Adaptive Sortierverfahren

Diplomarbeit
am Institut für Informatik
der TU Clausthal

Jörg Czeranski

Oktober 1998

Ich versichere, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Hilfsmittel und Quellen verwendet habe.

Clausthal-Zellerfeld, den 15. Oktober 1998

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit beschäftigt sich mit der Aufgabe, eine gegebene Liste zu sortieren und dabei auszunutzen, daß die Elementreihenfolge in der Liste eventuell nicht ganz zufällig sondern auf irgendeine Weise vorsortiert ist.

Klassische Sortierverfahren wie Mergesort oder Heapsort profitieren von einer solchen Vorsortierung nicht.

Es kann aber gute Gründe dafür geben, daß ein Sortierverfahren mit einer überwiegend oder auch nur wenig vorsortierten Liste als Eingabe schneller ablaufen soll als für Listen mit zufälliger Elementreihenfolge (ein solches Sortierverfahren nennt man adaptiv):

- Die Eingabeliste kann aufgrund ihrer vorherigen Bearbeitung teilweise vorsortiert sein, beispielsweise kann sie vorher nach einem Kriterium sortiert worden sein, das mit dem nächsten zu verwendenden korelliert.
- Die Liste kann aus einer bereits sortierten durch geringfügige Änderung – wie Einfügen, Verändern oder Umordnen einiger Elemente – entstanden sein.
- Die zu sortierende Liste kann allgemeiner durch einen Prozeß entstanden sein, der immer oder häufig Listen mit geringer Unordnung erzeugt.
- Kann eine Routine nur auf einer sortierten Liste operieren, ist aber nicht bekannt, ob die Eingabeliste immer sortiert ist, so kann man die Eingabeliste grundsätzlich mit einem adaptiven Verfahren sortieren.

  Ist die ursprüngliche Liste nicht sortiert, so ist das Sortieren sowieso notwendig, ist sie dagegen sortiert, so verursacht das adaptive Sortieren nur geringe zusätzliche Kosten.

Um dieses sehr weiträumige Thema weiter einzugrenzen, werden die Sortierverfahren auf vergleichsbasierte eingeschränkt: solche, die auf Listenelemente nur als Einheiten zugreifen. Nur das Vergleichen von Elementen mittels einer vorgegebenen Vergleichsfunktion ist erlaubt, keine direkten Zugriffe auf Sortierschlüssel, wie sie beispielsweise Radix-Sort verwendet.

Auch parallele Algorithmen sind ausgenommen. Das Sortieren auf Parallelrechnern ist zwar in letzter Zeit sehr in den Mittelpunkt des Interesses gerückt, Literatur zu adaptiven Sortierverfahren beschränkt sich aber überwiegend auf serielle Algorithmen – eine Ausnahme ist beispielsweise [LP96].

Untersucht werden in dieser Arbeit zuerst Maße, mittels derer man Unordnung quantitativ bestimmen kann, dann Algorithmen, die sich an die vorhandene Unordnung anpassen, insbesondere solche, die die Größe der Unordnung nicht als explizite Eingabe benötigen.

2 Notation

Dieses Kapitel führt die verwendete Notation mit einigen grundlegenden Beispielen zu ihrer Anwendung ein.

Die Notation in Abschnitt 2.1 ist grundlegend und wird in der gesamten Arbeit verwendet, auf die Abschnitte 2.2 und 2.3 zu Permutationen und Komplexität wird dagegen überwiegend ab Kapitel 5 bei der Untersuchung der Optimalität Bezug genommen, Permutationen werden außerdem in Abschnitt 4.5 beim Vorstellen des *Exc*-Maßes verwendet.

2.1 Listen

$\mathbf{N} = \{0, 1, 2, \dots\}$ bezeichne die Menge der natürlichen Zahlen.

Eine endliche Folge von Elementen einer total geordneten Menge heiße im folgenden *Liste*. Die leere Liste ist zugelassen. Soweit es nicht anders angegeben ist, wird aber davon ausgegangen, daß alle Elemente einer Liste verschieden sind. Kapitel 5 geht auf die Frage ein, welche Auswirkung diese Einschränkung hat.

Für Elemente $x_1, \dots, x_n$ ($n \in \mathbf{N}$) einer Menge, auf der eine totale Ordnung „$\leq$" definiert ist, bezeichne $(x_1, \dots, x_n)$ die n-elementige Liste mit den Elementen x_1 bis x_n, speziell (x) eine einelementige Liste und $()$ die leere Liste mit $n = 0$ (nicht zu verwechseln mit der *leeren Menge* $\emptyset$).

i wird als *Index* oder *Position* des Elements x_i in der Liste $(x_1, \dots, x_n)$ bezeichnet.

XY bezeichne die *Konkatenation* $(x_1, \dots, x_n, y_1, \dots, y_m)$ der beiden Listen $X = (x_1, \dots, x_n)$ und $Y = (y_1, \dots, y_m)$ und $|X|$ die *Länge* n der Liste $X = (x_1, \dots, x_n)$.

Sei $f : \{1, \dots, m\} \longrightarrow \{1, \dots, n\}$ eine streng monoton steigende Funktion (damit ist f injektiv, und es gilt $m \leq n$). Mit einem so gewählten f heiße

$$Y = (x_{f(1)}, \dots, x_{f(m)}) \tag{2.1}$$

eine *Teilliste* von $X = (x_1, \dots, x_n)$, in Zeichen $Y \sqsubseteq X$. Die in Y enthaltenen Elemente bilden eine Teilmenge der in X enthaltenen und haben untereinander dieselbe Reihenfolge wie in X.

Insbesondere erhält man mit $m = 0$ und $f : \emptyset \longrightarrow \{1, \dots, |X|\}$ die leere Liste als Teilliste jeder Liste X.

Beispiel: Sei

$$X = (x_1, x_2, x_3, x_4, x_5, x_6) = (1, 3, 4, 6, 2, 5)$$

eine Liste über der Menge der natürlichen Zahlen mit der üblichen Ordnung $\leq$ und

$$f : \{1, 2\} \longrightarrow \{1, \dots, 6\},$$
$$f(1) = 3, \quad f(2) = 5.$$

f ist streng monoton steigend, folglich ist

$$Y = (x_{f(1)}, x_{f(2)}) = (x_3, x_5) = (4, 2)$$

eine Teilliste von X, kurz $(4, 2) \sqsubseteq X$.

Ein Paar (a, b) heiße *Inversion* in der Liste X, wenn (a, b) eine Teilliste von X ist und a größer ist als b. Eine Liste ist offensichtlich genau dann (aufsteigend) sortiert, wenn sie keine Inversionen hat.

In obigem Beispiel ist wegen $4 > 2$ die Teilliste $(4, 2)$ auch eine Inversion in der Liste X.

Jede Teilmenge E_Y der Menge E_X aller Elemente einer festen Liste X bestimmt eindeutig eine Teilliste Y von X, die genau die Elemente von E_Y als Listenelemente enthält. Dadurch bildet $\sqsubseteq$ auf der Menge der Teillisten einer festen Liste X eine partielle Ordnung, die zu der partiellen Ordnung $\subseteq$ auf der Menge aller Teilmengen einer festen Menge E_X (der Potenzmenge von E_X) isomorph ist.

Weiterhin ist $\sqsubseteq$ auch auf der Menge aller Listen über einer festen Menge eine partielle Ordnung.

2.2 Permutationen

Die Liste $Y = (y_1, \dots, y_m)$ heiße *Permutation* der Liste $X = (x_1, \dots, x_n)$, wenn es eine bijektive Funktion π von $\{1, \dots, n\}$ auf $\{1, \dots, m\}$ gibt, die jedem Index i eines Elements von X einen Index j eines Elements von Y mit $x_i = y_j$ zuordnet:

$$j = \pi(i), \quad x_i = y_j \qquad \text{für alle } i \in \{1, \dots, n\}. \tag{2.2}$$

Insbesondere ist dann $m = n$ und die Menge aller in X auftretenden Elemente stimmt mit der Menge aller in Y auftretenden Elemente überein.

Ferner gilt

$$X = (y_{\pi(1)}, \dots, y_{\pi(n)}) \tag{2.3}$$

und

$$Y = (x_{\pi^{-1}(1)}, \dots, x_{\pi^{-1}(n)}). \tag{2.4}$$

Man beachte, daß π auf $\{1, \ldots, n\}$ bijektiv ist und damit π^{-1} wohldefiniert ist.

Weiterhin wird auch π selbst als *Permutation* bezeichnet, genauer als die Permutation, die X in Y überführt. Ist Y sortiert, so wird π auch als die Permutation, die X sortiert, bezeichnet und $X^{\leq}$ für Y geschrieben.

Diese Verwendung entspricht dem modernen Begriff der Permutation als Bijektion auf einer (oft endlichen) Menge.

Es sollte jeweils aus dem Kontext klar hervorgehen, ob „Permutation“ eine permutierte Liste oder eine permutierende Funktion bezeichnet. Um Verwechslungen zu vermeiden, werden Listen im folgenden immer mit lateinischen Großbuchstaben und Permutationsfunktionen mit griechischen Kleinbuchstaben bezeichnet.

2.3 Komplexität

Gibt es zu Funktionen $f, g : \mathbf{N} \longrightarrow \mathbf{R}$ (dabei bezeichne $\mathbf{R}$ die Menge der reellen Zahlen) ein reelles $c > 0$ und ein natürliches n_0, so daß

$$f(n) \leq c \cdot g(n) \qquad \text{(für alle } n \geq n_0\text{)} \tag{2.5}$$

gilt, dann liegt f in der Komplexitätsklasse $O(g)$, in Zeichen:

$$f = O(g) \quad \text{oder} \quad f(n) = O(g(n)). \tag{2.6}$$

Gibt es zu ebensolchen f, g ein reelles $d > 0$ und ein natürliches n_0, so daß

$$f(n) \geq d \cdot g(n) \qquad \text{(für alle } n \geq n_0\text{)} \tag{2.7}$$

gilt, so schreibt man

$$f = \Omega(g) \quad \text{oder} \quad f(n) = \Omega(g(n)). \tag{2.8}$$

Schließlich sei

$$f = \Theta(g) \quad \text{oder} \quad f(n) = \Theta(g(n)), \tag{2.9}$$

wenn (2.5) und (2.7) gleichzeitig gelten.

Zur Verdeutlichung der Anwendungsweise diene der Beweis des folgenden Satzes, der in Abschnitt 5.2.1 gebraucht wird:

Für Funktionen f und g mit nichtnegativen Werten ist

$$\max\{f, g\} = \Theta(f + g). \tag{2.10}$$

Beweis: Ist bei festem n

$$0 \leq f(n) \leq g(n), \tag{2.11}$$

so gilt

$$\begin{aligned} \frac{1}{2}(f(n) + g(n)) &\leq \frac{1}{2}(g(n) + g(n)) \\ &= g(n) = \max\{f(n), g(n)\} \\ &\leq f(n) + g(n). \end{aligned} \tag{2.12}$$

Analog gilt auch für $0 \leq g(n) \leq f(n)$:

$$\frac{1}{2}(f(n) + g(n)) \leq \max\{f(n), g(n)\} \leq f(n) + g(n). \tag{2.13}$$

Wählt man mit obiger Bezeichnung $n_0 = 0$, $c = 1$ und $d = \frac{1}{2}$, so folgt unmittelbar die Behauptung (2.10).

2.3.1 Zeit- und Speicherkomplexität

Vergleich zweier Elemente, Kopieren eines Elements und Vertauschen eines Elements werden als jeweils eine Operation gezählt.

Für die Speicherkomplexität wird die Anzahl der Zeiger in eine Liste gezählt.

Strenggenommen entspricht dabei beim Bearbeiten einer n-elementigen Liste eine Speicherkomplexität von $O(1)$ Zeigern einer von $O(\log n)$ Bit. Aus zwei Gründen wird dieses Detail aber nicht berücksichtigt:

Zum einen ist die Listenlänge in der Praxis meistens durch die größte in einem Maschinenwort darstellbare Zahl begrenzt – dann ist die Speicherkomplexitäten auf diese Weise in Einheiten von Maschinenworten gemessen – zum anderen müßte man, wenn man noch längere Listen zuließe, auch einzelne grundlegende Operationen auf Zeigern als $O(\log n)$ Operationen zählen.

Legt man dagegen für den Speicherbedarf die Größe eines Zeigers als Einheit zugrunde, kann man die Komplexität durch Multiplikation mit $\log n$ in eine auf Bit basierende umrechnen. Der einzige Nachteil, der dadurch entsteht, ist daß sublogarithmische Terme bei der Speicherkomplexität nicht korrekt berücksichtigt werden.

3 Vorsortiertheit

Um Ordnung bzw. Unordnung in einer zu sortierenden Liste zu messen, wird in der Literatur oft Mannilas Definition [Man85] für Vorsortiertheitsmaße ("measures of presortedness") verwendet:

Eine Funktion M, die jeder Liste eine natürliche Zahl zuordnet, heiße *Maß für Vorsortiertheit*, wenn sie folgende fünf Bedingungen erfüllt: (dann gibt diese Zahl anschaulich den „Abstand" der Liste von der zugehörigen sortierten Liste an)

1. Für jede (aufsteigend) sortierte Liste X ist $M(X) = 0$.

2. Seien $X = (x_1, \ldots, x_n)$ und $Y = (y_1, \ldots, y_n)$.

   Gilt $x_i < x_j$ genau dann, wenn $y_i < y_j$ gilt (für alle $i, j \in \{1, \ldots, n\}$), so ist $M(X) = M(Y)$.

   Diese Bedingung stellt sicher, daß die Vorsortiertheit nur von der Permutation abhängt, die eine gegebene Liste in eine sortierte Liste überführt, nicht von den Werten der Listenelemente.

   In Zusammenhängen, in denen diese Bedingung erfüllt ist, genügt es daher, statt beliebiger Listen nur Permutationen der Listen $(1, \ldots, n)$ mit $n \in \mathbf{N}$ zu betrachten.

3. Ist X eine Teilliste von Y, dann gilt $M(X) \leq M(Y)$.

   Diese und die folgenden Bedingungen beziehen sich auf die Anzahl der beim Sortieren auszuführenden Operationen:

   Es kann nicht aufwendiger sein, eine Teilliste zu sortieren als die gesamte Liste; also sollte eine Teilliste auch in keinem Fall als ungeordneter angesehen werden als die Gesamtliste, in der sie enthalten ist.

4. Ist jedes Element der Liste X kleiner oder gleich jedem Element der Liste Y, dann gilt $M(XY) \leq M(X) + M(Y)$.

   In diesem Fall kann man X und Y einzeln sortieren und dann konkatenieren, um die zu XY gehörende sortierte Liste zu erhalten. Deshalb sollte die Unordnung in der Gesamtliste nicht größer sein können als die Summe der Unordnung der Teile.

5. Für jedes a gilt $M((a)X) \leq |X| + M(X)$.

   Um $(a)X$ zu sortieren, kann man zuerst X sortieren und dann a in $|X|$ Operationen einfügen. Auch hier wird daher die entsprechende Schranke verlangt.

Die Bedingungen 1 und 5 könnte man auch in abgeschwächter Form verwenden oder auf sie verzichten – beispielsweise wird in [MEP96] das *Runs*-Maß (Abschnitt 4.3) mit einem Minimum von 1 statt 0 definiert.

Man würde dadurch aber die obere Schranke (6.5) für die Werte eines Maßes verlieren, und es ist fraglich, ob ein Verzicht auf die Bedingungen einen nennenswerten Vorteil bringt oder ob es günstiger ist, die Bedingungen durch eine Normierung des Maßes einzuhalten.

Die Bedingungen 2, 3 und 4 zur Elementunabhängigkeit, zur Monotonie bei einer Teillisten-Beziehung und zur Konkatenation sind sicher in dieser Form notwendig.

Estivill-Castro untersucht in seiner Dissertation [Est91] den Raum der Maße genauer. Zum einen schränkt er die Bedingungen im wesentlichen auf 1 und 2 ein – Maße, die diesen beiden Bedingungen genügen, heißen *Unordnungsmaße* ("measures of disorder") – zum anderen führt er neue Bedingungen ein, die nicht auf Teillisten und Konkatenation, sondern auf Permutationen beruhen – solche Maße heißen *normale* Unordnungsmaße und stehen mit Metriken in Zusammenhang.

Es gibt normale Unordnungsmaße, die keine Maße für Vorsortiertheit sind, und umgekehrt. Viele der wichtigen Maße sind aber sowohl normale Unordnungsmaße als auch Vorsortiertheitsmaße.

In dieser Arbeit wird auf Unordnungsmaße nicht weiter eingegangen, da es ihren Rahmen sprengen würde.

4 Bekannte Maße

Zur Erläuterung der Notation in den Abbildungen betrachte man die Liste

$$X = (x_1, \dots, x_6) = (1, 3, 4, 2, 6, 5)$$

in Abbildung 1. Jeder Punkt stellt ein Listenelement dar. Unten sind die Indizes der Elemente in der Liste aufgetragen, links aufsteigend sortiert ihre Werte. In der Abbildung ist $x_4 = 2$ durch Hilfslinien hervorgehoben.

Benachbarte Elemente sind durch Linien verbunden, so daß man an einer steigenden Linie ein sortiertes Paar erkennt – z. B. $(1, 3)$ am Anfang der Liste – und an einer fallenden eine Inversion – z. B. $(6, 5)$ am Ende.

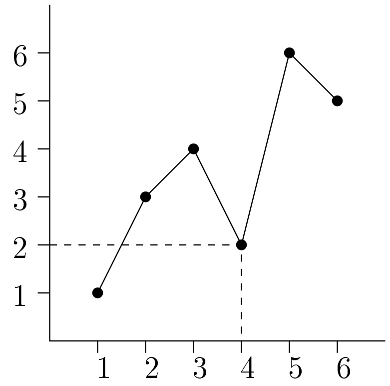


Abbildung 1: $X = (1, 3, 4, 2, 6, 5)$, $x_4 = 2$

Bedingung 2 aus Kapitel 3 stellt sicher, daß man für die Betrachtung der Maße statt einer Permutation der Liste $(1, \dots, n)$ mit $n \in \mathbf{N}$ gleichwertig etwa eine Liste von Namen verwenden kann, die alphabetisch sortiert werden soll:

$$N = (a_1, \dots, a_6) = (\text{Herrmann, Peters, Schmidt, Meier, Werner, Schulze})$$

Die zugehörige sortierte Liste ist

$$N^{\leq} = (\text{Herrmann, Meier, Peters, Schmidt, Schulze, Werner}).$$

Man definiere nun eine Funktion f, die jedem Element der Liste N seine Position in der sortierten Liste $N^{\leq}$ zuordnet:

a	Herrmann	Peters	Schmidt	Meier	Werner	Schulze
$f(a)$	1	3	4	2	6	5

Ersetzt man in N jedes Element durch sein Bild unter f, so erhält man eine Liste, die eine Permutation der Liste $(1, \ldots, 6)$ ist und sich bezüglich beliebiger Maße genauso verhält wie die ursprüngliche Liste N.

Analog kann man zur Vereinfachung der Betrachtung jede n-elementige Liste in eine gleichwertige Permutation von $(1, \ldots, n)$ überführen. Diese Liste bzw. die zugehörige Abbildung f tatsächlich zu berechnen, entspricht natürlich dem Sortieren der ursprünglichen Liste.

Die aus N erhaltene neue Liste $(f(a_1), \ldots, f(a_6))$ ist die oben eingeführte Liste $X = (1, 3, 4, 2, 6, 5)$, die im folgenden als Beispiel dienen soll.

4.1 Die Maße M_0 und M_{01}

Sei $M_0(X) = 0$ für jede Liste X und sei $M_{01}(X) = 0$ für jede sortierte Liste und 1 sonst. Diese beiden Funktionen sind triviale Beispiele für Vorsortiertheitsmaße.

4.2 Das Inv-Maß

Sei $Inv(X) = |\{(a, b) \sqsubseteq X \mid a > b\}|$, die Anzahl der Inversionen in X, d. h. der Paare, deren Elemente in der falschen Reihenfolge sind. [Man85][EW89]

Beispiel: $(1, 3, 4, 2, 6, 5)$ enthält die Inversionen $(3, 2)$, $(4, 2)$, und $(6, 5)$ (Abbildung 2), also ist $Inv((1, 3, 4, 2, 6, 5)) = 3$.

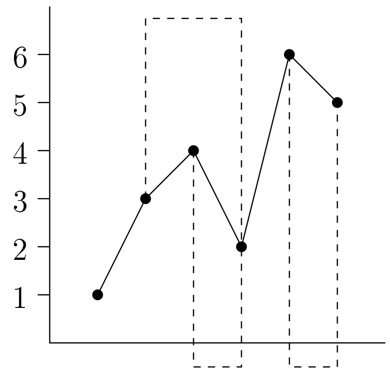


Abbildung 2: $X = (1, 3, 4, 2, 6, 5)$ mit den drei Inversionen $(3, 2)$, $(4, 2)$, $(6, 5)$

Inv ist ein Maß:

1. Eine sortierte Liste X enthält keine Inversionen, also ist $Inv(X) = 0$.
2. $X = (x_1, \ldots, x_n)$, $Y = (y_1, \ldots, y_n)$. Gilt $x_i < x_j$ genau dann, wenn $y_i < y_j$ gilt, ist (x_i, x_j) genau dann eine Inversion, wenn (y_i, y_j) eine ist, und damit $Inv(X) = Inv(Y)$.

3. Sei $X \sqsubseteq Y$, dann ist jede Inversion in X auch eine Inversion in Y, also $Inv(X) \leq Inv(Y)$.

4. Ist jedes Element der Liste X kleiner oder gleich jedem Element der Liste Y, so liegt jede Inversion von XY entweder ganz in X oder ganz in Y, also $Inv(XY) = Inv(X) + Inv(Y)$.

