HOWTO zur Sortierung in Perl

$Id: perl-sorting-HOWTO.txt,v 1.22 2019/11/26 19:37:07 tsbirn Exp $

Dieses Dokument beschreibt, wie in Perl das Sortieren von Arrays und Hashes durchgefÃ¼hrt wird.

INHALTSVERZEICHNIS

---

1) Array sortieren

Die Perl-Funktion "sort" sortiert Arrays standardmÃ¤Ã-^_ig "alphabetisch aufsteigend" ("lexikografisch", etwa wie im Telefonbuch oder in einem Lexikon), indem sie die Ordnungsreihenfolge von jeweils 2 Elementen des Arrays mit Hilfe der String-Vergleichsoperation "cmp" (compare) ermittelt:

```
  @sorted = sort @array;
```

Die Standard-Vergleichs-Funktion "cmp" kann auch explizit als "anonyme Funktion" direkt nach "sort" angegeben werden, folgender Aufruf ist also vÃ¶llig identisch zu obigem:

```
  @sorted = sort { $a cmp $b } @array;
```

Alternativ kann die Vergleichs-Funktion explizit definiert und ihr Name beim "sort"-Aufruf mitgegeben werden. Folgender Aufruf ist also wiederum vÃ¶llig identisch:

```
  sub cmp_func { $a cmp $b }
  @sorted = sort cmp_func @array;
```

1.1) Wie arbeitet "sort"?

Die zwei von Perl fest per Name "$a" und "$b" in der Vergleichs-Funktion verwendeten Variablen zeigen automatisch auf die 2 Elemente des Arrays, die gerade miteinander verglichen werden sollen. Eigentlich sind es aus PerformancegrÃ¼nden ALIASE fÃ¼r die Arraylemente (also weitere Namen). Eine Zuweisung an "$a" oder "$b" Ã¤ndert also das entsprechende Element im Array --> keine gute Idee!).

Die Funktion "sort" ruft auf geschickte Art und Weise (Quicksort-Verfahren) solange die Vergleichs-Funktion mit je 2 Array-Elementen auf, bis ALLE Elemente gemÃ¤Ã-^_ der Vergleichs-Funktion sortiert sind. Die Vergleichs-Funktion muss folgende Ergebnisse fÃ¼r die beiden Element-Aliase "$a" und "$b" zurÃ¼ckliefern:

Erg	Bedeutung
1	Falls "$a" grÃ¶Ã-^_er "$b"
0	Falls "$a" gleich "$b"
-1	Falls "$a" kleiner "$b"

Um zuzusehen, welche Elemente die Perl-Funktion "sort" in welcher Reihenfolge vergleicht, baut man eine "print"-Anweisung in die Vergleichs-Funktion ein. Dann sieht man pro Aufruf der Vergleichs-Funktion durch "sort" die beiden gerade miteinander verglichenen Elemente (man sieht auch, dass viele Elemente mehrfach an die Vergleichs-Funktion Ã¼bergeben werden).

```
  sub cmp_func
```

```
  {
      print "cmp_func: '$a' cmp '$b' -> ", ($a cmp $b), "\n";
      $a cmp $b;
  }
```

1.2) Array absteigend und numerisch sortieren

StandardmÃ¤ÃM-^_ig sortiert "sort" wie gesagt aufsteigend alphabetisch. Vertauscht man "$a" und "$b", so wird "umgekehrt" (d.h. normalerweise "alphabetisch absteigend") sortiert, d.h. man spart den Aufruf der Funktion "reverse" ein.

```
  @rev_sorted =         sort { $b cmp $a } @array;     # Umgekehrt sortieren
  @rev_sorted = reverse sort { $a cmp $b } @array;     # Analog (langsamer?)
```

Durch Austausch der Element-Vergleichfunktion "cmp" gegen "<=>" kann auch "numerisch aufsteigend" (oder "absteigend") sortiert werden:

```
  @sorted     = sort { $a <=> $b } @array;   # Numerisch aufsteigend
  @rev_sorted = sort { $b <=> $a } @array;   # Numerisch absteigend
```

