Dr. U. Köthe *Heidelberg, 7. Dezember 2016*

Übungsblatt 7

Aufgabe 7.1 Generische Implementation von Insertion Sort [12 Punkte]

Wir haben in der Vorlesung den Sortieralgorithmus "Insertion Sort" kennengelernt, der für kleine Arrays (bis ca. 30 Elemente) das schnellste Sortierverfahren ist. Zur Erinnerung ist der Algorithmus für Arrays vom Typ `std::vector<double>` am Ende der Aufgabe angegeben. Sie sollen diese Implementation mittels Templates für beliebige Typen generalisieren. Geben Sie Ihre Lösung im File `sort_template.cpp` ab.

(a) In der Vorlesung haben wir außerdem eine Funktion behandelt, mit der man prüfen kann, ob ein Container sortiert ist. Implementieren Sie als Vorübung eine generische Version dieser Funktion mit der Signatur:

```
template <typename ElementType, typename LessThanFunctor>
bool check_sorted(std::vector<ElementType> const & v,
                  LessThanFunctor less_than)
```

wobei `LessThanFunctor` ein Funktor ist, mit dem man eine beliebige Sortierung übergeben kann. Testen Sie diese Funktion mittels `assert()`, indem Sie mehrmals `std::sort()` und `std::random_shuffle()` aufrufen, so dass `check_sorted()` abwechselnd `true` und `false` zurückgeben sollte. Verwenden Sie als Funktor eine Lambda-Funktion, die die *absteigende* Sortierung realisiert. Beachten Sie, dass die Funktion auch für leere Arrays sowie für Arrays mit nur einem oder mit duplizierten Elementen funktionieren muss.

(b) Lösen Sie dieselbe Aufgabe mit Iteratoren, also mit der Signatur

```
template <typename Iterator, typename LessThanFunctor>
bool check_sorted(Iterator begin, Iterator end,
                  LessThanFunctor less_than)
```

Die Iteratoren sollen dabei die Semantik von *forward iterators* haben, d.h. Sie dürfen nur die folgenden Operationen verwenden:

```
Iterator iter = begin; // Kopie eines Iterators erzeugen
++iter  // Iterator zum nächsten Element bewegen
        // beachte: begin wird dadurch nicht verändert!
iter == begin // zeigen die Iteratoren auf das gleiche Element?
iter != end   // zeigen die Iteratoren auf verschiedene Elemente?
*iter         // auf das aktuelle Element zugreifen
```

Testen Sie Ihre Implementation wiederum mittels `assert()`.

(c) Implementieren Sie "Insertion Sort" mit der Signatur

```
template <typename ElementType, typename LessThanFunctor>
void insertion_sort(std::vector<ElementType> & v,
                    LessThanFunctor less_than)
```

und testen Sie Ihre Implementation wie in Teilaufgabe (a), indem Sie `std::sort()` durch Ihre Funktion ersetzen (wiederum mit absteigender Sortierung).

(d) Implementieren Sie "Insertion Sort" mit der Signatur

```
template <typename Iterator, typename LessThanFunctor>
void insertion_sort(Iterator begin, Iterator end,
                    LessThanFunctor less_than)
```

und testen Sie Ihre Implementation wie in Teilaufgabe (b), indem Sie `std::sort()` durch Ihre Funktion ersetzen, ebenfalls mit absteigender Sortierung. Die Iteratoren sollen die Semantik von *bidirectional iterators* haben, d.h. zusätzlich zu den Operationen in (b) ist jetzt auch `--iter` erlaubt, um einen Iterator zum vorhergehenden Element zu bewegen. Tipp: Für den unbekannten Elementtyp des Containers können Sie jetzt den Universaltyp `auto` verwenden.