5. Sei $X = (x_1, \ldots, x_n)$. Jede Inversion von $(a)X$ liegt entweder ganz in X oder hat die Form (a, x_i) mit $i \in \{1, \ldots, n\}$, also gilt

$$Inv((a)X) \leq n + Inv(X).$$

4.3 Das *Runs*-Maß

Sei $Runs(X) = |\{i \mid 1 \leq i < |X|,\ x_i > x_{i+1}\}|$; das ist die Anzahl der Inversionen, deren Elemente in $X = (x_1, \ldots, x_n)$ benachbart sind. [EW89]

Seien $(x_{i_1}, x_{i_1+1}), \ldots, (x_{i_k}, x_{i_k+1})$ mit $i_1 < \ldots < i_k$ diese Inversionen. Um Bezug auf die Ränder der Liste X nehmen zu können, ergänze man $i_0 = 0$ und $i_{k+1} = n$. Dann ist $X = Y_0 Y_1 \ldots Y_k$ mit $Y_j = (x_{i_j+1}, \ldots, x_{i_{j+1}})$ für alle $j \in \{0, \ldots, k\}$.

Zu jeder Inversion benachbarter Elemente (x_{i_j}, x_{i_j+1}) liegen die beiden Elemente in verschiedenen Y, nämlich in Y_{j-1} und Y_j. Damit gilt für jedes $(x_l, x_{l+1}) \sqsubseteq Y_j$, daß $x_l \leq x_{l+1}$ ist, denn sonst wäre (x_l, x_{l+1}) eine weitere solche Inversion.

Also ist $x_{i_j+1} \leq \ldots \leq x_{i_{j+1}}$ für $Y_j = (x_{i_j+1}, \ldots, x_{i_{j+1}})$, Y_j ist folglich sortiert. Die Y_j sind damit die maximalen zusammenhängenden sortierten Teillisten von X – diese Teillisten werden auch "runs" (Läufe) genannt – und $Runs(X) = k$ ist eins weniger als ihre Anzahl. Diese Verminderung ist eine Normierung, die garantiert, daß *Runs* für eine sortierte Liste den Wert 0 hat, wie es für ein Maß erforderlich ist.

Beispiel: $X = (1, 3, 4, 2, 6, 5)$, die fraglichen Inversionen sind $(4, 2)$ und $(6, 5)$, und somit $Runs(X) = 2$ (die Elemente der dritten Inversion von X, $(3, 2)$, sind in X nicht benachbart). Weiterhin sind $Y_0 = (1, 3, 4)$, $Y_1 = (2, 6)$ und $Y_2 = (5)$. (Siehe Abbildung 3)

Runs ist ein Maß:

1. Eine sortierte Liste X enthält keine Inversionen, insbesondere keine mit benachbarten Elementen, also ist $Runs(X) = 0$.

2. $X = (x_1, \ldots, x_n)$, $Y = (y_1, \ldots, y_n)$. Gilt $x_i < x_j$ genau dann, wenn $y_i < y_j$ gilt, ist (x_i, x_j) genau dann eine Inversion, wenn (y_i, y_j) eine ist. Weiterhin sind x_i und x_j genau dann benachbart, wenn es auch y_i und y_j sind. Folglich ist $Runs(X) = Runs(Y)$.

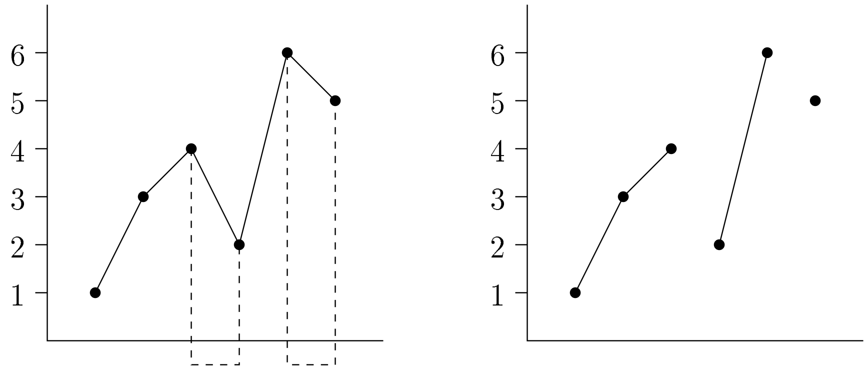


Abbildung 3: $X = (1, 3, 4, 2, 6, 5)$, links mit den zwei relevanten Inversionen $(4, 2)$, $(6, 5)$, rechts mit den drei Teillisten $(1, 3, 4)$, $(2, 6)$ und (5)

3. Sei $X = (x_1, \ldots, x_n)$ eine Teilliste von $Y = (y_1, \ldots, y_m)$, und sei $f : \{1, \ldots, n\} \longrightarrow \{1, \ldots, m\}$ die Funktion, die die Elemente von X in die Elemente von Y abbildet:

   $y_{f(i)} = x_i$ für $i \in \{1, \ldots, n\}$ und $f(i) < f(i+1)$ für $i \in \{1, \ldots, n-1\}$.

   Sei nun (x_j, x_{j+1}) eine beliebige Inversion benachbarter Elemente in X. Es gilt $y_{f(j)} = x_j > x_{j+1} = y_{f(j+1)}$. Enthielte $(y_{f(j)}, \ldots, y_{f(j+1)})$ keine Inversion benachbarter Elemente, so würde $y_k \leq y_{k+1}$ für alle k mit $f(j) \leq k < f(j+1)$ gelten. Dann wäre aber $y_{f(j)} \leq y_{f(j+1)}$ im Widerspruch zu $x_j > x_{j+1}$.

   Also muß zu jeder Inversion benachbarter Elemente (x_j, x_{j+1}) in X auch eine in Y liegen, speziell in $(y_{f(j)}, \ldots, y_{f(j+1)})$. Und zu verschiedenen solchen Inversionen in X sind auch die Inversionen in Y verschieden, da $(y_{f(j)}, \ldots, y_{f(j+1)})$ und $(y_{f(j')}, \ldots, y_{f(j'+1)})$ mit $j \neq j'$ keine zwei Elemente gemeinsam haben. Folglich ist $Runs(X) \leq Runs(Y)$.

4. Seien $X = (x_1, \ldots, x_n)$ und $Y = (y_1, \ldots, y_m)$, und gelte $x_i \leq y_j$ für alle $i \in \{1, \ldots, n\}$ und $j \in \{1, \ldots, m\}$. In XY sind die Elemente benachbart, die in X oder in Y benachbart sind, und zusätzlich (x_n, y_1), falls X und Y beide nicht leer sind. Es gilt aber $x_n \leq y_1$, also ist (x_n, y_1) keine Inversion und liefert keinen Beitrag zu $Runs(XY)$. Folglich ist $Runs(XY) = Runs(X) + Runs(Y)$.

5. Sei $X = (x_1, \ldots, x_n)$. Ist $n = 0$, so ist $(a)X$ sortiert, und es gilt

   $$Runs((a)X) = 0 = |X| + Runs(X).$$

   Andernfalls enthält $(a)X$ dieselben Inversionen benachbarter Elemente wie X und höchstens noch zusätzlich (a, x_1). Für $n > 0$ gilt also

   $$Runs((a)X) \leq 1 + Runs(X) \leq |X| + Runs(X).$$

4.4 Das *Rem*-Maß

Sei $Rem(X) = |X| - \max\{|Y| \mid Y \sqsubseteq X, Y \text{sortiert}\}$, die Anzahl der Elemente, die man mindestens aus X streichen (engl. "remove") muß, um eine sortierte, in der Regel kürzere Liste zu erhalten. [Man85][EW89]

Dies ist genau die Hälfte der Anzahl der Streich- und Einfügeoperationen, die benötigt werden, um X zu sortieren: Für jede Streichung muß genau eine Einfügung vorgenommen werden, um die ursprüngliche Länge der Liste wieder zu erhalten. Weiterhin kann man offensichtlich nicht mehr Operationen benötigen, wenn man zuerst die Streichungen und dann die Einfügungen durchführt.

Man betrachte nun eine Liste Z, die man nach den Streichungen und vor den Einfügungen erhält (deren Anzahl beträgt jeweils $|X| - |Z|$). Z muß eine Teilliste von X sein, da nur Elemente aus X gestrichen wurden, und eine Teilliste der sortierten Liste $X^{\leq}$, da $X^{\leq}$ aus Z nur durch Einfügungen entsteht. Wegen $Z \sqsubseteq X^{\leq}$ muß auch Z eine sortierte Liste sein.

Also ist Z eine Liste, die der Definition von $Rem(X)$ genügt, folglich gilt $Rem(X) \leq |X| - |Z|$. Andererseits kann man die Streichungen, deren Anzahl $Rem(X)$ bestimmen, um passende Einfügungen ergänzen, um $X^{\leq}$ zu erhalten, somit gilt auch $|X| - |Z| \leq Rem(X)$, zusammen also

$$Rem(X) = |X| - |Z|.$$

Beispiel: $(1,3,4,2,6,5)$ kann durch Streichen von 2 und 5 in $(1,3,4,6)$ überführt werden (Abbildung 4) oder auch durch Streichen von 2 und 6 in $(1,3,4,5)$, aber es genügt nicht, nur ein Element zu streichen, um eine sortierte Teilliste zu erhalten.

Damit gilt $Rem((1,3,4,2,6,5)) = 2$.

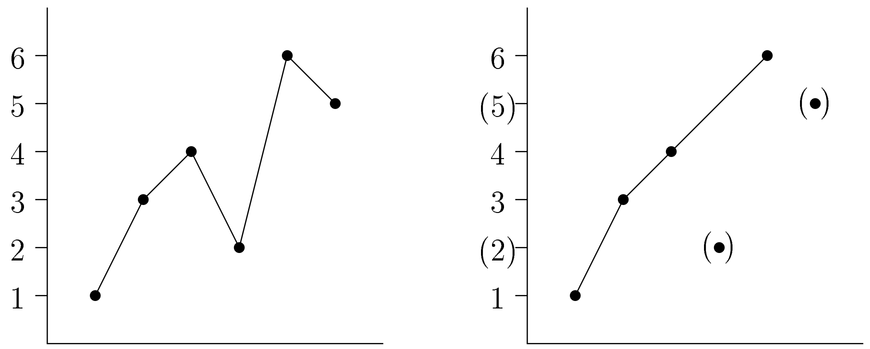


Abbildung 4: $X = (1,3,4,2,6,5)$, nach minimalem Streichen: $(1,3,4,6)$

Rem ist ein Maß:

1. Ist X sortiert, so ist offensichtlich $Rem(X) = 0$.
2. $X = (x_1, \ldots, x_n)$, $Y = (y_1, \ldots, y_n)$. Gilt $x_i < x_j$ genau dann, wenn $y_i < y_j$ gilt, so führen Streichungen in X genau dann zu einer sortierten Teilliste von X, wenn die Streichungen an den gleichen Positionen in Y zu einer sortierten Teilliste von Y führen, also $Rem(X) = Rem(Y)$.
3. Sei $X \sqsubseteq Y$. Erhält man eine sortierte Liste Z durch Streichungen in Y, so kann man durch höchstens ebensoviele Streichungen eine sortierte Teilliste W von X erhalten, indem man dieselben Elemente streicht, die in Y gestrichen wurden, außer den Elementen, die nicht in X liegen. Da nach der Konstruktionsvorschrift $W \sqsubseteq Z$ gilt, muß auch W sortiert sein. Folglich gilt $Rem(X) \leq Rem(Y)$.
4. Ist jedes Element der Liste X kleiner oder gleich jedem Element der Liste Y, so ist eine Teilliste von XY genau dann sortiert, wenn sie die Konkatenation zweier sortierter Teillisten von X und Y ist. Damit ist $Rem(XY) = Rem(X) + Rem(Y)$.
5. Sei $X = (x_1, \ldots, x_n)$. Ist $n = 0$, so ist $(a)X$ sortiert, und es gilt $Rem((a)X) = 0 = |X| + Rem(X)$.

   Andernfalls erhält man jede sortierte Teilliste von X durch eine zusätzliche Streichung (von a) auch in $(a)X$. Also gilt in diesem Fall
   $$Rem((a)X) \leq 1 + Rem(X) \leq |X| + Rem(X).$$

4.5 Das Exc-Maß

Sei $Exc(X)$ die Mindestanzahl der paarweisen Vertauschungen (engl. "exchanges") von Elementen, die nacheinander ausgeführt werden müssen, um $X = (x_1, \ldots, x_n)$ in eine sortierte Liste zu überführen. [Man85][EW89]

$\pi : \{1, \ldots, n\} \longrightarrow \{1, \ldots, n\}$ sei die Permutation, die X sortiert, genauer: sei π so gewählt, daß $X^{\leq} = (x_{\pi^{-1}(1)}, \ldots, x_{\pi^{-1}(n)})$ sortiert ist. (Siehe auch Abschnitt 2.2, insbesondere Gleichung (2.4))

$Exc(X)$ ist dann die minimale Anzahl von Transpositionen, deren Produkt π ist.

Man betrachte nun eine Zerlegung $\pi = \pi_1 \pi_2 \cdots \pi_k$ in disjunkte Zykel π_j der Längen c_j; dabei wird für jeden Fixpunkt $i = \pi(i)$ von π ein Zykel der Länge 1 gewählt. Somit gilt $\sum_{j=1}^{k} c_j = n$. Ein Zykel der Länge c_j kann als Produkt von $c_j - 1$, aber nicht weniger, Transpositionen ausgedrückt werden.

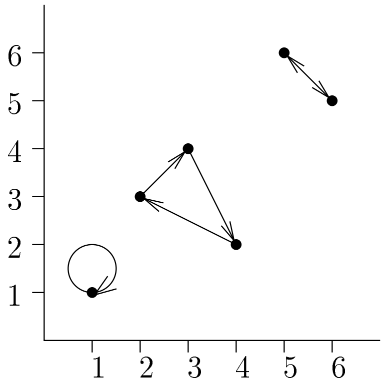
Folglich ist
$$Exc(X) = \sum_{j=1}^{k} (c_j - 1) = \sum_{j=1}^{k} c_j - k = n - k.$$

Beispiel: $X = (1, 3, 4, 2, 6, 5)$, $\pi = (1)(2\ 3\ 4)(5\ 6)$ mit der Zerlegung

$$\pi = \pi_1\pi_2\pi_3, \quad \pi_1 = (1), \quad \pi_2 = (2\ 3\ 4) = (2\ 3)(3\ 4), \quad \pi_3 = (5\ 6).$$

(Siehe Abbildung 5)

Also ist $Exc(X) = 6 - 3 = 3$ und $\pi = (2\ 3)(3\ 4)(5\ 6)$.



$$\begin{array}{l} 1 \longmapsto 1 \\ 2 \longmapsto 3 \\ 3 \longmapsto 4 \\ 4 \longmapsto 2 \\ 5 \longmapsto 6 \\ 6 \longmapsto 5 \end{array}$$

Abbildung 5: Links die Zykel der Permutation, die $(1, 3, 4, 2, 6, 5)$ sortiert, rechts die Permutation als Abbildungsvorschrift der Indizes

Exc ist ein Maß:

1. Ist $X = (x_1, \ldots, x_n)$ sortiert, zerfällt die zugehörige Permutation in n Zykel der Länge 1. Somit ist $Exc(X) = n - n = 0$.

2. $X = (x_1, \ldots, x_n)$, $Y = (y_1, \ldots, y_n)$. Gilt $x_i < x_j$ genau dann, wenn $y_i < y_j$ gilt, so werden X und Y durch dieselbe Index-Permutation in eine sortierte Liste $X^{\leq}$ bzw. $Y^{\leq}$ überführt, also gilt $Exc(X) = Exc(Y)$.

3. Seien $X = (x_1, \ldots, x_n)$, $Y = (y_1, \ldots, y_m)$ mit $X \sqsubseteq Y$ und sei

   $$f : \{1, \ldots, n\} \longrightarrow \{1, \ldots, m\}$$

   die Funktion, die die Elemente von X in die Elemente von Y abbildet: $y_{f(i)} = x_i$ für $i \in \{1, \ldots, n\}$. Zu einem beliebigen zu X gehörenden Zykel $(i_1\ i_2\ \cdots\ i_k)$ ist $(f(i_1)\ f(i_2)\ \cdots\ f(i_k))$ ein zu Y gehörender Zykel.

   Besteht die zu X gehörende Permutation aus j_X disjunkten Zykeln, so besteht die zu Y gehörende Permutation aus entsprechenden und höchstens $|Y| - |X| = m - n$ weiteren, die durch die in Y zusätzlich vorhandenen Elemente entstehen: $j_Y \leq j_X + (m - n)$. Es gilt also

   $$Exc(Y) = m - j_Y \geq m - (j_X + m - n) = n - j_X = Exc(X).$$

4. Ist jedes Element der Liste X kleiner oder gleich jedem Element der Liste Y, so ist die zu XY gehörende Permutation das Produkt der zu X gehörenden Permutationen und der zu Y gehörenden mit verschobenen Indizes:

   Zu jedem zu Y gehörenden Zykel $(i_1\ i_2\ \cdots\ i_k)$ gibt es einen Zykel $(i_1 + |X|\ i_2 + |X|\ \cdots\ i_k + |X|)$, der zu XY gehört. Seien $n = |X|$ und $m = |Y|$, und sei k die Anzahl der Zykel der zu X gehörenden Permutation und ℓ die der zu Y gehörenden. Damit gilt

$$\begin{aligned} Exc(XY) &= (n+m) - (k+\ell) = (n-k) + (m-\ell) \\ &= Exc(X) + Exc(Y). \end{aligned}$$

5. Sei $X = (x_1, \ldots, x_n)$ und sei $k = n - Exc(X)$ die Anzahl der disjunkten Zykel der zugehörigen Permutation. Die Zykelstruktur von $(a)X$ entspricht der von X ergänzt um einen Zykel der Länge 1. Damit gilt

$$Exc((a)X) = (n+1) - (k+1) = n - k = Exc(X) \leq |X| + Exc(X).$$

4.6 Das Par-Maß

Sei $Par(X) = \max(\{0\} \ \cup\ \{j - i \mid 1 \leq i < j \leq |X|,\ x_i > x_j\})$, das ist der maximale Abstand, den die Elemente einer Inversion in X voneinander haben. [EW89] (Dieses Maß ist aus einer Untersuchung paralleler Sortierverfahren abgeleitet, daher stammt der Name.)

Beispiel: $(1, 3, 4, 2, 6, 5)$ enthält die Inversionen $(3, 2)$ mit Abstand 2, $(4, 2)$ mit Abstand 1 und $(6, 5)$ mit Abstand 1 (Abbildung 6).

Folglich ist $Par((1, 3, 4, 2, 6, 5)) = 2$.

Par ist ein Maß:

1. Ist X sortiert, gilt $Par(X) = \max(\{0\} \ \cup\ \emptyset) = 0$.

2. $X = (x_1, \ldots, x_n)$, $Y = (y_1, \ldots, y_n)$. Gilt $x_i < x_j$ genau dann, wenn $y_i < y_j$ gilt, so haben die Inversionen in X dieselben Abstände wie in Y, also ist $Par(X) = Par(Y)$.

3. Sei $X \sqsubseteq Y$, dann enthält Y mindestens die Inversionen, die auch X enthält, mit eventuell größeren Abständen zwischen den beiden Elementen jeder Inversion. Damit gilt $Par(X) \leq Par(Y)$.

4. Ist jedes Element der Liste X kleiner oder gleich jedem Element der Liste Y, so liegt jede Inversion von XY entweder ganz in X oder ganz in Y. Folglich ist

$$Par(XY) = \max\{Par(X), Par(Y)\} \leq Par(X) + Par(Y).$$

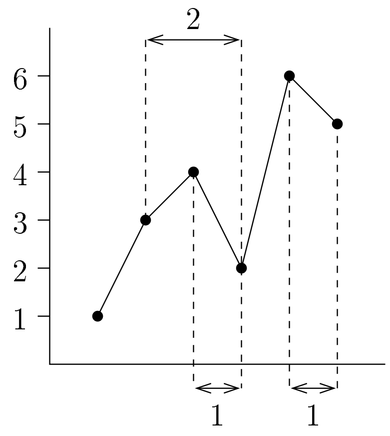


Abbildung 6: Die Abstände der Inversionen in $(1, 3, 4, 2, 6, 5)$

5. Keine zwei Elemente können in X einen Abstand größer als $|X| - 1$ haben, also gilt $Par(X) \leq |X| - 1$ und

$$Par((a)X) \leq |X| \leq |X| + Par(X).$$

4.7 Das Osc'-Maß

4.7.1 Osc ist kein Maß

Sei

$$Osc(X) = \sum_{i=1}^{n} \left|Cross_X(x_i)\right| \qquad \text{mit } n = |X| \tag{4.1}$$

und

$$Cross_X(x_i) = \{j \mid 1 \leq j < n,\ \min\{x_j, x_{j+1}\} < x_i < \max\{x_j, x_{j+1}\}\},$$

hierbei sind $Cross_X(x_i)$ anschaulich die Positionen der Schnittpunkte, die man erhält, wenn man die Liste X als einen Funktionsgraphen zeichnet und durch den Punkt (i, x_i) eine Waagerechte legt. [LP93]

Osc steht für "oscillation" (Oszillation) und soll andeuten, daß Osc beschreibt, wie stark der Funktionsgraph oszilliert.

Beispiel: $X = (1, 3, 4, 2, 6, 5)$, die Positionen, nach denen jeweils die gedachte

Linie gekreuzt wird, sind:

$$\begin{aligned}
Cross_X(x_1) &= Cross_X(1) = \emptyset, \\
Cross_X(x_2) &= Cross_X(3) = \{3,4\}, && \text{(Abbildung 7)} \\
Cross_X(x_3) &= Cross_X(4) = \{4\}, \\
Cross_X(x_4) &= Cross_X(2) = \{1\}, \\
Cross_X(x_5) &= Cross_X(6) = \emptyset, \\
Cross_X(x_6) &= Cross_X(5) = \{4\}. && \text{(Abbildung 7)}
\end{aligned}$$

Daraus ergibt sich

$$Osc(X) = \sum_{i=1}^{6} \left| Cross_X(x_i) \right| = 0+2+1+1+0+1 = 5. \quad \text{(Abbildung 8)}$$

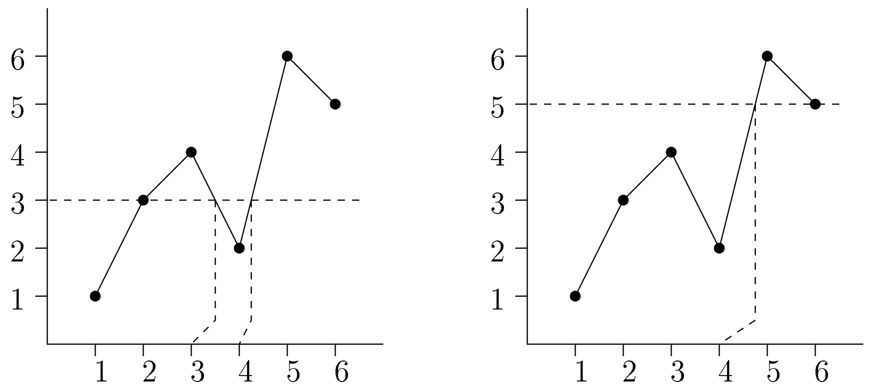


Abbildung 7: $X = (1,3,4,2,6,5)$, $Cross_X(3) = \{3,4\}$, $Cross_X(5) = \{4\}$

In [LP93] wird Osc als Maß im Sinne von [Man85] bezeichnet. Das ist allerdings in zweierlei Hinsicht nicht richtig, wie die folgenden Abschnitte zeigen.

