Universität des Saarlandes
FR 6.2 Informatik
Prof. Dr. Hans-Peter Lenhof
Dipl. Inform. Andreas Hildebrandt

Programmierung II, SS 2003
Musterlösung zur 2. Übung

Hinweis: Wie in der Vorlesung angekündigt, gehen die ersten drei Übungen nicht in die Wertung ein!

Zum Kompilieren der Musterlösungen muss sich eine Datei mit Namen **Makefile** im selben Verzeichnis wie der Quellcode befinden, die folgenden Inhalt hat:

```
aufgabe1:
        g++ zufall.cpp -o zufall -g -Wall -pedantic
aufgabe2:
        g++ insertionSort.cpp -o insertionSort -g -Wall -pedantic
aufgabe3:
        g++ mergeSort.cpp -o mergeSort -g -Wall -pedantic
aufgabe4:
        g++ laufzeit.cpp -o laufzeit -g -Wall -pedantic
```

Mit folgenden Aufrufen kann man damit die Musterlösungen kompilieren.

- `make aufgabe1` Musterlösung zur Aufgabe 1 wird kompiliert.
- `make aufgabe2` Musterlösung zur Aufgabe 2 wird kompiliert.
- `make aufgabe3` Musterlösung zur Aufgabe 3 wird kompiliert.
- `make aufgabe4` Musterlösung zur Aufgabe 4 wird kompiliert.

Aufgabe 1: So ein Zufall
Der folgende Code befindet sich in der Datei **zufall.cpp**

```
#include <vector>
#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
  // Ueberpruefen der Kommandozeilenargumente
  if (argc < 3)
  {
    cout << "Aufruf mittels " << argv[0] << " Länge Startwert" << endl;
    return 1;
  }


  // Kovertieren der Argumente nach integer
  int laenge = atoi(argv[1]);
  int seed   = atoi(argv[2]);

  // Erzeugen des vectors in der richtigen Laenge
  vector<int> zahlen(laenge);

  // Initialisieren des Zufallszahlengenerators
  srand(seed);

  // Fuellen des vectors
  for (int i=0; i<laenge; i++)
  {
    zahlen[i] = rand();
  }

  // Ausgeben des vectors
  for (int i=0; i<laenge; i++)
  {
    cout << i << " " << zahlen[i] << endl;
  }

  return 0;
}
```

Aufgabe 2: InsertionSort
Der folgende Code befindet sich in der Datei **insertionSort.cpp**

```
#include <iostream>
#include <vector>

using namespace std;

// Hier ist besonders wichtig, dass wir eine
// Referenz auf den vector verwenden, da wir
// ansonsten nur eine Kopie von daten sortieren
// wuerden. Der urspruengliche vector bliebe
// unveraendert
void insertionSort(vector<int>& daten)
{
  int naechster;

  // Wir muessen einmal ueber den Vektor iterieren,
  // wobei wir das erste Element ueberspringen koennen
  // (ein einelementiger Vektor ist schon sortiert)
  for (int j=1; j<(int) daten.size(); j++)
  {
    // betrachte das naechste noch nicht einsortierte
    // Element
    naechster = daten[j];

    // und sortiere es an der passenden Stelle ein
    int i = j-1;

    while ((i >= 0) && (daten[i] > naechster))
    {
      // Wir muessen alle Zahlen die groesser sind
      // als naechster um eine Position nach rechts
      // verschieben.
      daten[i+1] = daten[i];

      --i;
    }

    daten[i+1] = naechster;
  }
}

int main(int argc, char** argv)
{
  // Ueberpruefen der Kommandozeilenargumente
  if (argc < 3)
  {
    cout << "Aufruf mittels " << argv[0] << " Länge Startwert" << endl;
```

```
    return 1;
  }


  // Kovertieren der Argumente nach integer
  int laenge = atoi(argv[1]);
  int seed   = atoi(argv[2]);

  // Erzeugen des vectors in der richtigen Laenge
  vector<int> zahlen(laenge);

  // Initialisieren des Zufallszahlengenerators
  srand(seed);

  // Fuellen des vectors
  for (int i=0; i<laenge; i++)
  {
    zahlen[i] = rand();
  }

  // sortieren des vectors
  insertionSort(zahlen);

  // Ausgeben des vectors
  for (int i=0; i<laenge; i++)
  {
    cout << i << " " << zahlen[i] << endl;
  }

  return 0;
}
```