Nicht-generische Implementation von Insertion Sort:

```
// pass-by-reference, weil v verändert werden soll
void insertion_sort(std::vector<double> & v)
{
    for(int k=1; k<v.size(); ++k)
    {
        double current = v[k]; // Element, das jetzt einsortiert wird
        int j = k;             // Anfangsposition der Lücke
        while(j > 0)
        {
            if(current < v[j-1])
            {
                v[j] = v[j-1]; // Lücke eine Position nach links
            }
            else
            {
                break;         // Lücke jetzt an richtiger Position
            }
            --j;
        }
        v[j] = current;        // Element in die Lücke einfügen
    }
}
```

Aufgabe 7.2 Aufwand von Insertion Sort [14 Punkte]

Bei Sortieralgorithmen ist es üblich, den Aufwand durch die notwendige Anzahl der Aufrufe des Vergleichsoperators `"<"` auszudrücken. Bestimmen Sie auf diese Weise den Aufwand von "Insertion Sort" für ein Array der Größe n.

(a) Fall 1: günstigster Fall. Hier kommt der Algorithmus mit einem Minimum an Vergleichen aus. Wann tritt dieser Fall ein? Wieviele Vergleiche (ausgedrückt als Funktion $f_1(n)$) sind dann notwendig? Vereinfachen Sie $f_1(n)$ mit Hilfe der Ω-Notation, d.h. suchen Sie eine möglichst einfache Funktion $g_1(n)$ so dass $f_1(n) \in \Omega(g_1(n))$. Warum muss hier die Ω-Notation verwendet werden?

(b) Fall 2: schlechtester Fall. Jetzt benötigt der Algorithmus ein Maximum an Vergleichen. Wann tritt dieser Fall ein? Wieviele Vergleiche (ausgedrückt als Funktion $f_2(n)$) sind nun notwendig? Vereinfachen Sie die Funktion $f_2(n)$ mit Hilfe der $\mathcal{O}$-Notation zu $f_2(n) \in \mathcal{O}(g_2(n))$. Warum muss hier die $\mathcal{O}$-Notation verwendet werden?

(c) Fall 3: typischer Fall. Hierfür nimmt man an, dass die Daten anfangs zufällig angeordnet sind. Dadurch läuft die innere Schleife von Insertion Sort im Durchschnitt bis zur Hälfte durch, bevor die `break`-Anweisung erreicht wird. Wieviele Vergleiche (ausgedrückt als Funktion $f_3(n)$) sind unter dieser Annahme notwendig? Vereinfachen Sie die Funktion $f_3(n)$ mit Hilfe der $\mathcal{O}$-Notation zu $f_3(n) \in \mathcal{O}(g_3(n))$.

(d) Bestimmen Sie empirisch die in der Ω- bzw. $\mathcal{O}$-Notation versteckten Konstanten für die drei Fälle und vergleichen Sie mit dem entsprechenden Resultat für `std::sort()`. Geben Sie den zugehörigen Code im File `sort_time.cpp` ab.

Schreiben Sie dafür zunächst Funktionen `insertion_sort_best_time()`, `insertion_sort_worst_time()` und `insertion_sort_typical_time()`, die die Zeit $t_{i,n}$ messen, die im Fall i für das Sortieren eines Arrays der Größe n benötigt wird. Die entsprechende Funktion für `std::sort()` sieht z.B. so aus (dazu muss C++11 aktiviert und `<chrono>` inkludiert sein):

```
double std_sort_time(int n)
{
    // fülle Array der Größe n mit zufällig angeordneten Daten
    std::vector<double> v(n);
    for(int k=0; k<n; ++k)
    {
        v[k] = k;
    }
    std::random_shuffle(v.begin(), v.end());

    // bestimme Anfangs- und Endzeit für das Sortieren
    auto start = std::chrono::high_resolution_clock::now();
    std::sort(v.begin(), v.end());
    auto stop  = std::chrono::high_resolution_clock::now();

    // berechne die Zeitdauer (in Sekunden) und gib sie zurück
    std::chrono::duration<double> diff = stop - start;
    return diff.count();
}
```

Für den typischen Fall von Insertion Sort ersetzen Sie einfach `std::sort()` durch `insertion_sort()`. Für den günstigsten und schlechtesten Fall müssen Sie außerdem das Array anders füllen.

Rufen Sie diese Funktionen nun für verschiedene Arraygrößen auf. Wählen Sie die Größen so, dass die Ausführung zwischen 0.02s und 3s dauert (kürzere Zeiten können zu ungenauen Messungen führen, besonders beim MinGW-Compiler). Wiederholen Sie die Messung für jedes i und n dreimal und verwenden Sie die schnellste Zeit (gelegentlich kommen falsche Zeiten heraus, weil die Ausführung Ihres Programms vom Betriebssystem kurz unterbrochen wurde, um z.B. Emails zu checken).