4.7.2 Bedingung 4: Konkatenation zweier Listen

Zu Bedingung 4 betrachte man im bisher verwendeten Beispiel die Teillisten $Y_1 = (1,3,4,2)$ und $Y_2 = (6,5)$. Jedes Element in Y_1 ist kleiner als jedes Element in Y_2 und es gilt (siehe Abbildung 9)

$$Osc(Y_1) = 2, \quad Osc(Y_2) = 0,$$
$$Osc(X) = 5 > Osc(Y_1) + Osc(Y_2),$$

nicht wie Bedingung 4 verlangt,

$$Osc(X) \leq Osc(Y_1) + Osc(Y_2).$$

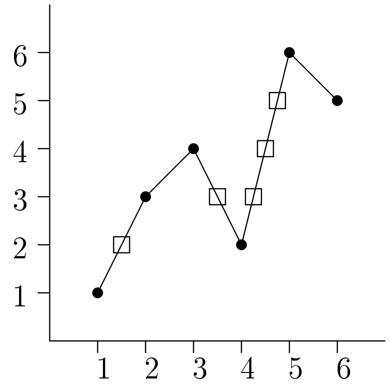


Abbildung 8: $Osc((1,3,4,2,6,5)) = 5$; die Schnittpunkte, die einen Beitrag leisten, sind durch Quadrate markiert.

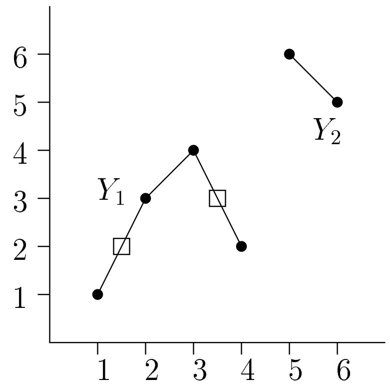


Abbildung 9: $X = Y_1Y_2$, $Osc(Y_1) = 2$, $Osc(Y_2) = 0$; die Schnittpunkte, die einen Beitrag leisten, sind durch Quadrate markiert.

Vergleicht man $Cross_X$, $Cross_{Y_1}$ und $Cross_{Y_2}$, so stellt man fest, daß der Index 4 dreimal als Element des Funktionswertes von $Cross_X$ auftritt (bei den Elementen 3, 4 und 5), aber weder bei $Cross_{Y_1}$ noch verschoben bei $Cross_{Y_2}$. Ansonsten stimmen die Funktionswerte überein (wäre $Cross_{Y_2}$ nicht immer leer, würden sich die Werte noch durch eine Indexverschiebung unterscheiden):

x_i	1	3	4	2	6	5
$Cross_X(x_i)$	$\emptyset$	$\{3,4\}$	$\{4\}$	$\{1\}$	$\emptyset$	$\{4\}$
$Cross_{Y_1}(x_i)$	$\emptyset$	$\{3\}$	$\emptyset$	$\{1\}$		
$Cross_{Y_2}(x_i)$					$\emptyset$	$\emptyset$

Dieser Beitrag von 3 fehlt also, um Bedingung 4 zu erfüllen.

Die Listenelemente 3, 4 und 5 sind genau diejenigen, deren Wert zwischen dem des letzten Elements von Y_1, 2, und dem des ersten Elements von Y_2, 6, liegen: Die Waagerechten durch 3, 4 und 5 schneiden die Verbindungslinie von 2 zu 6 in X, aber diese Verbindungslinie wird weder bei $Cross_{Y_1}$ noch bei $Cross_{Y_2}$ berücksichtigt, da sie ganz hinter dem letzten Element von Y_1 und ganz vor dem ersten Element von Y_2 liegt (Abbildungen 8 und 9).

Um Bedingung 4 zu erfüllen, muß man diese Schnittpunkte also berücksichtigen. Es ist naheliegend, die Schnittpunkte der Waagerechten durch 3 und 4 in $Cross_{Y_1}$ und die Schnittpunkte der Waagerechten durch 5 in $Cross_{Y_2}$ zu berücksichtigen, da 3 und 4 Elemente von Y_1 sind und 5 Element von Y_2 ist.

Dazu definiere man $Cross'_X$ wie folgt:

$$Cross'_X(x_i) = Cross_X(x_i) \cup \{0 \mid \text{falls } x_i < x_1\} \cup \{|X| \mid \text{falls } x_i > x_n\}. \tag{4.2}$$

Man beachte, daß 0 und $n = |X|$ niemals Elemente einer Menge $Cross_X(x_i)$ sind, und daß daher die Vereinigung disjunkt ist.

Ferner berücksichtigt diese Korrektur nicht nur die Verbindungslinie zwischen 2 und 6 im Beispiel, sondern auch eine Linie am Anfang der Liste Y_1 nach unten und eine am Ende der Liste Y_2 nach oben, denn auch da können ja weitere Listen angefügt werden. Dadurch ergeben sich folgende Unterschiede zwischen $Cross$ und $Cross'$:

x_i	1	3	4	2	6	5
$Cross_X(x_i)$	$\emptyset$	$\{3,4\}$	$\{4\}$	$\{1\}$	$\emptyset$	$\{4\}$
$Cross_{Y_1}(x_i)$	$\emptyset$	$\{3\}$	$\emptyset$	$\{1\}$		
$Cross_{Y_2}(x_i)$					$\emptyset$	$\emptyset$
$Cross'_X(x_i)$	$\emptyset$	$\{3,4\}$	$\{4\}$	$\{1\}$	$\{6\}$	$\{4\}$
$Cross'_{Y_1}(x_i)$	$\emptyset$	$\{3,4\}$	$\{4\}$	$\{1\}$		
$Cross'_{Y_2}(x_i)$					$\{2\}$	$\{0\}$

Die Elemente 6 und 5 liegen in Y_2 um vier Positionen weiter links als in X, dadurch sind die Werte in $Cross'_{Y_2}$ um 4 kleiner als die zugehörigen in $Cross'_X$.

Die Lage der bei $Cross'$ berücksichtigten Schnittpunkte ist noch einmal in den Abbildungen 10 und 11 dargestellt.

4.7.3 Bedingung 5: Verlängern einer Liste

Auch Bedingung 5,

$$M((a)X) \leq |X| + M(X),$$

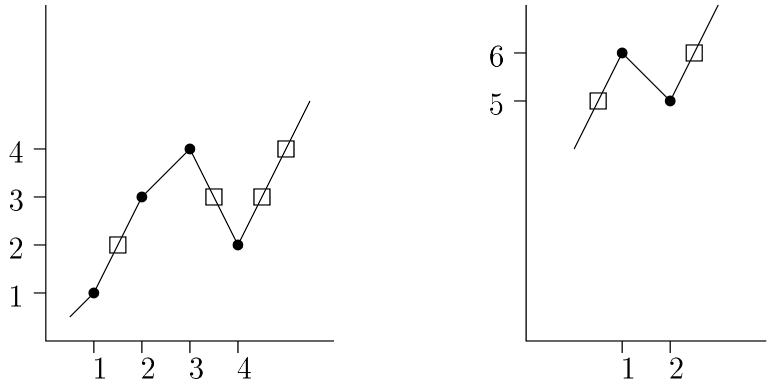


Abbildung 10: Links: $\sum |Cross'_{Y_1}| = 4$; rechts: $\sum |Cross'_{Y_2}| = 2$

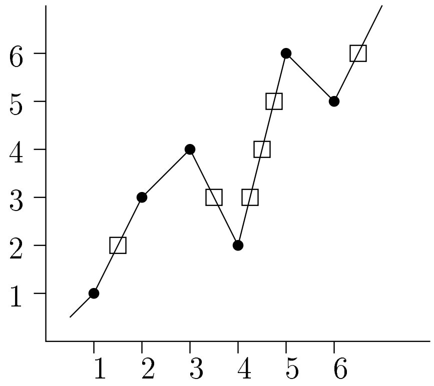


Abbildung 11: $\sum |Cross'_X| = 6 = \sum |Cross'_{Y_1}| + \sum |Cross'_{Y_2}|$

wird von Osc nicht erfüllt, wie dieses Gegenbeispiel zeigt (Abbildung 12):

$$L = (1, 5, 2, 6, 3), \quad Osc(L) = 6,$$

aber

$$Osc((4)L) = 12 > |L| + Osc(L) = 5 + 6 = 11.$$

Dieses Beispiel zeigt auch, daß $\sum |Cross'|$ kein Maß ist, denn

$$\sum \left|Cross'_L\right| = 8,$$
$$\sum \left|Cross'_{(4)L}\right| = 18 > |L| + \sum \left|Cross'_L\right| = 5 + 8 = 13.$$

$\sum |Cross'|$ wächst also beim Hinzufügen eines Elements zur Liste zu schnell, um Bedingung 5 zu erfüllen.

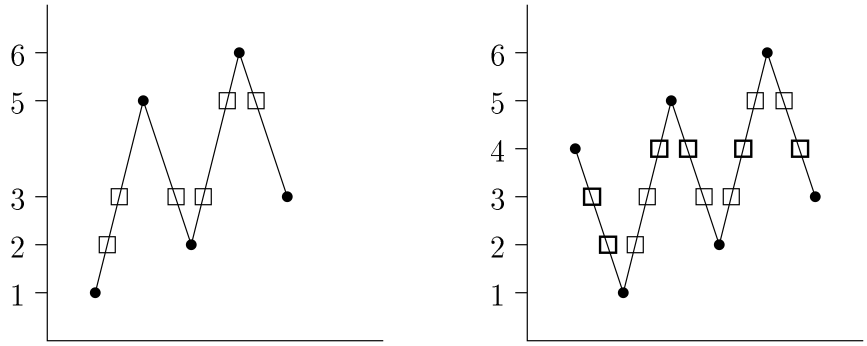


Abbildung 12: $Osc((1,5,2,6,3)) = 6$, $Osc((4,1,5,2,6,3)) = 12$

Eine Skalierung der Funktion mit einer positiven Konstante beeinflußt nicht die Bedingungen 1 bis 4 aus Kapitel 3, da in diesen nur Summen von Funktionswerten miteinander und mit Null verglichen werden. Insofern lohnt es sich zu untersuchen, ob eine Skalierung genügt, um Bedingung 5 zu erfüllen.

Nun ist aber eine Skalierung mit einer beliebigen reellen Konstante nicht möglich, da die Funktionswerte eines Maßes nach Voraussetzung immer natürliche Zahlen sein müssen. Folglich kann man nur mit dem Kehrwert eines gemeinsamen Teilers aller $\sum |Cross'|$ (oder eines Vielfachen von ihm) skalieren.

Folgende Werte von $\sum |Cross'|$ sind in der bisherigen Betrachtung aufgetreten:

$$\sum \left|Cross'_{Y_1}\right| = 4, \qquad \sum \left|Cross'_{Y_2}\right| = 2,$$
$$\sum \left|Cross'_{X}\right| = 6,$$
$$\sum \left|Cross'_{L}\right| = 8, \qquad \sum \left|Cross'_{(4)L}\right| = 18.$$

Diese Werte haben als einzigen nichttrivialen gemeinsamen Teiler die 2.

Eine nähere Untersuchung zeigt, daß $\sum |Cross'|$ tatsächlich immer gerade ist. Dazu betrachte man die möglichen Richtungen, aus denen ein durch alle Punkte (i, x_i) (mit $i \in \{1, \ldots, |X|\}$) laufender Funktionsgraph die x_i von links her erreicht und in die er sie nach rechts verläßt:

- Ist ein Listenelement x_i größer als seine beiden Nachbarn, oder liegt x_i am Listenanfang ($i = 1$) und ist größer als sein Nachbar x_{i+1}, so enthält $Cross'_X(x_i)$ eine gerade Anzahl (möglicherweise 0) Indizes kleiner als i und eine ungerade Anzahl Indizes größer als i:

  Der Funktionsgraph muß von unten links her entweder direkt durch (i, x_i) gehen, oder genausooft die Waagerechte durch (i, x_i) von unten her schneiden, wie er sie von oben her schneidet, da er x_i von unten erreicht – am linken Rand oder von x_{i-1} aus.

Von x_i aus verläuft der Funktionsgraph nach unten zu x_{i+1} und muß dann die Waagerechte einmal von unten her schneiden (und zusätzlich beliebig gleich oft von oben und unten), um am Ende der Liste nach rechts oben zu verlaufen.

$|Cross'_X(x_i)|$ ist in diesem Falle also ungerade.

- Ist ein Listenelement x_i kleiner als seine beiden Nachbarn, oder liegt x_i am Listenende ($i = |X|$) und ist kleiner als sein Nachbar x_{i-1}, so enthält analog $Cross'_X(x_i)$ eine ungerade Anzahl Indizes kleiner als i und eine gerade Anzahl Indizes größer als i.

  $|Cross'_X(x_i)|$ ist also auch in diesem Falle ungerade.

- Ist $x_{i-1} > x_i > x_{i+1}$, so kann man aus den vorigen beiden Fällen schließen, daß $Cross'_X(x_i)$ wegen $x_i < x_{i-1}$ eine ungerade Anzahl Indizes kleiner als i und wegen $x_i > x_{i+1}$ eine ungerade Anzahl Indizes größer als i enthält.

  $|Cross'_X(x_i)|$ ist dann also gerade und spielt für die Betrachtung, ob $\sum |Cross'_X|$ gerade ist, keine Rolle.

- Im letzten Fall (x_i ist in Bezug auf seine Nachbarn in sortierter Reihenfolge oder liegt an einem oder beiden Rändern) erreicht der Funktionsgraph x_i von unten und geht nach oben weiter, $Cross'_X(x_i)$ enthält also sowohl eine gerade Anzahl von Indizes kleiner als i als auch eine gerade Anzahl größer als i.

  $|Cross'_X(x_i)|$ ist also auch hier gerade und kann außer acht gelassen werden.

Die ersten beiden Fälle treten gleich oft auf, da der Funktionsgraph im ersten seine Richtung von einer aufsteigenden in eine absteigende und im zweiten Fall von einer absteigenden in eine aufsteigende ändert, und da er weiterhin am Anfang und am Ende der Liste eine aufsteigende Richtung hat und seine Richtung in den anderen beiden Fällen nicht ändert.

Weil die Fälle mit ungeradem $|Cross'_X|$ immer paarweise auftreten, ist also $\sum |Cross'_X|$ tatsächlich immer gerade.

Folglich ist $\frac{1}{2} \sum |Cross'|$ immer eine natürliche Zahl und gleichzeitig die kleinstmögliche Skalierung, da ja $\frac{1}{2} \sum |Cross'_{Y_2}|$ bereits 1 ist (Y_2 war als die Liste $(6, 5)$ definiert).

Diese Funktion wächst gerade langsam genug, um für die eingangs dieses Abschnittes definierte Liste $L = (1, 5, 2, 6, 3)$ und das zusätzliche Element 4 die Bedingung 5 zu erfüllen:

$$\begin{aligned}\frac{1}{2}\sum\left|Cross'_{(4)L}\right| &= \frac{1}{2}\cdot 18 = 9\\ &= |L| + \frac{1}{2}\sum\left|Cross'_{L}\right| = 5 + \frac{1}{2}\cdot 8 = 9.\end{aligned}$$

4.7.4 Osc' ist ein Maß

Die Überlegungen der vorigen Abschnitte legen

$$Osc'(X) = \frac{1}{2}\sum_{i=1}^{|X|}\left|Cross'_X(x_i)\right| \tag{4.3}$$

als Kandidaten für ein Maß nahe.

Tatsächlich ist Osc' ein Maß. Zum einen hat Osc' wie bereits gezeigt natürliche Funktionswerte, zum anderen erfüllt es auch die fünf Bedingungen, die an ein Maß gestellt werden:

1. Ist X sortiert, so liegt der Wert keines x_i zwischen benachbarten x_j und x_{j+1} und es gilt auch weder $x_i < x_1$ noch $x_i > x_n$, $Cross'_X(x_i)$ ist also leer für beliebiges i (siehe auch Definition (4.2)). Damit ist $Osc'(X) = 0$ wie gefordert.

2. $X = (x_1, \dots, x_n)$, $Y = (y_1, \dots, y_n)$. Gilt $x_i < x_j$ genau dann, wenn $y_i < y_j$ gilt, so gilt auch $\min\{x_j, x_{j+1}\} < x_i < \max\{x_j, x_{j+1}\}$ genau dann, wenn $\min\{y_j, y_{j+1}\} < y_i < \max\{y_j, y_{j+1}\}$ gilt. Auch gilt unter dieser Voraussetzung offensichtlich $x_i < x_1$ genau dann, wenn $y_i < y_1$ gilt, und $x_i > x_n$ genau dann, wenn $y_i > y_n$ gilt.

   Damit ist $Cross'_X(x_i) = Cross'_Y(y_i)$ (die Funktionswerte von $Cross'$ enthalten ja nicht die Elemente der Liste sondern ihre Indizes) und $Osc'(X) = Osc'(Y)$.

3. Seien $X = (x_1, \dots, x_n)$, $Y = (y_1, \dots, y_m)$ mit $X \sqsubseteq Y$ und sei

   $$f : \{1, \dots, n\} \longrightarrow \{1, \dots, m\}$$

   die Funktion, die die Elemente von X in die Elemente von Y abbildet: $y_{f(i)} = x_i$ für $i \in \{1, \dots, n\}$.

   - Enthält nun $Cross'_X(x_i)$ den Index $j \in \{1, \dots, n-1\}$, so hat das Element $x_i = y_{f(i)}$ einen Wert zwischen x_j und x_{j+1}:

     $$x_j = y_{f(j)} < x_i = y_{f(i)} < x_{j+1} = y_{f(j+1)}$$

oder

$$x_j = y_{f(j)} > x_i = y_{f(i)} > x_{j+1} = y_{f(j+1)}.$$

Die Elemente $y_{f(j)}$ und $y_{f(j+1)}$ müssen in Y nicht benachbart sein, aber da $y_{f(i)}$ einen Wert zwischen diesen beiden hat, muß es auch zwei benachbarte y_k, y_{k+1} mit $f(j) \leq k$ und $k{+}1 \leq f(j{+}1)$ geben, zwischen deren Wert $y_{f(i)}$ liegt. Damit ist also $k \in Cross'_Y(y_{f(i)})$.

- Enhält $Cross'_X(x_i)$ den Wert 0, so ist

$$x_i = y_{f(i)} < x_1 = y_{f(1)}.$$

Nun ist entweder $y_{f(i)} < y_1$, dann ist $0 \in Cross'_Y(y_{f(i)})$; oder es ist $y_1 < y_{f(i)} < y_{f(1)}$, dann gibt es benachbarte y_k, y_{k+1} mit $1 \leq k$ und $k+1 \leq f(1)$, zwischen deren Wert $y_{f(i)}$ liegt, und damit ist $k \in Cross'_Y(y_{f(i)})$.

- Enhält schließlich $Cross'_X(x_i)$ den Wert $|X| = n$, so gilt

$$x_i = y_{f(i)} > x_n = y_{f(n)}.$$

Es folgt analog zum vorigen Fall, daß entweder $y_{f(i)} > y_m$ ist, dann ist $|Y| = m \in Cross'_Y(y_{f(i)})$; oder es ist $y_{f(n)} < y_{f(i)} < y_m$, dann gibt es benachbarte y_k, y_{k+1} mit $f(n) \leq k$ und $k+1 \leq m$, zwischen deren Wert $y_{f(i)}$ liegt, und damit ist $k \in Cross'_Y(y_{f(i)})$.

Da die gefundenen Indizes im ersten Fall für $j \in Cross'_X(x_i)$ jeweils in $\{f(j), \ldots, f(j+1)-1\}$, im zweiten Fall in $\{0, \ldots, f(1)-1\}$ und im dritten Fall in $\{f(n), \ldots, m\}$ liegen, sind sie alle voneinander verschieden.

Folglich ist $|Cross'_Y(y_{f(i)})| \geq |Cross'_X(x_i)|$ für $i \in \{1, \ldots, n\}$ und

$$\begin{aligned} Osc'(Y) = \frac{1}{2}\sum_{j=1}^{m}\left|Cross'_Y(y_j)\right| &\geq \frac{1}{2}\sum_{i=1}^{n}\left|Cross'_Y(y_{f(i)})\right| \\ &\geq \frac{1}{2}\sum_{i=1}^{n}\left|Cross'_X(x_i)\right| = Osc'(X). \end{aligned}$$

4. Seien $X = (x_1, \ldots, x_n)$ und $Y = (y_1, \ldots, y_m)$, und sei $x_i \leq y_j$ für alle $i \in \{1, \ldots, n\}$ und $j \in \{1, \ldots, m\}$.

   Dann gilt $Cross'_{XY}(x_i) = Cross'_X(x_i)$ für $i \in \{1, \ldots, n\}$, da $x_i > x_n$ offensichtlich genau dann gilt, wenn $x_n < x_i < y_1$ ist, und da ferner kein x_i zwischen zwei y_j, y_{j+1} liegt.

Außerdem enthält $Cross'_{XY}(y_j)$ genau die Elemente von $Cross'_Y(y_i)$ um n erhöht für $j \in \{1, \dots, m\}$, da analog zum vorigen Fall $y_j < y_1$ genau dann gilt, wenn $x_n < y_j < y_1$ ist, und da ferner kein y_j zwischen zwei x_i, x_{i+1} liegt.