2) Hash(-SchlÃ¼ssel) sortieren

Auch die Sortierung von Hash-SchlÃ¼sseln nach ihrem zugehÃ¶rigen Hash-Wert ist mÃ¶glich, um auf die Hash-Elemente in einer nach den Hash-Werten sortierten Reihenfolge zugreifen zu kÃ¶nnen:

```
  %hash = ("tom" => 17, "hans" => 35, "rick" => 5, "helmut" => 99);

  # Hash-SchlÃ¼ssel alphabetisch sortieren
  @keys_sorted_by_value = sort { $hash{$a} cmp $hash{$b} } keys %hash;
  foreach (@keys_sorted_by_value) {
      print "$_ hat Wert $hash{$_}\n";
  }

  # Hash-SchlÃ¼ssel numerisch sortieren
  @keys_sorted_by_value = sort { $hash{$a} <=> $hash{$b} } keys %hash;
```

Der Trick ist hier, die Vergleichs-Funktion fÃ¼r 2 Hash-SchlÃ¼ssel auf Basis der ihnen zugeordneten Hash-Werte zu definieren. D.h. bei jedem Vergleich zweier Hash-SchlÃ¼ssel wird auf die Ihnen zugeordneten Hash-Werte zugegriffen. Also werden die Hash-SchlÃ¼ssel gemÃ¤ÃM-^_ den ihnen zugeordneten Hash-Werten sortiert.

3) Hierarchisch sortieren

Sogar nach mehreren Kriterien kann "hierarchisch" sortiert werden (hier erst ohne Beachtung der GROSS/Kleinschreibung, wenn die Elemente dabei gleich sind, dann mit Beachtung der GROSS/Kleinschreibung):

```
  @sorted = sort { uc($a) cmp uc($b) or $a cmp $b } @array;    # uc=uppercase
  @sorted = sort { "\L$a" cmp "\L$b") or $a cmp $b } @array;   # \L=lowercase
```

Liefert ein Kriterium keinen Unterschied (Ergebnis "0"), dann wird aufgrund der "or"-VerknÃ¼pfung nach dem nÃ¤chsten Kriterium sortiert. Liefert ein Kriterium einen Unterschied, dann bricht die Auswertung aufgrund der "Shortcut/Short circuit"-Eigenschaft der Auswertung von Booleschen AusdrÃ¼cken an dieser Stelle mit dem Ergebnis dieses Vergleichs ("1" oder "-1") ab. Ergibt keines der Sortierkriterien einen Unterschied, dann wird "0" zurÃ¼ckgeliefert (d.h. die Werte sind nicht unterscheidbar und damit "gleich").

TIP: In einer Vergleichs-Funktion im letzten Vergleichs-Schritt immer die Elemente direkt vergleichen, damit eine "eindeutige" ("stabile") Sortierung entsteht, die bei wiederholter Sortierung reproduziert wird:

```
  sub cmp_2_levels {
      $alter{$a} <=> $alter{$b} or
              $a cmp $b;
  }
  sort cmp_2_levels keys %alter;
```

3.1) Array hierarchisch sortieren

Folgendes Array enthÃ¤lt drei Werte pro Element (Zahl + GroÃM-^_buchstabe + Zeichen), die Sortierreihenfolge wird primÃ¤r vom 1. Wert bestimmt. Bei gleichem 1. Wert vom 2. Wert. Und bei gleichem 1.+2. Wert vom 3. Wert (Trennzeichen der Werte ist "_"):

```
  @arr = qw/ 1_D_100  20_A_200  3_B_8  20_A_-3 20_A_33 -5_B_49  1000_C_-40 /;
```

