10.1 KOMPLEXITÄT BEIM SORTIEREN

EINFÜHRUNG

Computer sind weit mehr als Numbercrunchers, also reine Rechenmaschinen zur Zahlenverarbeitung Vielmehr ist bekannt, dass ein beträchtlicher Teil der Laufzeit aller weltweit in Betrieb stehender Compute für das Sortieren und Suchen von Daten verwendet wird. Es ist darum wichtig, dass ein Programm nicht nu die richtigen Daten liefert, sondern auch optimiert wird. Dies betrifft:

- seine Länge
- seine Struktur und Übersichtlichkeit
- seine Laufzeit
- seinen Speicherbedarf

Grundsätzlich gilt, dass die Optimierung bereits zu Beginn der Problemlösung mit einbezogen werde sollte, denn es ist meist schwierig, ein salopp geschriebenes Programm im Nachhinein zu optimieren.

In diesem Kapitel untersuchst du die Laufzeitoptimierung beim Sortieren von Daten. Dabei wirst du auc Grenzen der Informatik und des Computereinsatzes kennen lernen, denn ein Problem, für das es wol einen algorithmischen Lösungsweg gibt, der schnellste Computer aber hunderte von Jahren zur Lösun benötigt, gilt als **unlösbares Problem.**

PROGRAMMIERKONZEPTE:*Komplexität, Laufzeit, Ordnung von Algorithmen, Sortierverfahren, Überlade von Operatoren*

SORTIEREN WIE KINDER: CHILDREN SORT

Das Sortieren bzw. Ordnen einer Menge von Objekten, für die es die Vergleichsoperationen *grösser*, *kleine* und *gleich* gibt [mehr...] , ist und bleibt eine Standardaufgabe der Informatik. Obschon du in allen gängige höheren Programmiersprachen Bibliotheksroutinen findest, mit denen du sortieren kannst, gehören di Konzepte des Sortierens zu deinem Standardwissen, denn es gibt immer wieder Situationen, wo du da Sortieren selbst implementieren oder optimieren musst.

Eine Ansammlung unsortierter Objekte wird als eine Menge bezeichnet. Im Computer werden die Objekt aber in einer eindimensionalen Datenstruktur gespeichert, wozu sich eine Liste besonders gut eigne [mehr...].

Im Programm betrachtest du Zwerge als Actors der Gamebibliothek *JGameGrid*. Du kannst sehr einfac ihre Spritebilder in einem Gitter darstellen [mehr...]. Die Höhe der Spritebilder (in Pixel) dienen dir a Mass für die Körpergrösse.

Oft werden Algorithmen direkt aus Verfahren übernommen, die man auch im täglichen Leben anwende Fragt man Kinder, wie sie eine Menge von Objekten der Grösse nach ordnen, so beschreiben sie da Verfahren oft so: "*Du nimmst dir das kleinste (oder grösste) Objekt und setzt es der Reihe nach hin* Dieses Lösungsverfahren klingt sehr plausibel, ist aber für den Computer ein Problem, denn er kann da kleineste oder grösste Objekte nicht wie wir Menschen auf einen Blick erfassen. Er muss es in de unsortierten Liste zuerst suchen, indem er der Reihe nach alle Objekte durchläuft und die Objekt

miteinander vergleicht. Um das Sortierverfahren, hier **Children Sort** genannt zu implementieren, benötig du eine Funktion *getSmallest(row)*, die von der übergebenen Liste den kleinsten Zwerg zurückliefert. Dab gehst du wie folgt vor:

Du speicherst das erste Listenelement in der Variablen *smallest* und durchläufst in einer for-Schleife al nachfolgenden Elemente. Ist das gerade betrachtete Element kleiner als *smallest*, so ersetzt du *smalle* durch dieses Element.