Damit ist

$$\begin{aligned} Osc'(XY) &= \frac{1}{2}\sum_{i=1}^{n}\left|Cross'_{XY}(x_i)\right| + \frac{1}{2}\sum_{j=1}^{m}\left|Cross'_{XY}(y_j)\right| \\ &= \frac{1}{2}\sum_{i=1}^{n}\left|Cross'_{X}(x_i)\right| + \frac{1}{2}\sum_{j=1}^{m}\left|Cross'_{Y}(y_j)\right| \\ &= Osc'(X) + Osc'(Y). \end{aligned}$$

5. Sei $X = (x_1, \dots, x_n)$ und a das Element, daß zu X hinzugefügt werden soll. Bezeichne k die Anzahl der Elemente in X, die kleiner sind als a und $n - k$ die Anzahl derer, die größer sind.

   a ist selbst das erste Element von $(a)X$, also $0 \notin Cross'_{(a)X}(a)$. Außerdem kann dadurch a nicht zwischen dem ersten und dem zweiten Element von $(a)X$ liegen, also $1 \notin Cross'_{(a)X}(a)$. Folglich enthält $Cross'_{(a)X}(a)$ höchstens die Indizes 2 bis $|(a)X| = n + 1$.

   Nur die Elemente $x_i > a$ liefern einen Beitrag – und zwar von höchstens zwei –, nämlich den Index $i - 1$, wenn $x_{i-1} < a < x_i$, und den Index i, wenn $x_i > a > x_{i+1}$ ist.

   Anders gezählt liefern nur die Elemente $x_j < a$ einen Beitrag – von höchstens zwei –, nämlich den Index $j - 1$, wenn $x_{j-1} > a > x_j$, und den Index j, wenn $x_j < a < x_{j+1}$ ist.

   Der mögliche Wert $n+1 \in Cross'_{(a)X}(a)$ ist hier noch nicht mitgezählt. Somit gilt

$$\begin{aligned} \left|Cross'_{(a)X}(a)\right| &\leq 2\min\{k, n-k\} + 1 \\ &\leq 2(n-k) + 1. \end{aligned}$$

   Die $Cross'_{(a)X}(x_i)$ enthalten gegenüber den $Cross'_X(x_i)$ um eins verschobene Indizes. Zusätzlich kommt für jedes $x_i < a$ noch der Index 0 und für $x_{i-1} > a > x_i$ auch der Index 1 dazu, letzteres ist allerdings für x_1 offensichtlich nicht möglich.

   Damit ist dieser zusätzliche Beitrag nicht größer als $2k - 1$.

Zusammen gilt:

$$\begin{aligned}
Osc'((a)X) &= \frac{1}{2}\Big(Cross'_{(a)X}(a) + \sum_{i=1}^{n}\big|Cross'_{(a)X}(x_i)\big|\Big) \\
&\leq \frac{1}{2}\Big(\sum_{i=1}^{n}\big|Cross'_X(x_i)\big| + \big(2(n-k)+1\big) + (2k-1)\Big) \\
&= \frac{1}{2}\Big(\sum_{i=1}^{n}\big|Cross'_X(x_i)\big| + 2n\Big) \\
&= |X| + Osc'(X).
\end{aligned}$$

Es bleibt zu klären, ob die Aussagen, die in [LP93] für Osc gezeigt werden, auch für Osc' gelten, insbesondere ob Adaptive Heapsort (siehe Kapitel 7.2) auch Osc'-optimal ist.

4.8 Weitere Maße

Die vorgestellen Maße sind keineswegs alle bekannten, sondern es ist eine Vielzahl weiterer, teilweise sehr spezialisierter Maße bekannt. Als Beispiele seien nur das Maß SMS in [MEP96] – es zählt disjunkte sortierte Teillisten – und das Maß *Block* in [CLP93] erwähnt – dieses zählt die zusammenhängenden Teillisten, die nach dem Sortieren auch wieder zusammenhängende Teillisten sind.

Inv, *Runs*, *Rem* und *Exc* sind die wohl offensichtlichsten Maße und werden auch in der Literatur immer wieder zu Vergleichen herangezogen. M_0 und M_{01} demonstrieren als triviale Beispiele, wie wenig ein Vorsortiertheitsmaß über eine Liste aussagen kann. *Par* und Osc' wurden als nicht so naheliegende Beispiele gewählt, Osc' ist gleichzeitig ein schon von der Definition her recht komplexes Beispiel.

5 Optimalität von Algorithmen

Auch in diesem Kapitel seien der Einfachheit halber mehrere gleiche Elemente in einer Liste nicht zugelassen.

Läßt man gleiche Elemente zu, so benötigt man Vergleichsfunktionen mit drei möglichen Ergebnissen: „$a < b$“, „$a = b$“ und „$a > b$“. Das führt vor allem dazu, daß durch einen Vergleich mehr Information (um einen konstanten Faktor) gewonnen wird.

Außerdem erhöht sich die Anzahl der unterscheidbaren Listen der Länge n über $n!$ hinaus. Gleichzeitig kann ein Algorithmus aber nicht zwischen mehr als $n!$ Permutationen wählen, so daß wesentlich verschiedene Listen durch dieselbe Permutation sortiert werden können: Die Listen $(2, 3, 1)$ und $(1, 2, 1)$ werden beide durch die Permutation $(1\ 2\ 3)$ (die wie in Abschnitt 2.2 auf die Indizes anzuwenden ist) sortiert, sind aber durch einen Vergleich des ersten Elements mit dem letzten unterscheidbar.

5.1 Optimales nicht-adaptives Sortieren

Um eine beliebige n-elementige Liste X zu sortieren, muß der verwendete Sortieralgorithmus unterscheiden, welche Permutationsfunktion π der $n!$ möglichen die Liste sortiert, bzw. äquivalent, welche Permutation X der sortierten Liste $X^{\leq}$ vorliegt:

$$X = \pi^{-1}(X^{\leq}). \tag{5.1}$$

Informationen über die gesuchte Permutation kann der Algorithmus bei vergleichsbasiertem Sortieren nur durch sukzesssives Vergleichen von jeweils zwei Elementen erhalten. Operationen auf anderen Daten, wie etwa Indexwerten, bearbeiten die bereits gewonnene Information nur, sie führen zu keiner neuen Information.

Da ein Vergleich nur zwei mögliche Ergebnisse haben kann („$a < b$“ oder „$a > b$“, siehe oben), erhält der Algorithmus durch einen Vergleich höchstens ein Bit an Information. Ein Vergleich kann auch weniger Information liefern: Haben die Vergleiche der Elemente a und b und der Elemente b und c beispielsweise die Ergebnisse $a < b$ und $b < c$ geliefert, so gewinnt man durch einen Vergleich von a mit c keine neue Information.

Führt ein Sortieralgorithmus bei gegebenem n nicht mehr als k_n Vergleiche aus, so gewinnt er also höchstens k_n Bit an Information und kann damit nicht mehr als 2^{k_n} Permutationen unterscheiden.

Der Algorithmus muß aber zwischen den $n!$ möglichen Permutationen unterscheiden, von denen nur eine die gegebene Liste der Länge n sortiert.

Folglich muß für beliebiges n

$$2^{k_n} \geq n!, \tag{5.2}$$

also

$$k_n \geq \log_2 n!, \qquad k_n = \Omega(\log n!) \tag{5.3}$$

gelten.

Die obere Schranke für die Anzahl der Vergleiche kann also bei gegebenem n nicht $\log_2 n!$ unterschreiten. $\Omega(\log n!)$ läßt sich nach der Stirlingschen Formel ($n! \sim \sqrt{2\pi n} \cdot (\frac{n}{e})^n$) durch $\Omega(n \log n)$ abschätzen. Damit erhält man die bekannte Komplexität des vergleichsbasierten Sortierproblems.

Es sind Algorithmen bekannt, die nur $O(n \log n)$ Vergleiche benötigen, die Schranke ist also scharf. Beispiele sind Heapsort und Mergesort, Heapsort kommt außerdem mit $O(1)$ zusätzlichem Speicher aus.

5.2 Adaptives Sortieren

Der Nutzen der Vorsortiertheitsmaße liegt nun darin, daß man diese informationstheoretische untere Schranke für die Anzahl der benötigten Operationen (Vergleiche und Vertauschungen) abschwächen kann, indem man außer $n = |X|$ einen weiteren Parameter zuläßt: den Wert des Maßes M, also $M(X)$.

5.2.1 Schwache M-Optimalität

Der erste Schritt besteht darin, die Anzahl der Vergleiche nicht nur in Abhängigkeit von n, sondern zusätzlich von einer oberen Schranke z des Maßes zu untersuchen:

$$z \geq M(X). \tag{5.4}$$

Die Anzahl der Vergleiche bei einem festen Maß M heiße $k_{n,z}$.

Ist diese Schranke gegeben, so muß der Algorithmus nicht mehr alle $n!$ Permutationen der sortierten Liste $X^{\leq}$ unterscheiden, sondern nur die, deren Maß durch z beschränkt ist. Die Menge dieser Listen heiße

$$b'(z, X, M) = \{Y \mid Y \text{ ist Permutation von } X \text{ und } M(Y) \leq z\} \tag{5.5}$$

(siehe [Man85]).

Man beachte dabei, daß die Anzahl $|b'(z, X, M)|$ nicht von den Elementen der Liste X sondern nur von ihrer Länge abhängt, da zum einen der Wert eines Maßes nicht von den Elementen abhängt und zum anderen „Permutation von X“ und „Permutation von $X^{\leq}$“ äquivalent sind, die Anzahl also auch nicht von der Reihenfolge der Elemente in X abhängt.

Mit den Parametern n und z (und dem Maß M) gilt damit

$$2^{k_{n,z}} \geq b'(z, X, M) \qquad (n = |X|). \tag{5.6}$$

Ein beliebiges Sortierverfahren benötigt also im "worst case" mindestens $\Omega(\log |b'(z, X, M)|)$ Vergleiche, um eine Liste X mit $M(X) \leq z$ zu sortieren.

Verwendet man diese Schranke direkt zur Definition der Optimalität, so ergibt sich eine Komplikation, dadurch daß $\log |b'(z, X, M)|$ nicht notwendig bei jedem Maß durch $\Omega(n)$ von unten beschränkt ist. Es sind aber mindestens $n - 1$ Vergleiche nötig, um festzustellen, daß eine gegebene Liste bereits sortiert ist, oder beispielsweise bei lokaler Unordnung festzustellen, wo sie sich in der Liste befindet.

Diese Information kann im allgemeinen bei lokaler Unordnung – einer Unordnung, die sich durch das Sortieren weniger Elemente eliminieren läßt, ohne daß die weiter links oder rechts liegenden Elemente bewegt werden müssen – auch nicht indirekt aus z gewonnen werden, da z dafür bereits Information über die Position der Unordnung in der Liste enthalten müßte.

Eine solche Positionsabhängigkeit wird durch Bedingung 4 aus Kapitel 3 verhindert: Man betrachte eine Liste mit lokaler Unordnung und füge am Anfang Elemente an, die kleiner als das Minimum der ursprünglichen Liste sind. Bedingung 4 verlangt, daß sich dadurch das Maß der Liste nicht vergrößert. Ebenso kann man am Ende der Liste größere Elemente anhängen, ohne das Maß zu vergrößern.

Soll das Maß die Position der lokalen Unordnung angeben, so muß es sich in diesen beiden Fällen unterschiedlich verhalten. Es kann aber nur endlich oft kleiner werden, da die Werte eines Maßes natürliche Zahlen sind. Dadurch kann nach hinreichend vielen Schritten die Wertdifferenz des Maßes nicht mehr angeben, ob die Unordnung nach vorne oder nach hinten verschoben wird.

Daher läßt man eine lineare Anzahl von Operationen unabhängig von der Vorsortiertheit zu und verwendet als Schranke

$$O\big(\max\{n, \log |b'(z, X, M)|\}\big), \tag{5.7}$$

bzw. äquivalent (siehe Gleichung 2.10)

$$O\big(n + \log |b'(z, X, M)|\big). \tag{5.8}$$

Sortiert ein Algorithmus bei Eingabe von z und X (und damit auch der Eingabe von $n = |X|$) die Liste X mit $O(n + \log |b'(z, X, M)|)$ Operationen, so heiße er *schwach M-optimal*.

5.2.2 M-Optimalität

Da oft keine Schranke $z \geq M(X)$ bekannt ist und ihre Berechnung durchaus mehr Operationen brauchen kann als das anschließende Sortieren, ist es wünschenswert, die Schranke implizit zu lassen. Dafür definiert man nach [Man85]

$$b(X, M) = b'(M(X), X, M). \tag{5.9}$$

$|b(X, M)|$ ist die Anzahl der Permutationen von X, die bezüglich des Maßes M mindestens so gut sortiert sind wie X.

Auch $|b(X, M)|$ hängt nicht direkt von den Elementen von X ab, sondern nur von der Länge $n = |X|$ und der Vorsortiertheit $M(X)$.

Analog zur Überlegung in 5.2.1 benötigt ein Sortierverfahren, dem eine beliebige Liste X mit bekannter Länge n und fester aber unbekannter Vorsortiertheit $M(X)$ als Eingabe gegeben wird, im "worst case" mindestens $\Omega(\log |b(X, M)|)$ Vergleiche.

Ein Algorithmus, der eine Liste X der Länge n als Eingabe bekommt und sie mit

$$O\big(n + \log |b(X, M)|\big) \tag{5.10}$$

Operationen sortiert, heiße optimal bezüglich des Maßes M, kurz *M-optimal*.

Da sich die Laufzeit eines M-optimalen Algorithmus (im "worst case") für jede Eingabeliste X an das Maß M anpaßt, wird ein solcher Algorithmus auch adaptiv (bezüglich des Maßes M) genannt.

Ein M-optimaler Algorithmus kann in der Praxis als Ersatz für einen nicht-adaptiven verwendet werden, da er keine zusätzlichen Eingaben benötigt. Ein schwach M-optimaler Algorithmus kann dagegen nur eingesetzt werden, wenn eine Schranke für die Vorsortiertheit bekannt ist oder mit akzeptablem Aufwand berechnet werden kann.

6 Wertebereiche der Vorsortiertheitsmaße

Will man für einen gegebenen Algorithmus zeigen, daß er bezüglich eines bestimmten Maßes M optimal ist, oder einen Algorithmus mit diesem Ziel entwerfen, so ist es hinderlich, die Menge der Listen $b(X, M)$ direkt zu betrachten.

Stattdessen ist es wünschenswert, für ein Maß M eine derartige Funktion f_M zu finden, daß $b(X, M)$ durch $f_M(M(X))$ von unten beschränkt ist. Braucht ein Algorithmus für eine solche Funktion f_M zum Sortieren einer Liste X (mit $n = |X|$) höchstens $O\big(n + \log f_M(M(X))\big)$ Operationen, so benötigt er wegen

$$f_M(M(X)) \leq |b(X, M)| \tag{6.1}$$

nicht mehr als $O(n + \log |b(X, M)|)$ Operationen und ist M-optimal.

f_M kann aber nicht beliebig klein gewählt werden, da sonst kein Algorithmus mit $O\big(n+\log f_M(M(X))\big)$ Operationen auskommen kann. Insbesondere muß jeder Sortieralgorithmus für mindestens eine Liste $\Omega(n \log n)$ Operationen brauchen, weil andernfalls die Komplexität des Sortierens im Widerspruch zu Abschnitt 5.1 nicht durch $\Omega(n \log n)$ von unten beschränkt wäre.

Das Maximum von $f_M(M(X))$ (über alle Listen X einer gegebenen Länge n) muß dafür mindestens die Größenordnung von $n!$ erreichen. Andererseits gilt

$$|b(X, M)| \leq n! \qquad (\text{mit } n = |X|), \tag{6.2}$$

da $b(X, M)$ nur Permutationen der n-elementigen Liste X enthält, zusammen mit (6.1):

$$f_M(M(X)) \leq n! \qquad (n = |X|). \tag{6.3}$$

$b(X, M)$ hängt wegen (5.5) und (5.9) monoton von $M(X)$ ab, daher bietet es sich an, auch f_M monoton zu wählen, um $f_M(M(X))$ gut an $b(X, M)$ anzunähern. $f_M(M(X))$ nimmt sein Maximum dann für maximales $M(X)$ und sein Minimum für minimales $M(X)$ an.

Da der Wahl von f_M – wie oben gezeigt – enge Grenzen gesetzt sind, ist es zunächst notwendig, die Wertebereiche der Maße zu untersuchen, um dann geeignete f_M auswählen zu können.

Um den Wertebereich eines Maßes nach oben abzuschätzen, beachte man zuerst, daß die leere Liste und jede einelementige Liste sortiert sind. Nach Bedingung 1 aus Kapitel 3 ist ihr Maß daher Null. Zusammen mit Bedingung 5,

$$M((a)X) \leq |X| + M(X), \tag{6.4}$$

bedingt dies, daß kein Maß schneller als quadratisch wachsen kann:

$$M(X) \leq \sum_{i=0}^{n-1} i = \frac{1}{2}n(n-1) \qquad (\text{mit } n = |X|). \tag{6.5}$$

Weiterhin ist nach Bedingung 3 das Maß einer Liste mindestens so groß wie das einer beliebigen Teilliste. Insbesondere ist die leere Liste () eine Teilliste jeder Liste X und hat Maß Null. Somit gilt:

$$() \sqsubseteq X, \quad M(X) \geq M(()) = 0. \tag{6.6}$$

Diese untere Schranke wird wegen $M(()) = 0$ von jedem Maß auch erreicht. Die obere Schranke (6.5) wird nicht von jedem Maß erreicht, aber zumindest von einem, nämlich *Inv* (siehe Abschnitt 6.1.2).

6.1 Maxima einzelner Maße

Einige der Maxima sind offensichtlich, andere – insbesondere zu *Inv* und *Exc* – sind beispielsweise in [Man85] angegeben. Eine gemeinsame Darstellung in Hinblick auf die Wahl der f_M findet sich in der Literatur jedoch nicht.

6.1.1 M_0 und M_{01}

M_0 hat immer den Wert 0, also ist auch das Maximum 0.

M_{01} hat für jede sortierte Liste den Wert 0 und für jede andere den Wert 1. Für $n \geq 2$ wird das Maximum z. B. durch die absteigende Liste $(n, \ldots, 1)$ erreicht:

$$M_{01}((n, \ldots, 1)) = 1 \qquad (n \geq 2). \tag{6.7}$$

6.1.2 *Inv*

Es gibt $\binom{n}{2} = \frac{1}{2}n(n-1)$ Möglichkeiten, ein Paar aus den Elementen einer Liste der Länge n zu wählen. Für die absteigende Liste $(n, \ldots, 1)$ ist jedes dieser Paare eine Inversion. Damit wird *Inv* für diese Liste bei gegebenem n maximal und es gilt:

$$Inv((n, \ldots, 1)) = \frac{1}{2}n(n-1). \tag{6.8}$$

Gleichzeitig ist diese auch für gegebenes n im wesentlichen die einzige Liste, für die *Inv* maximal wird, da jede nicht absteigend sortierte Liste mindestens ein Paar enthält, das keine Inversion ist.

Weitere Listen mit maximalem Inv erhält man für festes n nur, indem man andere Elemente als 1 bis n verwendet und diese absteigend sortiert in eine Liste fügt. Eine solche Liste entspricht der Liste $(n, \dots, 1)$ in der Weise, daß sich beim Vergleichen die Elemente an den entsprechenden Positionen gleich verhalten.

Sucht man für gegebenes n nach einer Liste X, die eine Permutation der Liste $(1, \dots, n)$ ist, und für die $Inv(X)$ maximal wird, so erhält man als eindeutige Lösung

$$X = (n, \dots, 1). \tag{6.9}$$

6.1.3 *Runs*

Eine Liste der Länge n (mit $n \geq 1$) enthält $n - 1$ Paare mit in der Liste benachbarten Elementen. In der absteigenden Liste $(n, \dots, 1)$ sind alle diese Paare Inversionen. Folglich wird $Runs$ – das die Anzahl der Inversionen unter diesen Paaren zählt – für diese Liste maximal:

$$Runs((n, \dots, 1)) = n - 1 \qquad (n \geq 1). \tag{6.10}$$

Diese Liste ist im wesentlichen für ein gegebenes n die einzige Lösung, jede Liste, die nicht absteigend sortiert ist, mindestens ein aufsteigendes Paar benachbarter Elemente enthalten muß.

6.1.4 *Rem*

Entfernt man aus einer n-elementigen Liste (mit $n \geq 1$) beliebige $n - 1$ Elemente, so erhält man eine einelementige sortierte Liste, also:

$$Rem(X) \leq n - 1 \qquad (n = |X| \geq 1). \tag{6.11}$$

Jede zweielementige Teilliste der absteigenden Liste $(n, \dots, 1)$ ist selbst absteigend, also nicht sortiert. Es genügt also nicht, nur $n - 2$ Elemente zu entfernen, um eine sortierte Teilliste von $(n, \dots, 1)$ zu erhalten. Folglich ist

$$Rem((n, \dots, 1)) = n - 1 \qquad (n \geq 1). \tag{6.12}$$

Auch für Rem ist $(n, \dots, 1)$ für gegebenes n im wesentlichen die einzige Liste, bei der das Maximum erreicht wird: Jede Liste X, die nicht absteigend sortiert ist, enthält mindestens ein aufsteigend sortiertes Paar. Dieses Paar ist eine zweielementige sortierte Teilliste von X und es gilt somit

$$Rem(X) \leq n - 2 \qquad (n = |X|, \quad X \text{ nicht absteigend}). \tag{6.13}$$

6.1.5 *Exc*

Zerfällt die Permutationsfunktion, die eine Liste X sortiert, in k disjunkte Zykel, so ist $Exc(X) = |X| - k$. Für $n = |X| \geq 1$ muß k mindestens 1 sein – dann besteht die Permutation aus nur einem Zykel und $Exc(X)$ ist $n-1$. Eine solche Liste X mit der zugehörigen Permutation π ist z. B.

$$X = (n, 1, 2, 3, \dots, n-1), \quad \pi = (1 \ n \ n-1 \ n-2 \ \cdots \ 2). \tag{6.14}$$

Für gegebenes $n > 2$ gibt es jeweils wesentlich verschiedene Listen, für die Exc maximal wird, da es für jedes $n > 2$ mehr als einen Zykel der Länge n gibt und jede Permutationsfunktion τ zu einer anderen Liste X_τ gehört, die durch sie sortiert wird:

$$X_\tau = \tau^{-1}(X^{\leq}). \tag{6.15}$$

6.1.6 *Par*

$Par(X)$ ist der maximale Abstand, den die Elemente einer Inversion in X voneinander haben. Seien $X = (x_1, \dots, x_n)$ und die fragliche Inversion $(x_i, x_j) \sqsubseteq X$ mit $1 \leq i < j \leq n, x_i > x_j$. Dann ist $Par(X) = j - i \geq 1$. Hat X dagegen keine Inversion, so ist X sortiert, und $Par(X)$ ist 0, also für $n > 1$ nicht maximal.