Zum Sortieren wird eine mehrstufige Vergleichs-Funktion definiert:

```
  @sorted = {
      # print "VERGLEICH: $a <=> $b", ($a[0] <=> $b[0] or
      #                                $a[1] cmp $b[1] or
      #                                $a[2] <=> $b[2]), "\n";
      my @a = split(/_/, $a);
      my @b = split(/_/, $b);
      $a[0] <=> $b[0] or
      $a[1] cmp $b[1] or
      $a[2] <=> $b[2] or
         $a cmp $b;
  } @arr;
```

Mit der "print"-Anweisung kann der Sortiervorgang beobachtet werden. Ausgegeben werden jeweils zwei Elemente, die miteinander verglichen werden und das Ergebnis -1 ($a kleiner $b), 0 ($a gleich $b) oder 1 ($a grÃ¶Ã-^_er $b).

UnschÃ¶n an dieser LÃ¶sung ist die mehrfache Zerlegung der Arrayelemente in die Einzelwerte, falls das gleiche Arrayelemente mehrfach verglichen wird. Dies wird durch die "Schwartzsche Transformation" weiter unten verbessert.

3.2) Hierarchisch sortieren bgzl. mehrerer getrennter Kriterien

In folgendem Beispiel sind Benutzer-IDs einer Bibliothek gemÃ¤Ã-^_ einiger Sortierkriterien zu sortieren. Ist bzgl. eines Kriteriums kein Unterschied vorhanden, wird das nÃ¤chste betrachtet. Zuletzt werden die Benutzer-IDs direkt verglichen, die auf jeden Fall unterschiedlich sind, damit eine "stabile Sortierung" entsteht. Die Kriterien sind:

1) Gezahlte GebÃ¼hr (Funktionsaufruf "&gebuehr" mit Benutzer-ID)
2) Anzahl geliehener BÃ¼cher (Hash "%buecher")
3) Nachname (Hash "%name")
4) Vorname (Hash "%vorname")
5) Benutzer-ID

```
  @benutzer_ids = sort {
      &gebuehr($a) <=> &gebuehr($b) or
      $buecher{$a} <=> $buecher{$b} or
         $name{$a} cmp $name{$b}    or
      $vorname{$a} cmp $vorname{$b} or
                $a <=> $b
  } @benutzer_ids;
```

In der Vergleichs-Funktion wird also eine Funktion "&gebuehr" aufgerufen und auf drei Hashes "%buecher", "%name" und "%vorname" zugegriffen, die jeweils die Benutzer-ID als Argument Ã¼bergeben bekommen.

4) Gemischte/Kombinierte Daten sortieren

4.1) Mischung aus Texten und Zahlen sortieren

Enthalten die zu sortierenden Daten Text und Zahlen gemischt (z.B. numerische Kapitelnummern und alphabetische Anhangnummern), dann kann man sich z.B. entscheiden fÃ¼r die Sortierung der Texte NACH den Zahlen:

```
  @kapitel_nr = qw/ 4 8 9 B 1 3 2 10 6 C 12 0 5 7 11 A /;

  @mixsort = sort {
      ($a =~ /^\d+$/) ?       # $a ist eine Zahl?
          ($b =~ /^\d+$/) ?   # $b ist eine Zahl?
              $a <=> $b :     # BEIDES --> Ergebnis Zahlenvergleich
              -1              # NEIN   --> -1 = Zahl kleiner (=VOR) Buchstabe
      :                       # $a ist KEINE Zahl!
          ($b !~ /^\d+$/) ?   # $b ist KEINE Zahl?
              $a cmp $b :     # BEIDES --> Ergebnis Textvergleich
              1               # NEIN   --> 1 = Buchstabe grÃ¶sser (=NACH) Zahl
    } @kapitel_nr;            # --> 0 1 2 3 4 5 6 8 9 10 11 A B C
```