Aufgabe 3: Mergesort
Der folgende Code befindet sich in der Datei **mergeSort.cpp**

```
#include <iostream>
#include <vector>

using namespace std;

// merge fuegt die beiden benachbarten sortierten Teilarrays
// v(start1,...,end1) und v(end1+1,...,end2) zu einem sortierten
// Array zusammen, und kopiert dieses dann in v zurueck
void merge(vector<int>& v, unsigned int start1, unsigned int end1,
 unsigned int end2)
{
```

```
  unsigned int index1=start1;
  unsigned int index2=end1+1;

  vector<int> sortiert(end2-start1+1);

  for (int index3=0; index3<(int)sortiert.size(); index3++)
  {
    if ( index1 == end1+1 )
    {
      sortiert[index3] = v[index2];
      index2++;
      continue;
    }
    if ( index2 == end2+1 )
    {
      sortiert[index3] = v[index1];
      index1++;
      continue;
    }
    if ( v[index1] > v[index2] )
    {
      sortiert[index3] = v[index2];
      index2++;
    }
    else
    {
      sortiert[index3] = v[index1];
      index1++;
    }
  }

  // kopiert den neuen vektor in den alten vektor
  for (int i=0; i<(int)sortiert.size(); i++)
  {
    v[i+start1] = sortiert[i];
  }
}

void mergeSort(vector<int>& daten, unsigned int p, unsigned int r)
{
  // Sind wir vielleicht schon fertig? Dann hat unser zu betrachtendes
  // Array die Laenge 0
  if (p < r)
  { // nein. schade.
    int q = (p+r)/2; // halbiere den vector

    // und sortiere rekursiv
    mergeSort(daten, p, q);
```

```
    mergeSort(daten, q+1, r);

    // fehlt nur noch das Kombinieren
    merge(daten, p, q, r);
  }
}

int main(int argc, char** argv)
{
  // Ueberpruefen der Kommandozeilenargumente
  if (argc < 3)
  {
    cout << "Aufruf mittels " << argv[0] << " Länge Startwert" << endl;
    return 1;
  }


  // Kovertieren der Argumente nach integer
  int laenge = atoi(argv[1]);
  int seed   = atoi(argv[2]);

  // Erzeugen des vectors in der richtigen Laenge
  vector<int> zahlen(laenge);

  // Initialisieren des Zufallszahlengenerators
  srand(seed);

  // Fuellen des vectors
  for (int i=0; i<laenge; i++)
  {
    zahlen[i] = rand();
  }

  // sortieren des vectors
  mergeSort(zahlen, 0, laenge-1);

  // Ausgeben des vectors
  for (int i=0; i<laenge; i++)
  {
    cout << i << " " << zahlen[i] << endl;
  }

  return 0;
}
```

Aufgabe 4: Laufzeitvergleich
Der folgende Code befindet sich in der Datei **laufzeit.cpp**

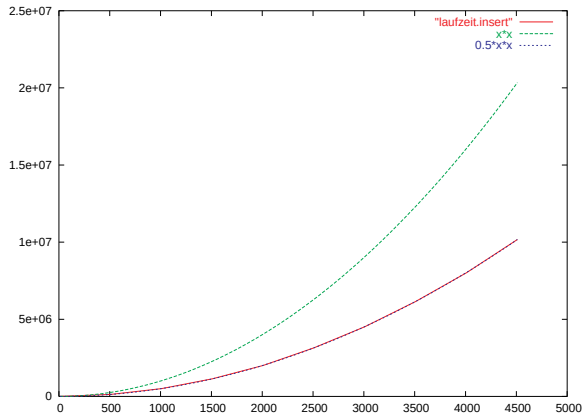
```
#include <iostream>
#include <vector>

using namespace std;

// merge fuegt die beiden benachbarten sortierten
// Teilarrays v(start1,...,end1) und
// v(end1+1,...,end2) zu einem sortierten Array zusammen,
// und kopiert dieses dann in v zurueck
int merge(vector<int>& v, unsigned int start1,
unsigned int end1, unsigned int end2)
{
  unsigned int index1=start1;
  unsigned int index2=end1+1;

  vector<int> sortiert(end2-start1+1);

  int zaehler = 0;

  for (int index3=0; index3<(int)sortiert.size(); index3++)
  {
    if ( index1 == end1+1 )
    {
      sortiert[index3] = v[index2];
      index2++;
      zaehler++;
      continue;
    }
    if ( index2 == end2+1 )
    {
      sortiert[index3] = v[index1];
      index1++;
      zaehler++;
      continue;
    }
    if ( v[index1] > v[index2] )
    {
      sortiert[index3] = v[index2];
      zaehler++;
      index2++;
    }
    else
    {
      sortiert[index3] = v[index1];
      zaehler++;
      index1++;
```

```
    }
  }

  // kopiert den neuen vektor in den alten vektor
  for (int i=0; i<(int)sortiert.size(); i++)
  {
    v[i+start1] = sortiert[i];
  }

  return zaehler;
}

int mergeSort(vector<int>& daten, unsigned int p, unsigned int r)
{
  int zaehler = 0;

  // Sind wir vielleicht schon fertig? Dann hat unser zu betrachtendes
  // Array die Laenge 0
  if (p < r)
  { // nein. schade.
    int q = (p+r)/2; // halbiere den vector

    // und sortiere rekursiv
    zaehler += mergeSort(daten, p, q);
    zaehler += mergeSort(daten, q+1, r);

    // fehlt nur noch das Kombinieren
    zaehler += merge(daten, p, q, r);
  }
  zaehler++; // fuer den p < r - Vergleich

  return zaehler;
}

// Hier ist besonders wichtig, dass wir eine
// Referenz auf den vector verwenden, da wir
// ansonsten nur eine Kopie von daten sortieren
// wuerden. Der urspruengliche vector bliebe
// unveraendert
int insertionSort(vector<int>& daten)
{
  int naechster;

  // Der Zaehler fuer die Anzahl der Vergleiche
  int zaehler = 0;

  // Wir muessen einmal ueber den Vektor iterieren,
  // wobei wir das erste Element ueberspringen koennen
```