Das Maximum erreicht $j - i$ offensichtlich für

$$i = 1, \quad j = n, \quad Par(X) = j - i = n - 1. \tag{6.16}$$

Die einzige Bedingung, die dabei an X gestellt wird, ist $x_1 > x_n$. Sie wird z. B. durch

$$X = (n, \dots, 1) \tag{6.17}$$

erfüllt.

Für festes $n > 2$ ist $(n, \dots, 1)$ nicht die einzige Liste, für die Par maximal wird.

6.1.7 *Osc'*

Es ist dem Maß Osc' nicht leicht anzusehen, für welche Listen es maximal wird. Sicher müssen sich kleine und große Werte abwechseln, um möglichst viele Schnittpunkte zu erzeugen.

Um die Struktur der gesuchten Listen im Detail herauszufinden, bietet es sich an, für geringe Längen ein Programm (Anhang A.1.2) die Listen suchen zu lassen.

Für $n \leq 12$ ergeben sich folgende Werte für die Maxima m_n und Häufigkeiten h_n des Auftretens:

n	1	2	3	4	5	6	7	8	9	10	11	12
m_n	0	1	2	4	6	9	12	16	20	25	30	36
h_n	1	1	3	4	20	36	252	576	5184	14400	158400	518400

Betrachtet man die Listen, bei denen die Maxima auftreten (siehe Anhang B.2), so stellt man fest, daß die auftretenden Listen bei geradem n abwechselnd aus Elementen $x_{2i-1} \geq \frac{n}{2}$ und $x_{2i} < \frac{n}{2}$ mit $i \in \{1, \dots, \frac{n}{2}\}$ bestehen, und daß für alle derartigen Listen Osc' auch maximal wird.

Bei ungeradem n treten abwechselnd Elemente größer als $\frac{n+1}{2}$ und kleiner als $\frac{n+1}{2}$ auf, mit dem Element $\frac{n+1}{2}$ an beliebiger Stelle eingefügt. Auch bei ungeradem n wird Osc' für alle so aufgebauten Listen maximal.

Zerlegt man nun obenstehende Tabelle in eine mit ungeraden und eine mit geraden n und faktorisiert die auftretenden h_n, so erhält man:

n	1	3	5	7	9	11
m_n	0	2	6	12	20	30
h_n	1	3	$2^2 \cdot 5$	$2^2 \cdot 3^2 \cdot 7$	$2^6 \cdot 3^4$	$2^6 \cdot 3^2 \cdot 5^2 \cdot 11$

n	2	4	6	8	10	12
m_n	1	4	9	16	25	36
h_n	1	2^2	$2^2 \cdot 3^2$	$2^6 \cdot 3^2$	$2^6 \cdot 3^2 \cdot 5^2$	$2^8 \cdot 3^4 \cdot 5^2$

An diesen Tabellen fällt auf, daß die h_{n+2} stark von den h_n abhängen. Bei geradem n sind sie jeweils ein Vielfaches, bei ungeradem n gilt dasselbe für h_n/n. Formt man die Zahlen entsprechen um, wird der genaue Zusammenhang sichtbar:

n	1	3	5	7	9	11
m_n	0	2	$6 = 2 \cdot 3$	$12 = 3 \cdot 4$	$20 = 4 \cdot 5$	$30 = 5 \cdot 6$
h_n/n	1	1	4	$4 \cdot 9$	$4 \cdot 9 \cdot 16$	$4 \cdot 9 \cdot 16 \cdot 25$

n	2	4	6	8	10	12
m_n	1	4	9	16	25	36
h_n	1	4	$4 \cdot 9$	$4 \cdot 9 \cdot 16$	$4 \cdot 9 \cdot 16 \cdot 25$	$4 \cdot 9 \cdot 16 \cdot 25 \cdot 36$

Bei den geraden n gilt also

$$m_n = \frac{1}{4} n^2, \tag{6.18}$$

$$h_n = \left(\frac{n}{2}!\right)^2, \tag{6.19}$$

bei den ungeraden

$$m_n = \frac{1}{4}(n-1)(n+1), \tag{6.20}$$

$$h_n = n\Big(\frac{n-1}{2}!\Big)^2. \tag{6.21}$$

Zusammen ergibt sich:

$$Osc'(X) \leq \Big\lfloor \frac{n}{2} \Big\rfloor \cdot \Big\lceil \frac{n}{2} \Big\rceil \qquad (n = |X| \leq 12). \tag{6.22}$$

Wie man in den Tabellen sieht, sind die Listen, für die Osc' maximal wird, für $n > 2$ nicht durch n und die Menge ihrer Elemente eindeutig bestimmt.

Es ist offen, ob die Maxima sich auch für $n > 12$ in der angegebenen Weise verhalten. Die einfache Form der Gleichungen (6.18) bis (6.21) legt diese Vermutung aber nahe.

6.2 Zusammenfassung

Bei vorgegebener Listenlänge n – dabei sei $n \geq 2$, um Spezialfälle auszuschließen – erreichen die Maße folgende Maxima:

M	X	$M(X)$	
M_0	$(1, \dots, n)$	0	
M_{01}	$(n, \dots, 1)$	1	(6.7)
Inv	$(n, \dots, 1)$	$\frac{1}{2}n(n-1)$	(6.8)
$Runs$	$(n, \dots, 1)$	$n-1$	(6.10)
Rem	$(n, \dots, 1)$	$n-1$	(6.12)
Exc	$(n, 1, 2, 3, \dots, n-1)$	$n-1$	(6.14)
Par	$(n, \dots, 1)$	$n-1$	(6.17)
Osc'	$(\frac{n}{2}+1, 1, \frac{n}{2}+2, 2, \dots, n, \frac{n}{2})$	$\lfloor\frac{n}{2}\rfloor \cdot \lceil\frac{n}{2}\rceil$	(6.22)

Nur für Inv, $Runs$ und Rem sind die angegebenen Listen als Permutationen von $(1, \dots, n)$ mit festem n, für die das jeweilige Maß maximal wird, eindeutig bestimmt. Für Osc' ist nur ein Beispiel mit geradem $n \leq 12$ angegeben.

Die Bedeutung der Maxima ergibt sich auch aus folgender Überlegung: Das Maximum eines Maßes M schränkt einen M-optimalen Algorithmus am wenigsten ein. Hat eine intuitiv betrachtet stark vorsortierte Liste eine Struktur, für die das Maß maximal wird, so ist nicht gesichert, daß der Algorithmus die Liste schnell sortieren kann.

Alle Listen in der obenstehenden Tabelle haben aber eine extrem regelmäßige Struktur und können sicher nicht als völlig unsortiert angesehen werden.

In vielen Fällen werden sie ja von jeweils anderen Maßen auch als stark vorsortiert erkannt.

Daher könnte es nützlich sein, bei der Suche nach neuen Maßen besonders auf die Schwächen der vorgestellten Maße zu achten, und nach Maßen zu forschen, die nicht für einfach strukturierte, regelmäßig aufgebaute Listen maximal werden.

7 Adaptive Sortierverfahren

7.1 Die Maße M_0 und M_{01}

Da M_0 konstant ist, hängt $|b(X, M_0)|$ nicht von den Elementen von X ab, sondern ist immer $n!$ (mit $n = |X|$). Unter Anwendung der Stirlingschen Formel erhält man

$$\begin{aligned} O(n \log n) &= O(n + n \log n) = O(n + \log n!) \\ &= O\big(n + \log |b(X, M_0)|\big). \end{aligned} \tag{7.1}$$

Dadurch erfüllt jeder Algorithmus, der mit $O(n \log n)$ Operationen auskommt, also jeder, der die klassische Schranke einhält, Bedingung (5.10) und ist M_0-optimal. Das sind beispielsweise Heapsort und Mergesort, nicht jedoch Quicksort – Quicksort kommt nur im Durchschnitt mit $O(n \log n)$ Operationen aus, nicht im ungünstigsten Fall (siehe [Knu73, 5.2.2, S. 122]).

Auch M_{01}-Optimalität ist recht einfach zu erreichen: Für die sortierte Liste $X^{\leq}$ enthält $b(X^{\leq}, M_{01})$ nur $X^{\leq}$ selbst und es ist $|b(X^{\leq}, M_{01})| = 1$. Für alle anderen Listen ist wie oben $|b(X, M_{01})| = n!$ (mit $n = |X|$).

Ein M_{01}-optimaler Algorithmus darf also für eine n-elementige sortierte Liste $O(n + \log 1) = O(n)$ Operationen und für eine unsortierte $O(n \log n)$ Operationen ausführen. Ein Algorithmus, der zuerst prüft, ob X bereits sortiert ist, und danach im anderen Fall mit $O(|X| \log |X|)$ Operationen sortiert, leistet dies.

7.2 Beispiele optimaler Algorithmen

Zu den anderen in Kapitel 4 beschriebenen Maßen sind auch optimale Algorithmen bekannt:

Local Insertion Sort [Man85] ist *Rem*-optimal [Man85, Theorem 8] und damit nach [CLP93, Theorem 23] auch *Exc*-optimal.

Blocksort [LP96] ist *Inv*-optimal.

Weitere Algorithmen werden im folgenden beschrieben.

In diesem Abschnitt sei jeweils $n = |X|$.

7.2.1 Natural Mergesort

Natural Mergesort [Knu73, 5.2.4] teilt zuerst die Eingabeliste in aufsteigende "runs", also maximale zusammenhängende sortierte Teillisten.

Dann werden jeweils paarweise aufeinanderfolgende Teillisten miteinander verschmolzen, so daß halbsoviele "runs" übrigbleiben. Diese Operation wird solange wiederholt, bis die gesamte Liste sortiert ist.

Der Unterschied zum klassischen Mergesort besteht darin, daß dieser als Teillisten für den ersten Durchlauf einzelne Elemente verwendet.

Natural Mergesort benötigt offensichtlich $O(\log \mathit{Runs}(X))$ Durchläufe und in jedem Durchlauf $O(n)$ Operationen, insgesamt also $O(n \log \mathit{Runs}(X))$ Operationen.

Nach [Man85, Lemma 2] gilt

$$\log |b(X, \mathit{Runs})| = \Omega(n \log \mathit{Runs}(X)) \tag{7.2}$$

und damit ist Natural Mergesort *Runs*-optimal.

Natural Mergesort benötigt $O(1)$ Speicher über den vom verwendeten Merge-Algorithmus benötigten Speicher hinaus (siehe auch Abschnitt 7.3).

7.2.2 Par-Guess-Sort

Par-Guess-Sort [EW89] rät einen Wert p für $\mathit{Par}(X)$, beginnend mit 1. Ist $p \geq \mathit{Par}(X)$, so ist jede Teilliste $(x_i, x_{i+p+1}, x_{i+2(p+1)}, \ldots)$ sortiert, da die Elemente in der Eingabeliste X den Abstand $p + 1 > \mathit{Par}(X)$ haben und daher keine Inversionen enthalten.

Ist dagegen $p < \mathit{Par}(X)$, so ist mindestens eine der Teillisten nicht sortiert und Par-Guess-Sort quadriert den Wert p, bis er groß genug ist. Anschließend werden die $p + 1$ Teillisten als sortierte Eingabelisten für einen Mergesort verwendet.

Daß p bereits groß genug ist, kann jeweils in linearer Zeit daran festgestellt werden, daß die oben angegebenen Listen auf Sortiertheit geprüft werden. Alternativ kann auch mit dem Verschmelzen der Teillisten begonnen werden und dabei jeweils geprüft werden, ob die Teillisten aufsteigend sortiert sind.

Wie in [EW89] gezeigt wird, ist Par-Guess-Sort *Par*-optimal.

Genau wie Natural Mergesort braucht auch Par-Guess-Sort zusätzlich zum verwendeten Merge-Algorithmus nur $O(1)$ Speicher. Es ist aber bei der Wahl des Merge-Algorithmus zu beachten, daß die zu verschmelzenden Teillisten nicht zusammenhängend sind.

7.2.3 Adaptive Heapsort

Adaptive Heapsort [LP93] ist stark an Heapsort angelehnt, baut aber nicht aus allen Listenelementen einen Heap auf, sondern nur aus denen, die als Maximum der noch nicht einsortierten Elemente in Frage kommen. Diese Datenstruktur wird „Maxtree“ genannt, der Algorithmus heißt danach in [LP89] auch „Maxtree-Sort“.

Adaptive Heapsort benötigt $O(n+n\log(Osc(X)/n))$ Operationen und $O(n)$ Speicher.

Aus den Definitionen von Osc, $Cross'$ und Osc' ((4.1), (4.2) und (4.3)) ergibt sich direkt

$$Osc(X) \leq \frac{1}{2} Osc'(X) \tag{7.3}$$

und damit

$$n + n\log\frac{Osc(X)}{n} = O\Big(n + n\log\frac{Osc'(X)}{n}\Big). \tag{7.4}$$

Folglich benötigt Adaptive Heapsort nicht mehr als $O(n+n\log(Osc'(X)/n))$ Operationen.

Nach [LP93] ist

$$\log|b(X, Osc)| = \Omega\Big(n\log\frac{Osc(X)}{n}\Big), \tag{7.5}$$

es ist aber offen, ob dieses Ergebnis auch auf Osc' übertragen werden kann. Wäre dies der Fall, so wäre Adaptive Heapsort Osc'-optimal.

7.3 Merge-Algorithmen

Sortierverfahren, die von Mergesort abgeleitet sind, verwenden einen Teilalgorithmus, der zwei bereits sortierte Listen X und Y zu einer sortierten Liste verschmilzt (verschmelzen = “to merge”). Dafür werden $O(|X| + |Y|)$ Operationen benötigt.

Wählt man den einfachsten Algorithmus, der dieses leistet – iteratives Vergleichen der jeweils ersten Elemente der Listen und Übertragen des kleineren an das Ende einer neuen Liste –, so benötigt man zusätzlich $O(|X| + |Y|)$ Speicher für diese neue Liste.

Es ist aber auch möglich, mit $O(|X| + |Y|)$ Operationen und $O(1)$ Speicher auszukommen, indem man jeweils Teillisten der Länge $\approx \sqrt{|X| + |Y|}$ miteinander verschmilzt, und eine am Ende liegende sortierte Teilliste der Länge $\lceil\sqrt{|X| + |Y|}\rceil$ permutiert, um so $O(\log\sqrt{|X| + |Y|}!)$ zusätzliche Bit zu speichern.

Zu Details sei auf [Kro69], [Par77] und [HL88] verwiesen.

Leider ist der praktische Nutzen gering, siehe auch Kapitel 8.

8 Vergleich der Algorithmen durch Simulation

Um zu untersuchen, wie sich die Algorithmen in der Praxis bewähren, wurden Natural Mergesort, Par-Guess-Sort und Adaptive Heapsort implementiert.

Der Merge-Algorithmus wurde als sukzessives Vergleichen und Kopieren in eine neue Liste implementiert. Um Datenbewegungen dabei zu minimieren, wird von zwei Listen abwechselnd zuerst die eine als Quell- und die andere als Zielliste verwendet, dann umgekehrt die erste als Ziel und die zweite als Quelle. Bei einer ungeraden Anzahl von Durchläufen wird zum Schluß noch einmal kopiert, um das Endergebnis im selben Speicherbereich zu speichern, in dem sich die Eingabe befand.

Für Par-Guess-Sort wurde derselbe Merge-Algorithmus verwendet. Dazu werden die Teillisten nach dem Bestimmen von $p \geq Par(X)$ einmal umkopiert, um sie in zusammenhängende Teillisten umzuwandeln.

Für Natural Mergesort wurde als zusätzliche Variante die Option eingebaut, absteigende "runs" vor dem ersten Durchlauf umzudrehen.

Weiterhin wurden eine schnelle und eine langsame Vergleichsfunktion implementiert, um zu untersuchen, mit welchem Anteil die für das Vergleichen benötigte Zeit in die Gesamtzeit eingeht.

Um verschieden stark vorsortierte Testdaten zu bekommen, wird zuerst eine Liste $(1, \ldots, n)$ zufällig permutiert und diese unverändert verwendet. Dann werden mehrfach Operationen angewendet, die die Ordnung in der Liste erhöhen:

1. zwei beliebige Elemente werden ausgewählt und vertauscht, wenn sie eine Inversion bilden,
2. ein Element wird auf die Position getauscht, die es in einer sortierten Liste hätte,
3. zwei Elemente werden wie bei Punkt 1 behandelt, aber eins der Elemente wird mit geringem Abstand zum Anfang der Liste und eins mit geringen Abstand zum Ende der Liste gewählt; der größtmögliche Abstand wird bei jedem Schritt vergrößert.

Es wird jeweils eine festgelegte der drei Operationen mit einer festgelegten Häufigkeit ausgeführt und dann die entstandene Liste als Eingabeliste verwendet. Dieser Ablauf wird zehnmal wiederholt, so daß insgesamt eine vollkommen unsortierte und zehn unterschiedlich stark vorsortierte Listen zur Verfügung stehen.

Für jede dieser elf Listen werden die Werte der vorgestellten Maße berechnet, damit erkennbar ist, welche der Vorsortierungs-Operationen von welchen Maßen besonders stark berücksichtigt werden.

Schließlich wird jede dieser Listen einmal mit jedem der implementierten Sortierverfahren sortiert und dabei die verbrauchte Rechenzeit gemessen. Zusätzlich wird dasselbe mit der systemseitig vorhandenen Funktion `qsort` – einer Quicksort-Implementation – durchgeführt, um abschätzen zu können, ob überhaupt ein Geschwindigkeitsgewinn zu erwarten ist.

Während des Implementierens hat sich gezeigt, daß zum Teil unbedeutend scheinende Änderungen am Programm recht große Wirkung auf die Ablaufgeschwindigkeit haben können. Insbesondere war es notwendig, die einzelnen Routinen durch den Compiler so umordnen zu lassen, daß Cache-Konflikte vermieden werden. Andernfalls können minimale Änderungen an einem Sortierverfahren dazu führen, daß sich für ein anderes Verfahren im selben Programm die Ablaufzeit drastisch ändert.

Eine Implementation des Mergeverfahrens nach [HL88] war wenig erfolgreich: Im Vergleich zu einer Routine, die die beiden Teillisten in einen zusätzlichen Speicherbereich kopiert und von da aus durch elementweises Vergleichen und Kopieren in die ursprüngliche Liste die Teillisten verschmilzt, war jenes Verfahren etwa um den Faktor 2 langsamer. Die Implementation des Mergeverfahrens nach [Par77] war noch sehr viel langsamer.

Der Einsatz eines solchen Verfahrens lohnt sich folglich wohl nur, wenn die zu bearbeitende Liste nur einmal, aber nicht zweimal in den verfügbaren Hauptspeicher paßt. Bei der Simulation wurde daher das einfache Verfahren mit $O(n)$ Speicherbedarf verwendet.

8.1 Details der Simulationsumgebung

Die Programme wurden auf einer DEC Alpha 21066A mit 233 MHz Prozessortakt, je 8 Kilobyte prozessorinternem Befehls- und Datencache und 256 Kilobyte externem Cache entwickelt und ausgeführt.

Der verwendete Rechner hatte mit 128 Megabyte genügend Hauptspeicher, um Effekte durch Paging auszuschließen. Zu einer Untersuchung des Paging-Verhaltens von Sortierverfahren sei beispielsweise auf [AEH84] verwiesen.

Als Betriebssystem kam Digital UNIX 4.0A mit der Version 5.2-036 des DEC-C-Compilers zum Einsatz.

Für das Umordnen der Routinen wurden der `pixie`-Profiler und die Umordnungsoption `-cord` des C-Compilers verwendet.

Die Zeitmessungen wurden mittels einer Systemfunktion durchgeführt, die die Abfrage der verwendeten Rechenzeit in Mikrosekunden gestattet. Dieser Wert wird aber durch das Betriebssystem nur unter Verwendung eines

1024 Hz-Taktgebers in Schritten von $\lfloor 1/1024 \rfloor = 976$ und $\lceil 1/1024 \rceil = 977$ aktualisiert.

Daher wurden für die Simulation die abgefragten Zeiten auf ganze Millisekunden gerundet (siehe Anhang A.4.8). Ein Abschneiden der drei letzten Dezimalstellen hätte dagegen für kurze Zeitabschnitte zu großen relativen Fehlern geführt.

8.2 Ergebnisse

Sieht man sich die Ergebnisprotokolle in Anhang B.1 an – die zugegebenermaßen recht umfangreich sind, nur lassen sich die Ergebnisse auch kaum mit wenigen Zahlen erfassen – so fällt folgendes auf:

Adaptive Heapsort ist durchweg der langsamste Algorithmus. Er profitiert zwar von niedrigen Osc'-Werten, wie man in Tabelle 2 auf Seite 99 gut ablesen kann (die Rechenzeit reduziert sich von 0.328s für die unsortierte Liste mit $Osc'(X) \approx 16 \cdot 10^6$ auf 0.151s für die in zehn Schritten vorsortierte Liste mit $Osc'(X) \approx 13 \cdot 10^4$), ist aber selbst für die stark vorsortierte Liste noch langsamer als die anderen vier Algorithmen für die zufällige Liste.

Die Variante von Natural Mergesort, die "down-runs" vor dem Sortieren umdreht, ist nicht nennenswert schneller als die andere Variante, in einigen Fällen sogar etwas langsamer.

