Labb 4 Algoritmer

Uppgift: Använd STL (Standard Template Library) för att implementera några algoritmer genom att använda de färdiga algoritmerna och containerna i STL.
Bara betyg G.

Tips:

I flera av uppgifterna nedan behövs testdata som är oordnad. Det finns två sätt att framställa dessa:

- Man kan generera slumpvisa data - lätt när det gäller heltal, svårare när vi ska sortera Personer.
- Man kan ta en lista med personer och sedan ”blanda” dem – detta går att göra med alla objekt som kan sorteras. Det finns en standard funktion för detta: random_shuffle

Några användbara std funktioner och i vilka bibliotek de finns:

Random numbers: C++ bibliotek är krångligt, det gamla C varianten enkel:

```
#include <time.h>
#include <stdlib.h>

    srand((unsigned)time(NULL));
    int r = rand();
```

och nu är r ett random positivt heltal (0 <= r <= RAND_MAX == 32767)
rand() ger en förutsägbar serie av randomtal om man vet vad seed är (srand sätter seed värdet). Det gör att det ibland för testning kan vara bra att skriva t.ex. ”`srand(17);`” där 17 bara är ett exempel. Nu kommer vi vid varje testkörning få exakt samma randomtal vilket förenklar vid felsökning.

Fylla en container med något:

```
    iota(v.begin(), v.end(), 101);    // fyller v med 101, 102, 103 etc.
```

Blanda v

```
    random_shuffle(v.begin(), v.end());
```

Lambdafunktioner är anonyma funktioner t.ex.:

```
     [](int i) {return i % 10 == 7; };
```

ger true för alla tal som har 7 som sista siffra.

I <vector> finns vector klassen som bl.a. har metoderna:

- push_back
- erase

I <algorithm>

- sort
- stable_sort
- remove

Att skriva std::begin(v) och std::end(v) är mer generellt än att skriva v.begin(). T.ex. så klarar detta skrivasätt en c-array deklarerad som int arr[10];

Uppgifter:

Alla uppgifterna nedan ska göra enligt mallen:

1. Skapar en ”container” i slumpvis ordning
2. Skriver ut containern.
3. Förändra den på något sätt
4. Skriver ut containern.

Vad som skiljer är vilken ”container” och hur den förändras

Uppgift 1: Sortering

Uppgift 1a: Sortera en std::vector<int> med hjälp av std::sort

Uppgift 1b: Sortera int[] med std::sort

Uppgift 1c: Sortera en std::vector<int> med hjälp av std::sort
men sortera den i sjunkande ordning genom att använda rbegin och rend

Uppgift 1d: Sortera en int[] med hjälp av std::sort
men sortera den i sjunkande ordning genom att använda ett lambda utryck som jämförelse operator och tredje argument till std::sort.

Uppgift 2: Att förändra innehållet i en container.

Ett problem med att algoritm-biblioteket använder sig av iteratorer och inte själva containers är att en iterator inte kan ta bort element i en container. Gör ett program som tar bort alla jämna tal ur en container. Använd std::remove_if för att flytta de jämna talen sist och erase för att ta bort dem. Villkoret till remove_if skrivs enklast som en lambda funktion.

Uppgift 3: Sortering av forward_list

Alla iteratorer kan inte göra allt. T.ex. så kan man i en enkellänkad lista bara gå åt ett håll, t.ex. std::forward_list har bara länkar framåt och kan inte heller ta flera steg utan kan bara gå ett steg i taget. Dvs. operationerna --, - och + saknas och då fungerar inte den vanliga std::sort funktionen. I denna uppgift ska du skriva en ForwardSort i samma stil som STL.

```
template <class ForwardIterator>
void ForwardSort(ForwardIterator begin, ForwardIterator end);
```

Den ska bara använda forward iterator funktionerna (dvs. man kan göra *it, ++it, it1!=it2 och inte så mycket mera).
Provkör den med en forward_list.

Kommentar

Självklart ska ni inte ha minnesläckor i denna labb (men det bör inte kunna uppstå några).
Ni kommer att återanvända delar av denna labb i nästa labb, då kommer ni att sortera mm. med hjälp av itteratorer som ni gjort till den dubbellänkade listan och String.