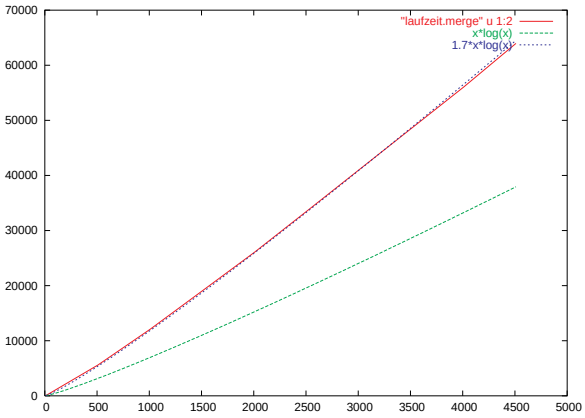
```
  // (ein einelementiger Vektor ist schon sortiert)
  for (int j=1; j<(int) daten.size(); j++)
  {
    // betrachte das naechste noch nicht einsortierte
    // Element
    naechster = daten[j];

    // und sortiere es an der passenden Stelle ein
    int i = j-1;

    while ((i >= 0) && (daten[i] > naechster))
    {
      // Wir muessen alle Zahlen die groesser sind
      // als naechster um eine Position nach rechts
      // verschieben.
      daten[i+1] = daten[i];

      --i;

      zaehler += 2; // Wir haben ja 2 Vergleiche in der while-Bedingung
    }

    daten[i+1] = naechster;
  }

  return zaehler;
}

int main(int argc, char** argv)
{
  // Ueberpruefen der Kommandozeilenargumente
  if (argc < 3)
  {
    cout << "Aufruf mittels " << argv[0] << " Startwert Verfahren" << endl;
    cout << "\t\tVerfahren == m -> mergeSort" << endl;
    cout << "\t\tVerfahren == i -> insertionSort" << endl;
    return 1;
  }


  // Kovertieren der Argumente nach integer
  int seed   = atoi(argv[1]);

  // Initialisieren des Zufallszahlengenerators
  srand(seed);

  // Hauptschleife
  for (long int laenge = 10; laenge<=5e3; laenge+=500)
```

```
  {
    // 20 Experimente durchfuehren

    long int mittelwert = 0;

    for (int i=0; i<20; i++)
    {
      // Erzeugen des vectors in der richtigen Laenge
      vector<int> zahlen(laenge);

      // Fuellen des vectors
      for (int i=0; i<laenge; i++)
      {
        zahlen[i] = rand();
      }

      // sortieren des vectors
      if (argv[2][0] == 'i') // argv[2][0] ist das erste Zeichen des
                             // 2. Kommandozeilenargumentes
      {
        mittelwert += insertionSort(zahlen);
      }
      else
      {
        mittelwert += mergeSort(zahlen, 0, zahlen.size()-1);
      }
    }

    // Mitteln
    mittelwert /= 20;

    // Und ausgeben
    cout << laenge << " " << mittelwert << endl;
  }

  return 0;
}
```

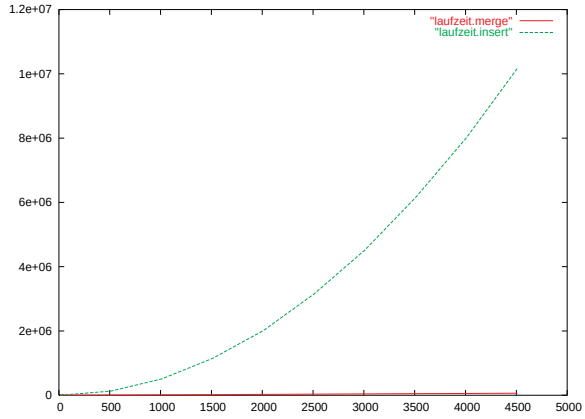
Visualisiert man die Ausgabe von `laufzeit` mittels **gnuplot**, so erhält man z.B. folgende Bilder:



Wie man sieht passt die experimentell bestimmte Laufzeit für eine Konstante C von 0.5 hervorragend zu der theoretisch erwarteten Laufzeit von n^2.



Auch für mergeSort stimmt die erwartete Laufzeit von $n * log(n)$ erstaunlich gut mit der experimentell bestimmten überein.



Zum Schluss noch ein Vergleich der gemessenen Laufzeiten. Der Unterschied zwischen n^2 und $n * log(n)$ wirkt sich offensichtlich schon für die hier betrachteten moderaten Eingabegrössen dramatisch aus!