Eine Adaption von Par-Guess-Sort an Par ist nicht festzustellen, In Tabelle 1 auf Seite 97 braucht Par-Guess-Sort 0.119s für die unsortierte Liste mit $Par(X) = 9998$ und 0.114s für die stark vorsortierte Liste mit $Par(X) = 865$. In Tabelle 2 ändert sich der Wert von Par durch das Vorsortieren nicht wesentlich – es wurden jeweils Elemente auf ihre Zielposition getauscht. In Tabelle 3 reduziert sich der Wert von Par von 9998 auf 2146, aber Par-Guess-Sort wird nicht merklich schneller.

Natural Mergesort ist nur in Tabelle 2 langsamer als `qsort`. Daß `qsort` stark davon profitiert, daß Elemente bereits auf ihrer endgültigen Position in der Liste sind, deutet auf eine Median-basierte Implementation hin. Daß Natural Mergesort bei einer vollkommen unsortierten Liste schneller ist als `qsort` läßt allerdings Zweifel an der Qualität dieser Quicksort-Implementation aufkommen.

Betrachtet man nur die Einträge, die Natural Mergesort im Zusammenhang mit *Runs* betreffen, so kann man in der Tat eine Korrelation feststellen. In Tabelle 2 wird Natural Mergesort sogar stärker beschleunigt als es mittels *Runs* vorhergesagt wurde. Bleibt $Runs(X)$ hoch, so wird auch Natural Mergesort nicht nennenswert schneller (Tabellen 1 und 3).

Mit der langsamen Vergleichsfunktion ändern sich die Ergebnisse im großen und ganzen kaum, sie sind daher nicht gesondert aufgeführt. Die Algorithmen verschwenden offenbar nicht sehr viel Rechenzeit mit Verwaltungsaufgaben und Kopieren von Daten.

Insgesamt sind die Ergebnisse recht enttäuschend. Die Simulationsergebnisse empfehlen Natural Mergesort als schnelles Sortierverfahren, die anderen Algorithmen führen offenbar so viele zusätzliche Vergleiche durch, daß die Wirkung der Adaption nicht zum Tragen kommt.

Auch bei 10^5-elementigen Listen war kein großer Unterschied zu den Ergebnissen der verwendeten 10^4-elementigen zu erkennen (diese Ergebnisse sind ebenfalls nicht abgedruckt). Möglicherweise ergeben sich echte Vorteile erst bei sehr viel längeren Eingabelisten.

Anhang

A Quelltexte

A.1 Hauptprogramme

A.1.1 sim.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#include "types.h"
#include "memory.h"
#include "util.h"
#include "time.h"
#include "measures.h"
#include "presort.h"
#include "count.h"
#include "mergesort.h"
#include "parguess.h"
#include "adaptive_heapsort.h"

#pragma ident "@(#)sim.c    1.44"


/* Faktor für die erwartete Laufzeit */
#define SCALE 1000.0


/* Werte der Maße für die aktuelle Liste */
static long inv_x, runs_x, rem_x, exc_x, par_x, osc_x;

/* erwartete Zeiten */
static double t_inv, t_runs, t_rem, t_exc, t_par, t_osc;

/* normalerweise auch die Werte für Inv und Par berechnen */
static int skip_expensive = 0;

/* normalerweise die Anzahlen für Vergleiche etc. nicht alle ausgeben */
static int print_counters = 0;
```

```
/* paßt compare für den Aufruf durch qsort an */
static int cmp(const void *a, const void *b)
{
    switch (compare((el_t *)a, (el_t *)b))
    {
    case In_Order:
        return -1;

    case Equal:
        return 0;

    case Out_Of_Order:
        return 1;
    }
}


/* stellt fest, ob list sortiert ist */
static int is_sorted(int n, el_t *list)
{
    int i;

    for (i = 0; i + 1 < n; i++)
        if (list[i] > list[i + 1])
            return 0; /* nein */

    return 1; /* ja */
}


static void print_line(char *name, int ms)
{
    if (print_counters)
    {
        printf("%s: %ld compare, %ld swap,",
            name, n_compare, n_swap);
        printf(" %ld copy, time = %d.%03ds\n",
            n_copy, ms / 1000, ms % 1000);
    }
```

```
    printf("%s", name);
    if (!skip_expensive)
        printf("%11.3f", ms / t_inv * SCALE);
    printf("%11.3f", ms / t_runs * SCALE);
    printf("%11.3f", ms / t_rem * SCALE);
    printf("%11.3f", ms / t_exc * SCALE);
    if (!skip_expensive)
        printf("%11.3f", ms / t_par * SCALE);
    printf("%11.3f", ms / t_osc * SCALE);

    printf("%3d.%03ds\n", ms / 1000, ms % 1000);
}


/* berechnet max{0, log_2 x} */
static double log_2(double x)
{
    if (x > 1.0)
        return log2(x);
    else
        return 0.0;
}


int main(int argc, char **argv)
{
    el_t *list;
    int n, i, m, k;
    int skip, do_randomize;
    int mode;

    skip = 1;

    if (skip_expensive = argc > skip &&
        !strcmp(argv[skip], "-fast"))
        skip++;

    if (do_randomize = argc > skip &&
        !strcmp(argv[skip], "-randomize"))
        skip++;

    if (slow_compare = argc > skip &&
        !strcmp(argv[skip], "-slow"))
        skip++;
```

```
if (print_counters = argc > skip &&
    !strcmp(argv[skip], "-counters"))
    skip++;

k = 0;
if (argc > skip + 1 && !strcmp(argv[skip], "-presort"))
{
    k = atoi(argv[skip + 1]);
    skip += 2;
}

if (argc != skip + 2)
{
    fprintf(stderr,
        "usage: %s [-fast] [-randomize] [-slow] "
        "[-counters] [-presort K] presort-mode N\n",
        argv[0]);
    exit(10);
}

mode = atoi(argv[skip]);
n = atoi(argv[skip + 1]);
if (!k)
    k = n;

if (do_randomize)
    randomize();

list = make_list(n);
shuffle_list(list);

printf("    |X| = %d\n    presort: mode %d,", n, mode);
printf(" 10 x %d presort steps\n\n", k);

if (!skip_expensive)
    printf("  %11s%11s%11s%11s%11s%11s\n",
        "Inv(X)", "Runs(X)", "Rem(X)",
        "Exc(X)", "Par(X)", "Osc'(X)");
else
    printf("  %11s%11s%11s%11s\n",
        "Runs(X)", "Rem(X)", "Exc(X)", "Osc'(X)");
```

```
for (m = 0; m <= 10; m++)
{
    el_t *tmp_list;
    int delta;

    printf("\n  ");

    if (!skip_expensive)
    {
        inv_x = inv(list);
        printf("%11ld", inv_x);
        fflush(stdout);

        t_inv = n + n * log_2((double)inv_x / n);
    }

    runs_x = runs(list);
    printf("%11ld", runs_x);
    fflush(stdout);

    t_runs = n + n * log_2((double)runs_x);

    rem_x = rem(list);
    printf("%11ld", rem_x);
    fflush(stdout);

    t_rem = n + n * log_2((double)rem_x);

    exc_x = exc(list);
    printf("%11ld", exc_x);
    fflush(stdout);

    t_exc = n + n * log_2((double)exc_x);

    if (!skip_expensive)
    {
        par_x = par(list);
        printf("%11ld", par_x);
        fflush(stdout);

        t_par = n + n * log_2((double)par_x);
    }

    osc_x = osc_(list);
    printf("%11ld\n", osc_x);
    fflush(stdout);
```

```
t_osc = n + n * log_2((double)osc_x / n);

tmp_list = copy_list(list);
n_compare = n_swap = n_copy = 0;
turn_down_runs = 0;

start_timer();
natural_mergesort(tmp_list);
delta = read_timer_ms();

if (!is_sorted(n, tmp_list))
    abort();
free_list(tmp_list);
print_line("M ", delta);

tmp_list = copy_list(list);
n_compare = n_swap = n_copy = 0;
turn_down_runs = 1;

start_timer();
natural_mergesort(tmp_list);
delta = read_timer_ms();

if (!is_sorted(n, tmp_list))
    abort();
free_list(tmp_list);
print_line("M'", delta);

tmp_list = copy_list(list);
n_compare = n_swap = n_copy = 0;

start_timer();
par_guess_sort(tmp_list);
delta = read_timer_ms();

if (!is_sorted(n, tmp_list))
    abort();
free_list(tmp_list);
print_line("P ", delta);

tmp_list = copy_list(list);
n_compare = n_swap = n_copy = 0;

start_timer();
adaptive_heapsort(tmp_list);
delta = read_timer_ms();
```

```
        if (!is_sorted(n, tmp_list))
            abort();
        free_list(tmp_list);
        print_line("H ", delta);

        tmp_list = copy_list(list);
        n_compare = n_swap = n_copy = 0;

        start_timer();
        qsort(tmp_list, n, sizeof(el_t), cmp);
        delta = read_timer_ms();

        if (!is_sorted(n, tmp_list))
            abort();
        free_list(tmp_list);
        print_line("Q ", delta);

        presort(list, mode, k);
    }

    free_list(list);
    return 0;
}
```

A.1.2 osc_max.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "types.h"
#include "memory.h"
#include "util.h"
#include "measures.h"

#pragma ident "@(#)osc_max.c    1.8"


/* bisheriges Maximum von Osc' */
static long maxval;

/* erste Liste, bei der das Maximum aufgetreten ist */
static el_t *maxlist;

/* Anzahl des Auftretens des Maximums */
static long n_max;
```

```
/* gibt eine Liste aus */
static void print_list(el_t *list)
{
    int i, n;

    n = length(list);
    printf("(");
    for (i = 0; i < n; i++)
    {
        if (i > 0)
            printf(", ");
        printf("%d", list[i]);
    }
    printf(")\n");
}


/* sucht in Permutationen der Liste nach denen mit maximalem Osc',
   verändert nur die letzten n − base Listenelemente */
static void permute_and_find(el_t *list, int base, int n)
{
    int i, j;

    if (base >= n)
    {
        long val;

        val = osc_(list);
        if (val > maxval)
        {
            free_list(maxlist);
            maxlist = copy_list(list);
            maxval = val;
            n_max = 1;
        }
        else if (val == maxval)
            n_max++;

        return;
    }
```

```
    for (i = 1; i <= n; i++)
    {
        for (j = 0; j < base; j++)
            if (list[j] == i)
                break;

        if (j >= base)
        {
            list[base] = i;
            permute_and_find(list, base + 1, n);
        }
    }
}


/* sucht in Permutationen der Liste und gibt die
   mit maximalem Osc' aus,
   verändert nur die letzten n − base Listenelemente */
static void permute_and_print(el_t *list, int base, int n)
{
    int i, j;

    if (base >= n)
    {
        long val;

        val = osc_(list);
        if (val > maxval)
            abort();
        if (val == maxval)
            print_list(list);

        return;
    }

    for (i = 1; i <= n; i++)
    {
        for (j = 0; j < base; j++)
            if (list[j] == i)
                break;
```

```
        if (j >= base)
        {
            list[base] = i;
            permute_and_print(list, base + 1, n);
        }
    }
}


/* sucht in allen Permutationen der Liste nach denen
   mit maximalem Osc' */
static void find_max(int n)
{
    el_t *list;

    list = make_list(n);
    permute_and_find(list, 0, n);
    free_list(list);
}


/* sucht in allen Permutationen der Liste und gibt die
   mit maximalem Osc' aus */
static void print_max(int n)
{
    el_t *list;

    list = make_list(n);
    permute_and_print(list, 0, n);
    free_list(list);
}


int main(int argc, char **argv)
{
    int n;
    int skip;

    skip = 1;
    if (argc != skip + 1)
    {
        fprintf(stderr, "usage: %s N\n", argv[0]);
        exit(10);
    }

    n = atoi(argv[skip]);
```

```
    maxval = -1;
    n_max = 0;
    maxlist = make_list(n);
    find_max(n);

    printf("n = %d, max Osc' = %ld, # = %ld\n",
        n, maxval, n_max);

    print_max(n);
    return 0;
}
```

A.2 Maße

A.2.1 measures.h

```
#pragma ident "@(#)measures.h   1.9"


long inv(el_t *list);
long runs(el_t *list);
long rem(el_t *list);
long exc(el_t *list);
long par(el_t *list);
long osc_(el_t *list);
```

A.2.2 measures.c

```
#include <stdlib.h>

#include "types.h"
#include "memory.h"
#include "measures.h"

#pragma ident "@(#)measures.c   1.19"


/* berechnet Inv(X) mit O(|X|^2) Operationen */
long inv(el_t *list)
{
    long val;
    int i, j, n;

    n = length(list);
    val = 0;

    for (i = 0; i < n; i++)
        for (j = i + 1; j < n; j++)
            if (list[i] > list[j])
                val++;

    return val;
}
```

```
/* berechnet Runs(X) mit O(|X|) Operationen */
long runs(el_t *list)
{
    long val;
    int i, n;

    n = length(list);
    val = 0;

    for (i = 0; i + 1 < n; i++)
        if (list[i] > list[i + 1])
            val++;

    return val;
}


/* berechnet Rem(X) nach [Dij80] mit O(|X| log |X|) Operationen */
static int longest_upsequence(el_t *list)
{
    int N, n, k;
    el_t *m;

    N = length(list);
    if (N == 0)
        return 0;

    k = n = 1;
    m = alloc_list(N + 1);
    m[1] = list[0];

    while (n < N)
    {
        if (m[k] <= list[n])
        {
            k++;
            m[k] = list[n];
        }
        else if (list[n] < m[1])
            m[1] = list[n];
        else
        {
            int i, j, h;
```

```
            i = 1;
            j = k;
            while (i < j - 1)
            {
                h = (i + j) / 2;
                if (m[h] <= list[n])
                    i = h;
                else
                    j = h;
            }
            m[j] = list[n];
        }

        n++;
    }

    free_list(m);
    return k;
}


long rem(el_t *list)
{
    int n;

    n = length(list);
    return n - longest_upsequence(list);
}


/* berechnet Exc(X) – nur für Permutationen von (1,... ,n) */
long exc(el_t *list)
{
    el_t *flags;
    int i, k, n, m;

    n = length(list);

    /* Listen von Flags sind nicht vorgesehen,
       deshalb wird eine normale Liste verwendet */
    flags = alloc_list(n);

    for (i = 0; i < n; i++)
        flags[i] = 0;
```

```
    m = 0;
    for (k = 0;; k++)
    {
        for (i = m; i < n && flags[i]; i++);
        if (i >= n)
            break;

        m = i + 1;

        while (!flags[i])
        {
            flags[i] = 1;

            /* Position, die dieses Element erhalten muß */
            i = list[i] - 1;
        }
    }

    free_list(flags);
    return n - k;
}


/* berechnet Par(X) mit O(|X|^2) Operationen,
   aber schnell für große Werte von Par(X) */
long par(el_t *list)
{
    int p, i, n;

    n = length(list);

    for (p = n - 1; p > 0; p--)
    {
        for (i = 0; i + p < n; i++)
            if (list[i] > list[i + p])
                /* fertig */
                return p;
    }

    /* keine Inversion gefunden */
    return 0;
}
```

```
/* berechnet Osc'(X) mit O(|X|) Operationen –
   nur für Permutationen von (1,... ,n) */
long osc_(el_t *list)
{
    long val;
    int i, n;

    n = length(list);
    val = 0;

    if (n > 0)
    {
        /* links korrigieren */
        val += list[0] - 1;

        /* rechts korrigieren */
        val += n - list[n - 1];
    }

    for (i = 0; i + 1 < n; i++)
        if (list[i] < list[i + 1])
            val += list[i + 1] - list[i] - 1;
        else
            val += list[i] - list[i + 1] - 1;

    if (val % 2 != 0)
        abort();

    return val / 2;
}
```

A.3 Sortierverfahren

A.3.1 mergesort.h

```
#pragma ident "@(#)mergesort.h  1.3"


/* kann auf 1 gesetzt werden, damit auch
   absteigende “runs” berücksichtigt werden */
extern int turn_down_runs;


void natural_mergesort(el_t *list);
```

A.3.2 mergesort.c

```
#include <stdlib.h>

#include "types.h"
#include "memory.h"
#include "count.h"
#include "merge.h"
#include "mergesort.h"

#pragma ident "@(#)mergesort.c  1.13"


/* normalerweise absteigende “runs” nicht berücksichtigen */
int turn_down_runs = 0;


/* findet den nächsten “run”, dreht optional “down-runs” um */
static int get_next_run(int block_i, int offset, el_t *list)
{
    int n, i, k;

    n = length(list);
    if (offset >= n)
        abort();

    if (offset + 1 == n)
        return 1;
```

```
    if (compare(&list[offset], &list[offset + 1]) !=
        Out_Of_Order)
    {
        for (i = offset + 1; i + 1 < n; i++)
            if (compare(&list[i], &list[i + 1]) ==
                Out_Of_Order)
                break;

        k = i + 1 - offset;
    }
    else if (turn_down_runs)
    {
        for (i = offset + 1; i + 1 < n; i++)
            if (compare(&list[i], &list[i + 1]) == In_Order)
                break;

        k = i + 1 - offset;

        if (k > 2 || offset + k == n)
            for (i = 0; 2 * i < k; i++)
                swap(&list[offset + i],
                    &list[offset + k - 1 - i]);
        else
            k = 1;
    }
    else
        k = 1;

    return k;
}


void natural_mergesort(el_t *list)
{
    int n;
    el_t *tmp_list, *dest_list;

    n = length(list);
    tmp_list = alloc_list(n);

    dest_list = flexible_mergesort(list, tmp_list,
        get_next_run);
    if (dest_list != list)
    {
        block_t all;
```

```
        all.offset = 0;
        all.n = n;
        copy_block(&all, dest_list, &all, list);
    }

    free_list(tmp_list);
}
```

A.3.3 merge.h

```
#pragma ident "@(#)merge.h  1.6"


typedef int (*next_block_f)(int block_i, int offset,
    el_t *list);


void merge(block_t *block1_p, block_t *block2_p,
    el_t *src_list, block_t *dest_block_p, el_t *dest_list);
el_t *flexible_mergesort(el_t *src_list, el_t *tmp_list,
    next_block_f get_next);

void copy_block(block_t *src_block_p, el_t *src_list,
    block_t *dest_block_p, el_t *dest_list);
```

A.3.4 merge.c

```
#include <stdlib.h>

#include "types.h"
#include "memory.h"
#include "count.h"
#include "merge.h"

#pragma ident "@(#)merge.c  1.8"


/* verschmilzt zwei benachbarte Blöcke der Liste src_list zu einem
   in der Liste dest_list an derselben Position */
void merge(block_t *block1_p, block_t *block2_p,
    el_t *src_list, block_t *dest_block_p, el_t *dest_list)
{
    int i1, i2, i_dest;
    int n1, n2;
```

```
i1 = block1_p->offset;
n1 = block1_p->n;

i2 = block2_p->offset;
n2 = block2_p->n;

if (i2 != i1 + n1)
    abort();

i_dest = i1;

/* erst jetzt schreiben, damit dest_block in block1 oder
   block2 gespeichert werden kann */
dest_block_p->offset = i_dest;
dest_block_p->n = n1 + n2;

while (n1 > 0 && n2 > 0)
    if (compare(&src_list[i1], &src_list[i2]) !=
        Out_Of_Order)
    {
        copy(&src_list[i1], &dest_list[i_dest]);
        i1++;
        i_dest++;
        n1--;
    }
    else
    {
        copy(&src_list[i2], &dest_list[i_dest]);
        i2++;
        i_dest++;
        n2--;
    }

while (n1 > 0)
{
    copy(&src_list[i1], &dest_list[i_dest]);
    i1++;
    i_dest++;
    n1--;
}
```

```
    while (n2 > 0)
    {
        copy(&src_list[i2], &dest_list[i_dest]);
        i2++;
        i_dest++;
        n2--;
    }
}


/* durchläuft einmal die Liste src_list, verschmilzt die
   in run_list angegebenen Blöcke paarweise und speichert
   das Ergebnis in dest_list */
static void merge_pass(el_t *src_list, el_t *dest_list,
    block_t *run_list)
{
    int i, n_runs;

    n_runs = number_of_blocks(run_list);

    for (i = 0; 2 * i + 1 < n_runs; i++)
        merge(&run_list[2 * i], &run_list[2 * i + 1],
            src_list, &run_list[i], dest_list);

    if (2 * i < n_runs)
    {
        copy_block(&run_list[2 * i], src_list,
            &run_list[i], dest_list);
        i++;
    }

    truncate_block_list(run_list, i);
}


/* sortiert die Blöcke in src_list;
   die Funktion get_next liefert jeweils den nächsten Block;
   das Ergebnis ist entweder in src_list oder in tmp_list,
   je nach Anzahl der merge_pass-Aufrufe */
el_t *flexible_mergesort(el_t *src_list, el_t *tmp_list,
    next_block_f get_next)
{
    int k, n, m;
    block_t *block_list;
    el_t *dest_list;
```

```
    n = length(src_list);

    /* mindestens ein Block */
    block_list = alloc_block_list(1);

    k = m = 0;
    for (;;)
    {
        int l;

        l = get_next(k, m, src_list);

        block_list[k].offset = m;
        block_list[k].n = l;

        k++;
        m += l;

        if (m > n)
            abort();
        if (m == n)
            break;

        block_list = extend_block_list(block_list, k + 1);
    }

    /* die Daten sind immer in src_list und sollen nach dest_list */
    dest_list = tmp_list;
    while (number_of_blocks(block_list) > 1)
    {
        merge_pass(src_list, dest_list, block_list);

        /* in der nächsten Runde andersrum */
        tmp_list = src_list;
        src_list = dest_list;
        dest_list = tmp_list;
    }

    if (number_of_blocks(block_list) != 1)
        abort();

    free_block_list(block_list);
    return src_list;
}
```

```
/* kopiert einen Block aus der Liste src_list nach
   dest_list, wird bei ungerader Blockanzahl verwendet */
void copy_block(block_t *src_block_p, el_t *src_list,
    block_t *dest_block_p, el_t *dest_list)
{
    int i, n;

    i = src_block_p->offset;
    n = src_block_p->n;

    dest_block_p->offset = i;
    dest_block_p->n = n;

    while (n > 0)
    {
        copy(&src_list[i], &dest_list[i]);
        i++;
        n--;
    }
}
```