Berechnen Sie jetzt für jeden Fall i und jede Größe n den Quotienten $C_{i,n} = \frac{t_{i,n}}{g_i(n)}$. Wenn Ihr Code und Ihre Lösungen für $g_i(n)$ korrekt sind, sollten die $C_{i,n}$ für jedes $i \in 1, 2, 3$ näherungsweise konstant, also unabhängig von n, sein. Berechnen Sie C_i

als Mittelwert über $C_{i,n}$. Bei `std::sort()` benutzen Sie entsprechend $C_{\text{sort},n} = \frac{t_{\text{sort},n}}{n \cdot \log(n)}$, weil dieser Algorithmus die Komplexität $\mathcal{O}(n \cdot \log(n))$ hat. Drücken Sie Ihre Ergebnisse durch Funktionen $t_i(n) \approx C_i\, g_i(n)$ bzw. $t_{\text{sort}}(n) \approx C_{\text{sort}}\, n\, \log(n)$ aus, die die Laufzeit als Funktion der Arraygröße angeben.

Führen Sie das Experiment sowohl mit Codeoptimierung (Compileroption `"-O2"` bei g++ bzw. `"/O2"` bei Visual Studio) als auch ohne (Compileroption `"-O0"` bzw. `"/Od"`) aus. Vergleichen Sie die Ergebnisse der acht Varianten! Beantworten Sie insbesondere die Fragen: Bis zu welchem n kann Insertion Sort im typischen Fall mit `std::sort()` mithalten? Wie groß ist der Effekt der Codeoptimierung?

Aufgabe 7.3 Algorithmische Komplexität [14 Punkte]

(a) Ein Programm mit algorithmischer Komplexität $f(n)$ benötigt zur Bearbeitung von $n = 32$ Datenelementen 5 Sekunden. Berechnen Sie (unter Angabe des Lösungswegs), wie lange das Programm für doppelt so viele Datenelemente, also für $n = 64$ benötigt, wenn die Komplexität $f(n)$

i) $\log_2(n)$ ii) n iii) $n \cdot \log_2(n)$ iv) n^2 v) 2^n

beträgt.

(b) Beweisen Sie mit Hilfe der Definition der $\mathcal{O}$-Notation, dass die Basis des Logarithmus bei der $\mathcal{O}$-Notation ohne Bedeutung ist. Das heißt, es gilt für alle $a, b > 0$:

$$\log_a(n) \quad \in \quad \mathcal{O}(\log_b(n))$$

(c) Sortieren Sie die folgenden Funktionen danach, wie schnell sie mit n wachsen, beginnend mit der effizientesten. Beweisen Sie die Korrektheit Ihrer Reihenfolge.

$$2^n \qquad n \cdot \log(n) \qquad n^{1/2} \qquad \log(n) \qquad n^2$$

Tipp: Raten Sie zunächst die richtige Reihenfolge (z.B. indem Sie die Funktionen zeichnen) und beweisen Sie dann für alle in der Reihenfolge benachbarten Paare, dass die jeweils rechte Funktion $f_{\text{rechts}}(n)$ schneller als die linke $f_{\text{links}}(n)$ wächst. Dazu können Sie z.B. das größte x berechnen, bei dem sich die beiden Funktionen schneiden, d.h. $f_{\text{rechts}}(x) = f_{\text{links}}(x)$. Für alle $n > x$ muss dann $f_{\text{rechts}}(n) > f_{\text{links}}(n)$ gelten. Eine andere Möglichkeit besteht darin, den Grenzwert

$$\lim_{n \to \infty} \frac{f_{\text{rechts}}(n)}{f_{\text{links}}(n)}$$

zu bestimmen (beispielsweise mit der Regel von l'Hospital). Ist die Reihenfolge korrekt, sind diese Grenzwerte für alle Paare unendlich.

Bitte laden Sie Ihre Lösung spätestens bis 14. Dezember 2016, 9:00 Uhr in Moodle hoch.