Eine Sortierung der Texte VOR den Zahlen erhÃ¤lt man mit:

```
  @kapitel_nr = qw/ 4 8 9 B 1 3 2 10 6 C 12 0 5 7 11 A /;

  @mixsort = sort {
      ($a =~ /^\d+$/) ?       # $a ist eine Zahl?
          ($b =~ /^\d+$/) ?   # $b ist eine Zahl?
              $a <=> $b :     # BEIDES --> Ergebnis Zahlenvergleich
              1               # NEIN   --> 1 = Zahl grÃ¶Ã-^_er (=NACH) Buchstabe
      :                       # $a ist KEINE Zahl!
```

```
          ($b !~ /^\d+$/) ?   # $b ist KEINE Zahl?
              $a cmp $b :     # BEIDES --> Ergebnis Textvergleich
              -1              # NEIN   --> -11 = Buchstabe kleiner (=VOR) Zahl
    } @kapitel_nr;            # --> A B C 0 1 2 3 4 5 6 8 9 10 11
```

4.2) Kombination aus Text und Zahl sortieren

Enthalten die zu sortierenden Daten eine KOMBINATION aus Text + Zahlen (z.B. Bauteilbezeichnungen A1 A5 B33 C160 X0), dann kann man sich z.B. entscheiden fÃ¼r eine Sortierung erst nach dem Textanteil und dann nach dem Zahlenanteil:

```
  @bauteile = qw/ A4 B8 A9 B0 A1 A3 AA5 B2 R9 A10 Z6 R11 ZZ123 /;

  @mixsort = sort {
      my ($at, $an) = ($a =~ m/(\D+)(\d+)/);   # Wiederholungsfaktor + = mind. 1
      my ($bt, $bn) = ($b =~ m/(\D+)(\d+)/);   # Wiederholungsfaktor + = mind. 1

      $at cmp $bt or $an <=> $bn;   # Erst nach Text, dann nach Zahl anordnen
    } @bauteile; # --> A1 A3 A4 A9 A10 AA5 B0 B2 B8 R9 R11 Z6 ZZ123
```

Sollen auch Bauteilnamen OHNE Zahlenanteil akzeptiert werden (z.B. A XX Z), muss zusÃ¤tzlich ein leerer Zahlenanteil akzeptiert werden:

```
  @bauteile = qw/ A4 XXX B8 A9 B0 A A1 A3 AA5 B2 R9 A10 Z6 VV R11 R ZZ123 /;

  @mixsort = sort {
      my ($at, $an) = ($a =~ m/(\D+)(\d*)/);   # Wiederholungsfaktor * statt +!
      my ($bt, $bn) = ($b =~ m/(\D+)(\d*)/);   # Wiederholungsfaktor * statt +!

      $at cmp $bt or $an <=> $bn;   # Erst nach Text, dann nach Zahl anordnen
    } @bauteile; # --> A A1 A3 A4 A9 A10 AA5 B0 B2 B8 R R9 R11 VV XXX Z6 ZZ123
```

Sollen auch Bauteilnamen ohne Buchstabenanteil bzw. ohne Zahlenanteil akzeptiert werden (z.B. 1 15 600 A XX Z), muss zusÃ¤tzlich sowohl ein leerer Buchstabenanteil als auch ein leerer Zahlenanteil akzeptiert werden:

```
  @bauteile = qw/ 1 A4 XXX 600 B8 A9 B0 A A1 A3 AA5 B2 R9 A10 Z6 15 VV R11 R ZZ123 /;

  @mixsort = sort {
      my ($at, $an) = ($a =~ m/(\D*)(\d*)/);   # Wiederholungsfaktor * statt +!
      my ($bt, $bn) = ($b =~ m/(\D*)(\d*)/);   # Wiederholungsfaktor * statt +!

      $at cmp $bt or $an <=> $bn;   # Erst nach Text, dann nach Zahl anordnen
    } @bauteile; # --> 1 15 600 A A1 A3 A4 A9 A10 AA5 B0 B2 B8 R R9 R11 VV XXX Z6 ZZ123
```

5) Schwartzsche Transformation

Ein vom Perl-Guru Randal L. Schwartz erfundener Trick (bzw. inzwischen ein sogenanntes "Perl Idiom"), um eine Liste von "Dingen" gemÃ¤Ã-^_ einem Kriterium zu sortieren, das eine recht "teure" Funktion des "Dings" ist (d.h. das pro "Ding" aufwendig zu berechnen ist).