A.3.5 parguess.h

```
#pragma ident "@(#)parguess.h   1.2"


void par_guess_sort(el_t *list);
```

A.3.6 parguess.c

```
#include "types.h"
#include "memory.h"
#include "count.h"
#include "merge.h"
#include "parguess.h"

#pragma ident "@(#)parguess.c   1.16"


/* C hat keine Closures */
static int global_p;
```

```
/* stellt fest, ob die Teilliste (x_offset, x_offset+step, x_offset+2step, ...)
   sortiert ist */
static int sublist_is_sorted(int offset, int step,
    int n, el_t *list)
{
    int i, j;

    for (i = offset; (j = i + step) < n; i = j)
        if (compare(&list[i], &list[j]) == Out_Of_Order)
            return 0;

    return 1;
}


/* die Blöcke sind bis auf Rundung alle gleich groß */
static int get_next_block(int block_i, int offset, el_t *list)
{
    /* (n − i)/(p + 1) aufrunden */
    return (length(list) + global_p - block_i) /
        (global_p + 1);
}


static int try_to_sort(int n, long p, el_t *list,
    el_t *tmp_list, int *swapped_p)
{
    int i, j, m, n_i;
    el_t *dest_list;

    /* leere Blöcke vermeiden */
    if (p + 1 > n)
        p = n - 1;

    for (i = 0; i < p + 1; i++)
    {
        /* (n − i)/(p + 1) aufrunden */
        n_i = (n + p - i) / (p + 1);

        if (!sublist_is_sorted(i, p + 1, n, list))
            return 0; /* p falsch geschätzt */
    }
```

```
    /* p richtig geschätzt; Blöcke zusammenstellen */
    m = 0;
    for (i = 0; i < p + 1; i++)
    {
        /* (n − i)/(p + 1) aufrunden */
        n_i = (n + p - i) / (p + 1);

        for (j = 0; j < n_i; j++)
            copy(&list[i + (p + 1) * j], &tmp_list[m + j]);

        m += n_i;
    }

    global_p = p;
    dest_list = flexible_mergesort(tmp_list, list,
        get_next_block);

    *swapped_p = dest_list != list;
    return 1; /* Erfolg */
}


void par_guess_sort(el_t *list)
{
    int i, n;
    long p;
    el_t *tmp_list;
    int swapped;

    n = length(list);

    if (sublist_is_sorted(0, 1, n, list))
        return;

    tmp_list = alloc_list(n);

    p = 2;
    for (;;)
    {
        if (try_to_sort(n, p, list, tmp_list, &swapped))
            break;

        p = p * p;
    }
```

```
    if (swapped)
        for (i = 0; i < n; i++)
            copy(&tmp_list[i], &list[i]);

    free_list(tmp_list);
}
```

A.3.7 adaptive_heapsort.h

```
#pragma ident "@(#)adaptive_heapsort.h  1.4"


void adaptive_heapsort(el_t *list);
```

A.3.8 adaptive_heapsort.c

```
#include <stdlib.h>

#include "types.h"
#include "memory.h"
#include "count.h"
#include "adaptive_heapsort.h"

#pragma ident "@(#)adaptive_heapsort.c  1.11"


typedef struct
{
    int left, right;
    el_t el;
} maxtree_node_t;


static int n_ceil;
static int *search_tree;
static int *ga, *gb;

static maxtree_node_t *maxtree;
static int maxtree_length;

static int heap_length, sorted_base;
static int *heap;
```

```
/* vergleicht zwei Elemente des Maxtree */
static compare_e compare_maxtree(int i, int j)
{
    return compare(&maxtree[i].el, &maxtree[j].el);
}


static void sift_up(int base, int n, int *list)
{
    int i, tmp;

    if (base >= n)
        return;

    if (base + 1 < n && compare_maxtree(list[0],
        list[base + 1]) == In_Order)

        if (compare_maxtree(list[base],
            list[base + 1]) == In_Order)
            i = base + 1;
        else
            i = base;

    else if (compare_maxtree(list[0], list[base]) == In_Order)
        i = base;
    else
        return;

    tmp = list[0];
    list[0] = list[i];
    list[i] = tmp;
    n_swap++;

    sift_up(base + i, n - i, list + i);
}


/* fügt ein neues Element in den Maxtree ein */
static void insert_into_heap(int i)
{
    int l;

    heap[heap_length] = i;
    heap_length++;
```

```
    for (l = heap_length / 2; l > 0; l /= 2)
        sift_up(l, heap_length - (l - 1), heap + l - 1);
}


/* erfernt das Maximum aus dem Maxtree und reorganisiert ihn */
static void delete_heap_max(void)
{
    int tmp;

    heap_length--;
    sorted_base--;

    tmp = heap[0];
    heap[0] = heap[heap_length];
    heap[sorted_base] = tmp;
    n_swap++;

    sift_up(1, heap_length, heap);
}


/* erzeugt zu einem Teil einer Liste einen Maxtree */
static void construct_maxtree(int i, int n, el_t *list)
{
    int l, r, k, j;
    int n_l, n_r;

tail_call:
    if (n < 1)
        return; /* fertig */

    l = n_ceil + i;
    r = n_ceil + i + n - 1;

    j = l;
    while (l < r)
    {
        int l_i, r_i;

        l_i = search_tree[l];
        r_i = search_tree[r];
```

```
    if (ga[l_i] >= i + n)
    {
        j = l;
        break;
    }

    if (gb[r_i] < i)
    {
        j = r;
        break;
    }

    l /= 2;
    r /= 2;

    if (search_tree[l] < i)
        l++;
    if (search_tree[r] >= i + n)
        r--;
    j = l;
}

j = search_tree[j];

k = maxtree_length;
maxtree[k].left = maxtree[k].right = -1;
copy(&list[j], &maxtree[k].el);
maxtree_length++;

l = i;
n_l = j - i;

r = j + 1;
n_r = i + n - (j + 1);

/* zuerst den kleineren Teil */
if (n_l > n_r)
{
    int tmp;

    tmp = r;
    r = l;
    l = tmp;
```

```
        tmp = n_r;
        n_r = n_l;
        n_l = tmp;
    }

    if (n_l > 0)
        maxtree[k].left = maxtree_length;
    if (n_r > 0)
        maxtree[k].right = maxtree_length + n_l;

    construct_maxtree(l, n_l, list);

    i = r;
    n = n_r;
    goto tail_call;
}


/* implementiert Adaptive Heapsort nach [LP93] */
void adaptive_heapsort(el_t *list)
{
    int n, k, i;
    int *stack;

    n = length(list);
    if (n == 0)
        return;

    for (n_ceil = 1; n_ceil < n; n_ceil *= 2);


    /* GA und GB bauen */

    if ((ga = malloc(n * sizeof *ga)) == NULL ||
        (gb = malloc(n * sizeof *gb)) == NULL ||
        (stack = malloc(n * sizeof *stack)) == NULL)
        exit(1);
```

```
k = 0; /* Elemente auf dem Stack */
for (i = 0; i < n; i++)
{
    while (k > 0 &&
        compare(&list[stack[k - 1]],
            &list[i]) != Out_Of_Order)
    {
        ga[stack[k - 1]] = i;
        k--;
    }

    stack[k] = i;
    k++;
}
while (k > 0)
{
    ga[stack[k - 1]] = i;
    k--;
}

k = 0; /* Elemente auf dem Stack */
for (i = n; i-- > 0;)
{
    while (k > 0 &&
        compare(&list[i],
            &list[stack[k - 1]]) == Out_Of_Order)
    {
        gb[stack[k - 1]] = i;
        k--;
    }

    stack[k] = i;
    k++;
}
while (k > 0)
{
    gb[stack[k - 1]] = i;
    k--;
}

free(stack);


/* den Suchbaum bauen */
```

```
if ((search_tree = malloc(2 * n_ceil *
    sizeof *search_tree)) == NULL)
    exit(1);

for (i = 0; i < n; i++)
    search_tree[n_ceil + i] = i;
for (; i < n_ceil; i++)
    search_tree[n_ceil + i] = -1;

for (i = n_ceil; i-- > 1;)
{
    int l, r;

    l = search_tree[2 * i];
    r = search_tree[2 * i + 1];

    if (r < 0)
        search_tree[i] = l;
    else if (compare(&list[l], &list[r]) == Out_Of_Order)
        search_tree[i] = l;
    else
        search_tree[i] = r;
}


/* den Maxtree bauen */

if ((maxtree = malloc(n * sizeof *maxtree)) == NULL)
    exit(1);

maxtree_length = 0;
construct_maxtree(0, n, list);

free(ga);
free(gb);
free(search_tree);


/* sortieren */

if ((heap = malloc(n * sizeof *heap)) == NULL)
    exit(1);

heap_length = 0;
sorted_base = n;
```

```
    insert_into_heap(0); /* die Wurzel des Maxtree */
    while (heap_length > 0)
    {
        int l, r;

        delete_heap_max();

        i = heap[sorted_base];
        l = maxtree[i].left;
        r = maxtree[i].right;

        if (l >= 0)
            insert_into_heap(l);
        if (r >= 0)
            insert_into_heap(r);
    }

    if (sorted_base != 0)
        abort();


    /* zurückkopieren */
    for (i = 0; i < n; i++)
        copy(&maxtree[heap[i]].el, &list[i]);

    free(maxtree);
    free(heap);
}
```

A.4 Hilfsroutinen

A.4.1 count.h

```
#pragma ident "@(#)count.h  1.5"


/* globale Zähler für Vergleichen, Vertauschen, Kopieren */
extern long n_compare, n_swap, n_copy;

/* kann auf 1 gesetzt werden, um eine
   langsame Vergleichsfunktion zu aktivieren */
extern int slow_compare;


/* mögliche Ergebnisse des Vergleichens */
typedef enum { In_Order, Equal, Out_Of_Order } compare_e;


compare_e compare(el_t *left, el_t *right);
void swap(el_t *a, el_t *b);
void copy(el_t *src, el_t *dest);
```

A.4.2 count.c

```
#include <stdio.h>
#include <string.h>

#include "types.h"
#include "count.h"

#pragma ident "@(#)count.c  1.9"


/* globale Zähler */
long n_compare, n_swap, n_copy;

/* normalerweise schnell vergleichen */
int slow_compare = 0;


/* vergleicht zwei Listenelemente mit drei möglichen
   Ergebnissen und zählt */
compare_e compare(el_t *left, el_t *right)
{
    n_compare++;
```

```
    /* langsam vergleichen? */
    if (slow_compare)
    {
        char l_str[20], r_str[20];
        int d;

        /* Zeit verschwenden, indem die Zahlen erst in Strings
           umgewandelt und dann als solche verglichen werden */
        sprintf(l_str, "%19d", *left);
        sprintf(r_str, "%19d", *right);

        d = strcmp(l_str, r_str);

        if (d < 0)
            return In_Order;
        else if (d == 0)
            return Equal;
        else
            return Out_Of_Order;
    }
    else
    {
        /* schnell vergleichen */
        if (*left < *right)
            return In_Order;
        else if (*left == *right)
            return Equal;
        else
            return Out_Of_Order;
    }
}


/* vertauscht zwei Listenelemente und zählt */
void swap(el_t *a, el_t *b)
{
    el_t tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
    n_swap++;
}
```

```
/* kopiert ein Listenelement und zählt */
void copy(el_t *src, el_t *dest)
{
    *dest = *src;
    n_copy++;
}
```

A.4.3 memory.h

```
#pragma ident "@(#)memory.h 1.5"


el_t *alloc_list(int n);
void free_list(el_t *list);
int length(el_t *list);

block_t *alloc_block_list(int n);
void truncate_block_list(block_t *list, int n);
block_t *extend_block_list(block_t *list, int n);
void free_block_list(block_t *list);
int number_of_blocks(block_t *list);
```

A.4.4 memory.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

#include "types.h"
#include "memory.h"

#pragma ident "@(#)memory.c 1.9"


/* die Listen enthalten vor dem ersten Element die Anzahl;
   Zeiger auf die Liste zeigen immer auf das erste Element */

struct list
{
    int n;
    el_t first_element;
};
```

```
struct block_list
{
    int n;
    block_t first_block;
};


/* Listen von el_t */

static int list_head_size(void)
{
    return offsetof(struct list, first_element);
}


static struct list *list_head(el_t *list)
{
    return (struct list *)((char *)list - list_head_size());
}


el_t *alloc_list(int n)
{
    struct list *list;

    list = malloc(n * sizeof(el_t) + list_head_size());
    if (list == NULL)
    {
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }

    list->n = n;
    return &list->first_element;
}


void free_list(el_t *list)
{
    free(list_head(list));
}
```

```
int length(el_t *list)
{
    return list_head(list)->n;
}


/* Listen von block_t */

static int block_list_head_size(void)
{
    return offsetof(struct block_list, first_block);
}


static struct block_list *block_list_head(block_t *list)
{
    return (struct block_list *)((char *)list -
        block_list_head_size());
}


block_t *alloc_block_list(int n)
{
    struct block_list *list;

    list = malloc(n * sizeof(block_t) +
        block_list_head_size());
    if (list == NULL)
    {
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }

    list->n = n;
    return &list->first_block;
}


void truncate_block_list(block_t *list, int n)
{
    struct block_list *l;

    l = block_list_head(list);

    if (l->n < n)
        abort();
```

```
    l->n = n;
}


block_t *extend_block_list(block_t *list, int n)
{
    struct block_list *l;

    l = block_list_head(list);

    l = realloc(l, n * sizeof(block_t) +
        block_list_head_size());
    if (l == NULL)
    {
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }

    l->n = n;
    return &l->first_block;
}


void free_block_list(block_t *list)
{
    free(block_list_head(list));
}


int number_of_blocks(block_t *list)
{
    return block_list_head(list)->n;
}
```

A.4.5 presort.h

```
#pragma ident "@(#)presort.h    1.2"


void presort(el_t *list, int mode, int n_iter);
```

A.4.6 presort.c

```
#include <stdlib.h>

#include "types.h"
#include "memory.h"
#include "presort.h"

#pragma ident "@(#)presort.c    1.8"


/* vertauscht die Elemente an den Positionen i und j,
   funktioniert auch für i = j */
static void swap(el_t *list, int i, int j)
{
    el_t tmp;

    tmp = list[i];
    list[i] = list[j];
    list[j] = tmp;
}


/* vertauscht die Elemente an den Positionen i und j,
   falls diese eine Inversion bilden */
static void swap_if(el_t *list, int i, int j)
{
    if (i < j && list[i] > list[j] ||
        i > j && list[i] < list[j])
        swap(list, i, j);
}


/* nur für Permutationen von (1,...,n) */
void presort(el_t *list, int mode, int n_iter)
{
    int n, iter;
    int i, j;

    n = length(list);
    if (n < 2)
        return;
```

```
    for (iter = 0; iter < n_iter; iter++)
        switch (mode)
        {
        case 1:
            i = random() % n;
            j = random() % n;
            swap_if(list, i, j);
            break;

        case 2:
            i = random() % n;
            j = list[i] - 1;

            if (i != j)
                swap(list, i, j);
            break;

        case 3:
            if (iter < n)
            {
                i = random() % (iter + 1);
                j = n - 1 - random() % (iter + 1);
            }
            else
            {
                i = random() % n;
                j = random() % n;
            }

            swap_if(list, i, j);
            break;
        }
}
```

A.4.7 time.h

```
#pragma ident "@(#)time.h   1.2"


void start_timer(void);
int read_timer_ms(void);
```

A.4.8 time.c

```
#include <string.h>
#include <sys/resource.h>

#include "time.h"

#pragma ident "@(#)time.c   1.4"


/* Zeitpunkt des letzten start_timer-Aufrufs */
static struct timeval t_start;


static void read_timer(struct timeval *t_p)
{
    struct rusage u;

    memset(&u, 0, sizeof u);
    getrusage(RUSAGE_SELF, &u);

    t_p->tv_sec = u.ru_utime.tv_sec;
    t_p->tv_usec = u.ru_utime.tv_usec;
}


/* merkt sich den jetzigen Zeitpunkt */
void start_timer(void)
{
    read_timer(&t_start);
}


/* berechnet die Differenz in Millisekunden zum gemerkten Zeitpunkt */
int read_timer_ms(void)
{
    struct timeval now, delta;

    read_timer(&now);
```

```
    delta.tv_sec = now.tv_sec - t_start.tv_sec;
    if (t_start.tv_usec > now.tv_usec)
    {
        delta.tv_sec--;
        delta.tv_usec = now.tv_usec +
            1000000 - t_start.tv_usec;
    }
    else
        delta.tv_usec = now.tv_usec - t_start.tv_usec;

    /* von Mikrosekunden auf Millisekunden runden */
    return delta.tv_sec * 1000 + (delta.tv_usec + 500) / 1000;
}
```

A.4.9 types.h

```
#pragma ident "@(#)types.h  1.5"


/* Typ eines Listenelements */
typedef int el_t;

typedef struct { int offset, n; } block_t;
```

A.4.10 util.h

```
#pragma ident "@(#)util.h   1.5"


el_t *make_list(int n);
el_t *copy_list(el_t *list);
void shuffle_list(el_t *list);
void randomize(void);
```

A.4.11 util.c

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

#include "types.h"
#include "memory.h"
#include "util.h"

#pragma ident "@(#)util.c   1.8"
```

```
/* erzeugt Liste (1,... ,n) */
el_t *make_list(int n)
{
    el_t *list;
    int i;

    list = alloc_list(n);
    for (i = 0; i < n; i++)
        list[i] = i + 1;

    return list;
}


/* kopiert eine Liste */
el_t *copy_list(el_t *list)
{
    el_t *new_list;
    int i, n;

    n = length(list);
    new_list = alloc_list(n);

    for (i = 0; i < n; i++)
        new_list[i] = list[i];

    return new_list;
}


/* verwürfelt eine Liste */
void shuffle_list(el_t *list)
{
    int i, j, k, n;

    n = length(list);
    for (i = 0; i < 5; i++)
        for (j = n; j > 1; j--)
        {
            el_t tmp;

            k = random() % j;
```

```
            tmp = list[k];
            list[k] = list[j - 1];
            list[j - 1] = tmp;
        }
}
```

/* wählt systemabhängig einen zufälligen Startwert
   für den Pseudozufallszahlengenerator */

```
void randomize(void)
{
    srandom(time(NULL) ^ getpid());
}
```

A.4.12 Makefile

```
#
#   @(#)Makefile    1.29
#

D=-g3
STD=-std1 -warnprotos
O=-fast -tune host
CFLAGS=$D $(STD) $(ADDFLAGS) $O
LDFLAGS=$O

CFILES=adaptive_heapsort.c count.c measures.c memory.c \
    merge.c mergesort.c osc_max.c parguess.c presort.c \
    sim.c time.c util.c
HFILES=adaptive_heapsort.h count.h measures.h memory.h \
    merge.h mergesort.h parguess.h presort.h time.h types.h \
    util.h

OBJS=sim.o adaptive_heapsort.o count.o measures.o \
    memory.o merge.o mergesort.o parguess.o presort.o time.o \
    util.o

SRCS=$(CFILES) $(HFILES)


all:    sim osc_max

pass1:  sim.pixie
```

```
pass2:  sim.cord
    cp sim.cord sim

sim:    $(OBJS)
    $(CC) $(LDFLAGS) -o $@ $(OBJS) -lm

osc_max:    osc_max.o measures.o memory.o util.o
    $(CC) $(LDFLAGS) -o $@ osc_max.o measures.o memory.o \
        util.o

sim.cord:   $(OBJS) sim.feedback
    $(CC) $(LDFLAGS) -o $@ -feedback sim.feedback \
        -cord $(OBJS) -lm

sim.pixie:  sim
    atom -tool pixie sim

sim.feedback:   sim.Counts
    prof -pixie -feedback $@ -truecycles 2 sim

adaptive_heapsort.o:    adaptive_heapsort.c types.h memory.h \
    count.h adaptive_heapsort.h

count.o:    count.c types.h count.h

measures.o: measures.c types.h memory.h measures.h

memory.o:   memory.c types.h memory.h

merge.o:    merge.c types.h memory.h count.h merge.h

mergesort.o:    mergesort.c types.h memory.h count.h merge.h \
    mergesort.h

osc_max.o:  osc_max.c types.h memory.h util.h measures.h

parguess.o: parguess.c types.h memory.h count.h merge.h \
    parguess.h

presort.o:  presort.c types.h memory.h presort.h

sim.o:  sim.c types.h memory.h util.h time.h \
    measures.h presort.h count.h mergesort.h parguess.h \
    adaptive_heapsort.h

time.o: time.c time.h
```

```
util.o: util.c types.h memory.h util.h


clean:
	rm -f osc_max sim
	rm -f sim.cord sim.pixie sim.Addrs sim.Counts sim.feedback
	rm -f adaptive_heapsort.o count.o osc_max.o measures.o
	rm -f memory.o merge.o mergesort.o parguess.o presort.o
	rm -f sim.o time.o util.o

$(SRCS):
	sccs get -s $@
```

B Ablaufprotokolle

B.1 Simulation der Sortierverfahren

Legende: M: Natural Mergesort, M': Natural Mergesort mit Umkehren der "down-runs", P: Par-Guess-Sort, H: Adaptive Heapsort, Q: `qsort`

Jeder Block beschreibt eine Liste, die jeweils erste, unbeschriftete Zeile gibt den Wert des Maßes an. Die letzte Spalte gibt die Rechenzeit des Algorithmus für diese Liste an. Die restlichen Felder geben das Verhältnis der tatsächlichen verbrauchten Rechenzeit zu einem erwarteten Wert an, dieser ist $n + n\log_2 \frac{M(X)}{n}$ Mikrosekunden für die Maße Inv und Osc und $n + n\log_2 M(X)$ für alle anderen Maße. Die lineare Skalierung ist dabei willkürlich.

Eine vollständige Tabelle besteht jeweils aus elf Blöcken, der nicht vorsortierten Liste und den in zehn Schritten vorsortierten Listen (mit den über der Tabelle angegebenen Parametern).

