten Funktion `quicksort()` können Sie verschiedene Arten von Containern sortieren:

```
int main()
{
   std::vector<int> v{ 1,5,3,8,6,2,9,7,4 };
   quicksort(std::begin(v), std::end(v));

   std::array<int, 9> a{ 1,2,3,4,5,6,7,8,9 };
   quicksort(std::begin(a), std::end(a));

   int a[]{ 9,8,7,6,5,4,3,2,1 };
   quicksort(std::begin(a), std::end(a));
}
```

In der Aufgabe war gefordert, dass der Benutzer eine eigene Vergleichsfunktion für den Sortieralgorithmus angeben kann. Die einzige Änderung tritt dabei in der Partitionierungsfunktion auf, wo wir zum Vergleich des aktuellen mit dem Pivotelement statt der Operatoren < und > jetzt die benutzerdefinierte Funktion verwenden:

```
template <class RandomIt, class Compare>
RandomIt partitionc(RandomIt first, RandomIt last, Compare comp)
{
   auto pivot = *first;
   auto i = first + 1;
   auto j = last - 1;
   while (i <= j)
   {
      while (i <= j && comp(*i, pivot)) i++;
      while (i <= j && !comp(*j, pivot)) j--;
      if (i < j) std::iter_swap(i, j);
   }

   std::iter_swap(i - 1, first);

   return i - 1;
}

template <class RandomIt, class Compare>
void quicksort(RandomIt first, RandomIt last, Compare comp)
{
   if (first < last)
   {
      auto p = partitionc(first, last, comp);
      quicksort(first, p, comp);
      quicksort(p + 1, last, comp);
   }
}
```

Mit dieser überladenen Funktion können wir einen Bereich in absteigender Reihenfolge sortieren, wie das folgende Beispiel zeigt:

```
int main()
{
   std::vector<int> v{ 1,5,3,8,6,2,9,7,4 };
   quicksort(std::begin(v), std::end(v), std::greater<>());
}
```

Es ist auch möglich, eine iterative Version des Quicksort-Algorithmus zu implementieren. Die Leistung ist in den meisten Fällen ebenso wie die der rekursiven Variante $O(n \times log(n))$, kann sich aber im schlimmsten Fall, wenn der Bereich bereits sortiert ist, zu $O(n^2)$ verschlechtern. Die Umwandlung der rekursiven in die iterative Version ist leicht durchzuführen. Wir verwenden dazu einen Stack, um die rekursiven Aufrufe zu emulieren und die Grenzen der Partitionen zu speichern. Der folgende Code zeigt die iterative Implementierung der Version, die den Operator < für den Vergleich von Elementen verwendet:

```
template <class RandomIt>
void quicksorti(RandomIt first, RandomIt last)
{
   std::stack<std::pair<RandomIt, RandomIt>> st;
   st.push(std::make_pair(first, last));
   while (!st.empty())
   {
      auto iters = st.top();
      st.pop();

      if (iters.second - iters.first < 2) continue;

      auto p = partition(iters.first, iters.second);

      st.push(std::make_pair(iters.first, p));
      st.push(std::make_pair(p+1, iters.second));
   }
}
```

Die iterative Implementierung kann genauso eingesetzt werden wie die rekursive Version:

```
int main()
{
   std::vector<int> v{ 1,5,3,8,6,2,9,7,4 };
   quicksorti(std::begin(v), std::end(v));
}
```

58. Kürzester Pfad zwischen zwei Knoten

Um die Aufgabe zu lösen, müssen Sie den Dijkstra-Algorithmus verwenden, der den kürzesten Pfad in einem Graphen findet. Beim ursprünglichen Algorithmus geht es um den kürzesten Pfad zwischen zwei Knoten, doch hier muss der jeweils kürzeste Pfad zwischen einem gegebenen und allen anderen Knoten in dem Graphen gesucht werden.