Beispielsweise ist die Sortierung einer Liste von Dateien nach ihrem Alter (-M = modification time) teuer, da pro Vergleich zwei Dateisystemzugriffe erfolgen:

```
  @sortiert = sort { -M $a <=> -M $b } @dateien;
```

Grund: GemÃ¤Ã-^_ dem weiter oben beschriebenen Verfahren, wie "sort" arbeitet, wird jedes Elemente eines Arrays mehrfach an die Vergleichs-Funktion Ã¼bergeben, die Berechung des Sortierkriteriums erfolgt also "mehrfach pro Ding".

Diesen Aufwand kann man auf den minimal notwendigen Aufwand reduzieren, indem man das Sortierkriterium pro "Ding" genau 1x berechnet, zusammen mit dem "Ding" zwischenspeichert und die Sortierung der "Dinge" auf Basis dieser Zwischenwerte durchfÃ¼hrt. Man opfert also Speicherplatz, um Laufzeit zu sparen.

Dazu wandelt man das ursprÃ¼ngliche 1-dimensionale Array "@dateien" in ein 2-dimensionales Array um, das aus einer Liste von (anonymen) Array-Referenzen auf die Paare ("Ding", Sortierkriterium) besteht. Pro "Ding" wird dabei 1x das Sortierkriterium berechnet (hier das Dateialter):

```
  @refliste = map { [ $_, -M ] } @dateien;
```

Danach sortiert man diese Referenzen nach dem darin als 2. Element gespeicherten Sortierkriterium:

```
  @refsortiert = sort { $a->[1] <=> $b->[1] } @refliste;
```

Und am Schluss wirft man das Sortierkriterium wieder weg und behÃ¤lt nur die ursprÃ¼nglichen "Dinge" wieder Ã¼brig:

```
  @sortiert = map { $_->[0] } @refsortiert;
```

Diese 3 Schritte fÃ¼hrt man normalerweise ohne die obigen Zwischenvariablen @refliste und @refsortiert "auf einen Schlag" durch und nennt das Ganze dann nach ihrem Erfinder "Schwartzsche Transformation":

```
  @sortiert =  map { $_->[0] }
              sort { $a->[1] <=> $b->[1] }
               map { [ $_, -M ] } @dateien;
```

5.1) Beispiele zu Schwartzscher Transformation

Hier die Sortierung des obigen Arrays mit drei Werten pro Element per Schwartzscher Transformation. In diesem Fall werden Arrayreferenzen auf 4-elementige Arrays mit Originalwert + seine drei Teilwerte sortiert:

```
  @arr = qw/ 1_D_100  20_A_200  3_B_8  20_A_-3 20_A_33 -5_B_49  1000_C_-40 /;

  @sorted =  map { $_->[0] }
            sort { $a->[1] <=> $b->[1] or
                   $a->[2] cmp $b->[2] or
                   $a->[3] <=> $b->[3] }
             map { [ $_, split(/_/, $_) ] } @arr;
```

Und hier die Sortierung des obigen Arrays mit Benutzer-IDs per Schwartzscher Transformation. In diesem Fall werden Arrayreferenzen auf 5-elementige Arrays mit Originalwert + vier Sortierkriterien sortiert:

```
  @sorted =  map { $_->[0] }
            sort { $a->[1] <=> $b->[1] or
                   $a->[2] <=> $b->[2] or
                   $a->[3] cmp $b->[3] or
                   $a->[4] cmp $b->[4] or
                   $a->[0] <=> $b->[0] }
             map { [ $_, &gebuehr($_), $buecher{$_}, $name{$_}, $vorname{$_} ] }
             @benutzer_ids;
```