$|X| = 10000$

presort: mode 1, 10 x 100000 presort steps

	$Inv(X)$	$Runs(X)$	$Rem(X)$	$Exc(X)$	$Par(X)$	$Osc'(X)$	
	24951117	5035	9807	9993	9998	16456199	
M	0.822	0.760	0.708	0.707	0.707	0.864	0.101s
M'	0.781	0.722	0.673	0.672	0.672	0.822	0.096s
P	0.969	0.895	0.835	0.833	0.833	1.018	0.119s
H	2.646	2.444	2.279	2.275	2.275	2.781	0.325s
Q	1.074	0.993	0.926	0.924	0.924	1.130	0.132s
	5119792	4964	9566	9982	8160	5010112	
M	0.990	0.746	0.696	0.693	0.707	0.993	0.099s
M'	1.000	0.753	0.703	0.700	0.715	1.003	0.100s
P	1.160	0.874	0.816	0.812	0.829	1.164	0.116s
H	2.690	2.026	1.891	1.883	1.922	2.698	0.269s
Q	1.200	0.904	0.844	0.840	0.857	1.204	0.120s
	2938397	4962	9455	9975	5198	2922761	
M	1.131	0.783	0.732	0.728	0.779	1.132	0.104s
M'	1.098	0.761	0.711	0.707	0.757	1.099	0.101s
P	1.239	0.859	0.802	0.798	0.854	1.240	0.114s
H	2.631	1.823	1.703	1.694	1.814	2.633	0.242s
Q	1.239	0.859	0.802	0.798	0.854	1.240	0.114s

(Fortsetzung)

	$Inv(X)$	$Runs(X)$	$Rem(X)$	$Exc(X)$	$Par(X)$	$Osc'(X)$	
	2070479	4954	9373	9953	3138	2054080	
M	1.116	0.731	0.683	0.679	0.769	1.117	0.097s
M'	1.093	0.716	0.669	0.665	0.753	1.094	0.095s
P	1.334	0.874	0.817	0.812	0.919	1.336	0.116s
H	2.554	1.672	1.564	1.555	1.760	2.557	0.222s
Q	1.242	0.814	0.761	0.756	0.856	1.244	0.108s
	1601610	4983	9283	9954	2339	1590064	
M	1.153	0.723	0.677	0.672	0.787	1.155	0.096s
M'	1.117	0.700	0.656	0.651	0.763	1.119	0.093s
P	1.358	0.851	0.797	0.791	0.927	1.359	0.113s
H	2.667	1.671	1.566	1.555	1.821	2.671	0.222s
Q	1.249	0.783	0.733	0.728	0.853	1.251	0.104s
	1319299	4969	9222	9957	1982	1306003	
M	1.169	0.708	0.663	0.658	0.786	1.171	0.094s
M'	1.218	0.738	0.692	0.686	0.820	1.221	0.098s
P	1.380	0.836	0.783	0.777	0.929	1.382	0.111s
H	2.660	1.612	1.510	1.498	1.790	2.665	0.214s
Q	1.268	0.768	0.720	0.714	0.853	1.270	0.102s
	1106276	5004	9150	9940	1516	1100199	
M	1.220	0.715	0.671	0.665	0.821	1.221	0.095s
M'	1.181	0.692	0.650	0.644	0.795	1.182	0.092s
P	1.463	0.858	0.805	0.798	0.986	1.465	0.114s
H	2.555	1.497	1.405	1.394	1.721	2.557	0.199s
Q	1.297	0.760	0.713	0.707	0.873	1.298	0.101s
	946251	5011	9086	9931	1373	944460	
M	1.269	0.722	0.678	0.672	0.840	1.270	0.096s
M'	1.177	0.670	0.629	0.623	0.779	1.177	0.089s
P	1.454	0.828	0.777	0.770	0.963	1.455	0.110s
H	2.631	1.497	1.406	1.394	1.742	2.632	0.199s
Q	1.256	0.715	0.671	0.665	0.832	1.256	0.095s
	833311	4999	9027	9921	1239	831322	
M	1.274	0.707	0.665	0.658	0.834	1.274	0.094s
M'	1.219	0.677	0.636	0.630	0.798	1.220	0.090s
P	1.558	0.865	0.813	0.806	1.020	1.559	0.115s
H	2.628	1.460	1.372	1.359	1.721	2.630	0.194s
Q	1.341	0.745	0.700	0.693	0.878	1.342	0.099s

(Fortsetzung)

	$Inv(X)$	$Runs(X)$	$Rem(X)$	$Exc(X)$	$Par(X)$	$Osc'(X)$	
	748408	4996	8979	9912	1181	744896	
M	1.301	0.707	0.665	0.658	0.839	1.302	0.094s
M’	1.315	0.715	0.672	0.666	0.848	1.316	0.095s
P	1.522	0.828	0.778	0.771	0.982	1.524	0.110s
H	2.671	1.453	1.366	1.352	1.722	2.674	0.193s
Q	1.342	0.730	0.686	0.680	0.866	1.344	0.097s
	674096	4989	8936	9904	865	670008	
M	1.329	0.708	0.665	0.659	0.874	1.330	0.094s
M’	1.258	0.670	0.630	0.624	0.827	1.260	0.089s
P	1.611	0.858	0.807	0.799	1.060	1.613	0.114s
H	2.657	1.415	1.331	1.317	1.748	2.661	0.188s
Q	1.357	0.723	0.680	0.673	0.892	1.359	0.096s

$|X| = 10000$

presort: mode 2, 10 x 5000 presort steps

	$Inv(X)$	$Runs(X)$	$Rem(X)$	$Exc(X)$	$Par(X)$	$Osc'(X)$	
	24951117	5035	9807	9993	9998	16456199	
M	0.822	0.760	0.708	0.707	0.707	0.864	0.101s
M’	0.814	0.752	0.701	0.700	0.700	0.856	0.100s
P	0.977	0.902	0.842	0.840	0.840	1.027	0.120s
H	2.670	2.467	2.300	2.296	2.296	2.807	0.328s
Q	1.066	0.985	0.919	0.917	0.917	1.121	0.131s
	17204626	4214	6075	6074	9999	13828672	
M	0.817	0.736	0.708	0.708	0.672	0.840	0.096s
M’	0.758	0.682	0.656	0.656	0.623	0.778	0.089s
P	1.013	0.913	0.877	0.877	0.833	1.041	0.119s
H	2.851	2.569	2.469	2.469	2.345	2.930	0.335s
Q	0.749	0.675	0.649	0.649	0.616	0.770	0.088s
	10982045	3003	3695	3691	9996	9901416	
M	0.766	0.677	0.661	0.661	0.595	0.776	0.085s
M’	0.730	0.645	0.630	0.630	0.567	0.740	0.081s
P	1.063	0.940	0.918	0.918	0.826	1.077	0.118s
H	3.045	2.693	2.630	2.630	2.366	3.086	0.338s
Q	0.604	0.534	0.521	0.521	0.469	0.612	0.067s
	7052865	2002	2246	2241	9996	6721457	
M	0.736	0.643	0.635	0.635	0.539	0.741	0.077s
M’	0.688	0.602	0.593	0.594	0.504	0.693	0.072s
P	1.128	0.986	0.973	0.973	0.826	1.135	0.118s
H	3.078	2.691	2.654	2.655	2.254	3.098	0.322s
Q	0.545	0.476	0.470	0.470	0.399	0.548	0.057s

(Fortsetzung)

	$Inv(X)$	$Runs(X)$	$Rem(X)$	$Exc(X)$	$Par(X)$	$Osc'(X)$	
	4286366	1276	1363	1358	9996	4165137	
M	0.708	0.610	0.605	0.605	0.483	0.711	0.069s
M'	0.688	0.592	0.587	0.587	0.469	0.691	0.067s
P	1.201	1.034	1.025	1.026	0.819	1.206	0.117s
H	3.233	2.783	2.760	2.761	2.205	3.247	0.315s
Q	0.472	0.406	0.403	0.403	0.322	0.474	0.046s
	2614549	779	804	799	9996	2598180	
M	0.664	0.566	0.563	0.564	0.420	0.665	0.060s
M'	0.653	0.556	0.554	0.554	0.413	0.654	0.059s
P	1.273	1.084	1.080	1.081	0.805	1.275	0.115s
H	3.289	2.800	2.788	2.791	2.079	3.292	0.297s
Q	0.476	0.405	0.404	0.404	0.301	0.477	0.043s
	1529103	468	479	475	9917	1522667	
M	0.654	0.547	0.545	0.546	0.378	0.655	0.054s
M'	0.678	0.567	0.565	0.566	0.392	0.679	0.056s
P	1.308	1.094	1.090	1.092	0.757	1.309	0.108s
H	3.197	2.675	2.666	2.669	1.849	3.200	0.264s
Q	0.497	0.415	0.414	0.414	0.287	0.497	0.041s
	888378	282	288	284	9923	881178	
M	0.696	0.569	0.567	0.568	0.364	0.697	0.052s
M'	0.736	0.602	0.600	0.601	0.385	0.737	0.055s
P	1.432	1.171	1.167	1.169	0.749	1.434	0.107s
H	3.225	2.637	2.628	2.634	1.688	3.230	0.241s
Q	0.549	0.449	0.447	0.448	0.287	0.549	0.041s
	510181	168	169	165	9995	500731	
M	0.674	0.536	0.536	0.538	0.315	0.677	0.045s
M'	0.674	0.536	0.536	0.538	0.315	0.677	0.045s
P	1.618	1.287	1.286	1.291	0.756	1.625	0.108s
H	3.267	2.598	2.595	2.606	1.526	3.280	0.218s
Q	0.584	0.465	0.464	0.466	0.273	0.587	0.039s
	250915	88	88	85	9923	251418	
M	0.690	0.523	0.523	0.526	0.273	0.690	0.039s
M'	0.690	0.523	0.523	0.526	0.273	0.690	0.039s
P	1.912	1.448	1.448	1.458	0.756	1.911	0.108s
H	3.151	2.386	2.386	2.402	1.247	3.149	0.178s
Q	0.690	0.523	0.523	0.526	0.273	0.690	0.039s

(Fortsetzung)

	$Inv(X)$	$Runs(X)$	$Rem(X)$	$Exc(X)$	$Par(X)$	$Osc'(X)$	
	128892	51	51	48	9211	129073	
M	0.683	0.480	0.480	0.486	0.226	0.682	0.032s
M'	0.683	0.480	0.480	0.486	0.226	0.682	0.032s
P	2.218	1.559	1.559	1.579	0.734	2.217	0.104s
H	3.221	2.263	2.263	2.293	1.066	3.220	0.151s
Q	0.875	0.614	0.614	0.623	0.289	0.874	0.041s

$|X| = 10000$

presort: mode 3, 10 x 50000 presort steps

	$Inv(X)$	$Runs(X)$	$Rem(X)$	$Exc(X)$	$Par(X)$	$Osc'(X)$	
	24951117	5035	9807	9993	9998	16456199	
M	0.822	0.760	0.708	0.707	0.707	0.864	0.101s
M'	0.806	0.744	0.694	0.693	0.693	0.847	0.099s
P	0.944	0.872	0.813	0.812	0.812	0.993	0.116s
H	2.678	2.474	2.307	2.303	2.303	2.816	0.329s
Q	1.066	0.985	0.919	0.917	0.917	1.121	0.131s
	8116316	4958	9666	9980	9389	7739239	
M	0.919	0.738	0.688	0.686	0.690	0.925	0.098s
M'	0.938	0.753	0.702	0.700	0.704	0.944	0.100s
P	1.088	0.874	0.815	0.812	0.817	1.095	0.116s
H	2.663	2.139	1.995	1.988	2.000	2.680	0.284s
Q	1.106	0.889	0.829	0.826	0.831	1.114	0.118s
	5192044	4961	9580	9984	8049	5092249	
M	0.968	0.731	0.682	0.679	0.694	0.971	0.097s
M'	1.008	0.761	0.710	0.707	0.723	1.011	0.101s
P	1.148	0.866	0.808	0.805	0.823	1.151	0.115s
H	2.685	2.026	1.891	1.883	1.925	2.692	0.269s
Q	1.178	0.889	0.829	0.826	0.844	1.181	0.118s
	3924251	4949	9512	9985	6461	3877412	
M	1.040	0.753	0.703	0.700	0.732	1.042	0.100s
M'	0.998	0.723	0.675	0.672	0.703	1.000	0.096s
P	1.227	0.889	0.830	0.826	0.864	1.229	0.118s
H	2.610	1.891	1.766	1.757	1.838	2.615	0.251s
Q	1.248	0.904	0.844	0.840	0.879	1.250	0.120s
	3140872	4963	9464	9978	4398	3106199	
M	1.065	0.746	0.697	0.693	0.756	1.067	0.099s
M'	1.033	0.723	0.676	0.672	0.733	1.035	0.096s
P	1.259	0.881	0.823	0.819	0.893	1.261	0.117s
H	2.625	1.838	1.717	1.708	1.862	2.630	0.244s
Q	1.140	0.798	0.746	0.742	0.809	1.142	0.106s

(Fortsetzung)

	$Inv(X)$	$Runs(X)$	$Rem(X)$	$Exc(X)$	$Par(X)$	$Osc'(X)$	
	2604624	4980	9430	9970	4019	2586059	
M	1.075	0.730	0.683	0.679	0.748	1.076	0.097s
M'	1.086	0.738	0.690	0.686	0.755	1.087	0.098s
P	1.252	0.851	0.796	0.791	0.871	1.254	0.113s
H	2.670	1.814	1.697	1.687	1.858	2.673	0.241s
Q	1.208	0.821	0.767	0.763	0.840	1.209	0.109s
	2242648	4973	9376	9972	3690	2234506	
M	1.090	0.723	0.676	0.672	0.747	1.090	0.096s
M'	1.067	0.708	0.662	0.658	0.732	1.068	0.094s
P	1.305	0.866	0.810	0.805	0.895	1.306	0.115s
H	2.577	1.709	1.599	1.589	1.767	2.578	0.227s
Q	1.294	0.858	0.803	0.798	0.887	1.295	0.114s
	1971166	4973	9340	9960	2990	1982492	
M	1.113	0.723	0.677	0.672	0.765	1.112	0.096s
M'	1.090	0.708	0.662	0.658	0.749	1.089	0.094s
P	1.334	0.866	0.810	0.805	0.917	1.332	0.115s
H	2.586	1.679	1.572	1.561	1.777	2.584	0.223s
Q	1.252	0.813	0.761	0.756	0.861	1.251	0.108s
	1758377	5009	9308	9965	2972	1751124	
M	1.159	0.737	0.691	0.686	0.782	1.159	0.098s
M'	1.100	0.700	0.656	0.651	0.742	1.100	0.093s
P	1.371	0.873	0.818	0.812	0.925	1.372	0.116s
H	2.601	1.655	1.551	1.540	1.755	2.603	0.220s
Q	1.265	0.805	0.754	0.749	0.853	1.266	0.107s
	1574094	5031	9250	9958	2165	1568130	
M	1.133	0.707	0.663	0.658	0.778	1.133	0.094s
M'	1.133	0.707	0.663	0.658	0.778	1.133	0.094s
P	1.386	0.865	0.811	0.805	0.952	1.387	0.115s
H	2.627	1.640	1.538	1.526	1.805	2.629	0.218s
Q	1.241	0.775	0.727	0.721	0.853	1.242	0.103s
	1433725	5018	9223	9949	2146	1416292	
M	1.139	0.700	0.656	0.651	0.771	1.142	0.093s
M'	1.164	0.715	0.670	0.665	0.787	1.166	0.095s
P	1.372	0.843	0.790	0.784	0.928	1.375	0.112s
H	2.634	1.617	1.517	1.506	1.782	2.639	0.215s
Q	1.237	0.760	0.713	0.707	0.837	1.240	0.101s

B.2 Osc'-Maximum

Der Aufruf `osc_max 6` (siehe Anhang A.1.2) erzeugt die nachfolgende Ausgabe.

Man sieht deutlich, daß die Elemente 1, 2 und 3 nur an den Positionen 2, 4 und 6 und die Elemente 4, 5 und 6 nur an den Positionen 1, 3 und 5 auftreten. Elemente großen und kleinen Wertes wechseln sich also ab.

```
n = 6, max Osc' = 9, # = 36
(4, 1, 5, 2, 6, 3)        (5, 2, 6, 1, 4, 3)
(4, 1, 5, 3, 6, 2)        (5, 2, 6, 3, 4, 1)
(4, 1, 6, 2, 5, 3)        (5, 3, 4, 1, 6, 2)
(4, 1, 6, 3, 5, 2)        (5, 3, 4, 2, 6, 1)
(4, 2, 5, 1, 6, 3)        (5, 3, 6, 1, 4, 2)
(4, 2, 5, 3, 6, 1)        (5, 3, 6, 2, 4, 1)
(4, 2, 6, 1, 5, 3)        (6, 1, 4, 2, 5, 3)
(4, 2, 6, 3, 5, 1)        (6, 1, 4, 3, 5, 2)
(4, 3, 5, 1, 6, 2)        (6, 1, 5, 2, 4, 3)
(4, 3, 5, 2, 6, 1)        (6, 1, 5, 3, 4, 2)
(4, 3, 6, 1, 5, 2)        (6, 2, 4, 1, 5, 3)
(4, 3, 6, 2, 5, 1)        (6, 2, 4, 3, 5, 1)
(5, 1, 4, 2, 6, 3)        (6, 2, 5, 1, 4, 3)
(5, 1, 4, 3, 6, 2)        (6, 2, 5, 3, 4, 1)
(5, 1, 6, 2, 4, 3)        (6, 3, 4, 1, 5, 2)
(5, 1, 6, 3, 4, 2)        (6, 3, 4, 2, 5, 1)
(5, 2, 4, 1, 6, 3)        (6, 3, 5, 1, 4, 2)
(5, 2, 4, 3, 6, 1)        (6, 3, 5, 2, 4, 1)
```

Die Listen der Länge 5, für die Osc' maximal wird, sind genau die vier Listen $(4, 1, 5, 2)$, $(4, 2, 5, 1)$, $(5, 1, 4, 2)$ und $(5, 2, 4, 1)$ ergänzt um das Element 3 an beliebiger Stelle, es ergeben sich $4 \cdot 5 = 20$ Möglichkeiten:

```
n = 5, max Osc' = 6, # = 20
(3, 4, 1, 5, 2)
(3, 4, 2, 5, 1)
(3, 5, 1, 4, 2)
(3, 5, 2, 4, 1)
(4, 1, 3, 5, 2)
(4, 1, 5, 2, 3)
(4, 1, 5, 3, 2)
(4, 2, 3, 5, 1)
(4, 2, 5, 1, 3)
(4, 2, 5, 3, 1)
```

```
(4, 3, 1, 5, 2)
(4, 3, 2, 5, 1)
(5, 1, 3, 4, 2)
(5, 1, 4, 2, 3)
(5, 1, 4, 3, 2)
(5, 2, 3, 4, 1)
(5, 2, 4, 1, 3)
(5, 2, 4, 3, 1)
(5, 3, 1, 4, 2)
(5, 3, 2, 4, 1)
```

Für die übrigen Längen $n \leq 12$ soll eine Zusammenfassung der Maxima und ihrer Häufigkeiten genügen:

```
n = 1, max Osc' = 0, # = 1
n = 2, max Osc' = 1, # = 1
n = 3, max Osc' = 2, # = 3
n = 4, max Osc' = 4, # = 4
n = 5, max Osc' = 6, # = 20
n = 6, max Osc' = 9, # = 36
n = 7, max Osc' = 12, # = 252
n = 8, max Osc' = 16, # = 576
n = 9, max Osc' = 20, # = 5184
n = 10, max Osc' = 25, # = 14400
n = 11, max Osc' = 30, # = 158400
n = 12, max Osc' = 36, # = 518400
```

Literatur

[AEH84] Timo O. Alanko, Hannu H. A. Erkiö, Ilkka J. Haikala: *Virtual Memory Behavior of Some Sorting Algorithms,* IEEE Transactions on Software Engineering 10 (1984), No. 4, 422–431

[CLP93] Svante Carlsson, Christos Levcopoulos, Ola Petersson: *Sublinear Merging and Natural Mergesort,* Algorithmica 9 (1993), 629–648

[Est91] Vladimir Estivill-Castro: *Sorting and Measures of Disorder,* Research Report CS-91-07, University of Waterloo, Canada, 1991

[EW89] Vladimir Estivill-Castro, Derick Wood: *A New Measure of Presortedness,* Information and Computation 83 (1989), 111–119

[Dij80] Edsger W. Dijkstra: *Some Beautiful Arguments Using Mathematical Induction,* Acta Informatica 13 (1980), 1–8

[GMPR77] L. J. Guibas, E. M. McCreight, M. F. Plaas, J. R. Roberts: *A New Representation of Linear Lists,* Proc. 9th Annual ACM Symposium on Theory of Computing (1977), 49–60

[HL88] Bing-Chao Huang, Michael A. Langston: *Practical In-Place Merging,* Communications of the ACM 31 (1988), No. 3, 348–352

[Knu73] Donald E. Knuth: *The Art of Computer Programming, Volume 3: Sorting and Searching,* Addison-Wesley, Reading, Massachusetts, 1973

[Kro69] M. A. Kronrod: *Optimal Ordering Algorithm Without Operational Field,* Soviet Math. Dokl. 10 (1969), No. 3, 744–746

[LP89] Christos Levcopoulos, Ola Petersson: *Heapsort – Adapted for Presorted Files,* Lecture Notes in Computer Science 382 (1989), 499–509

[LP93] Christos Levcopoulos, Ola Petersson: *Adaptive Heapsort,* Journal of Algorithms 14 (1993), 395–413

[LP96] Christos Levcopoulos, Ola Petersson: *Exploiting few inversions when sorting: Sequential and parallel algorithms,* Theoretical Computer Science 163 (1996), 211–238

[Man85] Heikki Mannila: *Measures of Presortedness and Optimal Sorting Algorithms,* IEEE Transactions on Computers 34 (1985), No. 4, 318–325

[MEP96] Alistair Moffat, Gary Eddy, Ola Petersson: *Splaysort: Fast, Versatile, Practical,* Software – Practice and Experience 126 (1996), No. 7, 781–797

[Par77] Luis Trabb Pardo: *Stable Sorting and Merging with Optimal Space and Time Bounds,* Siam Journal of Computing 6 (1977), No. 2, 351–372