Beim Chidren Sort verwendest du zwei Listen, eine Liste *startList* mit den gegebenen Objekten und d Liste *targetList*, die vorerst leer ist. Du suchst in *startList* das kleinste Element, nimmst es dort heraus un fügst es hinten in die *targetList* an, bis *startList* leer ist.

```
from gamegrid import *
from random import shuffle

def bodyHeight(dwarf):
    return dwarf.getImage().getHeight()

def updateGrid():
   removeAllActors()
   for i in range(len(startList)):
       addActor(startList[i], Location(i, 0))
   for i in range(len(targetList)):
       addActor(targetList[i], Location(i, 1))

def getSmallest(li):
    global count
    smallest = li[0]
    for dwarf in li:
        count += 1
        if bodyHeight(dwarf) < bodyHeight(smallest):
            smallest = dwarf
    return smallest

n = 7

makeGameGrid(n, 2, 170, Color.red, False)
setBgColor(Color.white)
show()

startList = []
targetList = []

for i in range(0 , n):
    dwarf = Actor("sprites/dwarf" + str(i) + ".png")
    startList.append(dwarf)
shuffle(startList)
updateGrid()
setTitle("Children Sort. Press <SPACE> to sort...")
count = 0
while not isDisposed() and len(startList) > 0:
    c = getKeyCodeWait()
    if c == 32:
        smallest = getSmallest(startList)
        targetList.append(smallest)
        startList.remove(smallest)
        count += 1
        setTitle("Count: " + str(count) + " <SPACE> for next step...")
        updateGrid()
setTitle("Count: " + str(count) + " All done")
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Beim Children Sort brauchst du neben der gegebenen unsortierten Liste der Länge n eine zweite Liste, di schliesslich auch die Länge n hat. Ist n sehr gross, kann dies zu einem Speicherplatzproblem werde [mehr...].

Du kannst dir leicht überlegen, wieviele elementare Schritte zur Lösung nötig sind: Unabhängig davon, wi die Objekte in der vorgegebenen Liste angeordnet sind, musst du sie zur Suche des Minimums zuerst mal, dann n-1 mal, usw. durchlaufen; dazu kommt jedesmal die Verschiebungsoperation von der Startlist in die Zielliste. Die Anzahl Operationen c ist daher die Summe aller natürlichen Zahlen von 2 bis n + 1, wi du auch mit der Zählvariablen count mitverfolgen kannst. Aus der Summenformel für natürliche Zahle ergibt sich :

$$c = \frac{(n+1)*(n+2)}{2} - 1 = \frac{n^2}{2} - \frac{3n}{2}$$

Beispielsweise ergeben sich für n = 1000 bereits

$$c = \frac{1000*1000}{2} + \frac{3*1000}{2} = 500000 + 1500 \approx 500000$$

Schritte. Wie du siehst, überwiegt für grosse n das quadratische Glied und man sagt darum:
Die Komplexität des Algorithmus ist von der Ordnung n-Quadrat
und schreibt dafür

Komplexität = $O(n^2)$

SORTIEREN BEIM KARTENSPIEL: INSERTION SORT

Nimmst du beim Ausspielen von Spielkarten Karte um Karte fächerartig in die Hand, so verwendest du meist intuit ein anderes Sortierverfahren: Du fügst jede neu aufgenommene Karte dort in die Hand ein, wo sie gemäss ihre Wertigkeit hineinpasst. Im deinem Programm, dass die ungeordneten Karten von der Startliste (dem Kartenstape in die Zielliste (deine Hand) einfügt, gehst du genau so vor:

Du nimmst Karte um Karte von links nach rechts aus der Startliste und durchläuft die bereits geordnete Zielliste ebenfalls von links nach rechts. Sobald die aufgenommene Karte höhere Wertigkeit als die zuletzt in der Han betrachtete ist, fügst du sie in die Zielliste ein.

```
from gamegrid import *
from random import shuffle

def cardValue(card):
    return card.getImage().getHeight()

def updateGrid():
   removeAllActors()
   for i in range(len(startList)):
       addActor(startList[i], Location(i, 0))
   for i in range(len(targetList)):
       addActor(targetList[i], Location(i, 1))
```

```
n = 9

makeGameGrid(n, 2, 130, Color.blue, False)
setBgColor(Color.white)
show()

startList = []
targetList = []

for i in range(0 , 9):
    card = Actor("sprites/" + "hearts" + str(i) + ".png")
    startList.append(card)

shuffle(startList)
updateGrid()
setTitle("Insertion Sort. Press <SPACE> to sort...")
count = 0

while not isDisposed() and len(startList) > 0:
    getBg().clear()
    c = getKeyCodeWait()
    if c == 32:
        pick = startList[0] # take first
        startList.remove(pick)
        i = 0
        while  i < len(targetList) and cardValue(pick) > cardValue(targetList[i]):
            i += 1
            count += 1
        targetList.insert(i, pick)
        count += 1
        setTitle("Count: " + str(count) + " <SPACE> for next step...")
        updateGrid()
setTitle("Count: " + str(count) + " All done")
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

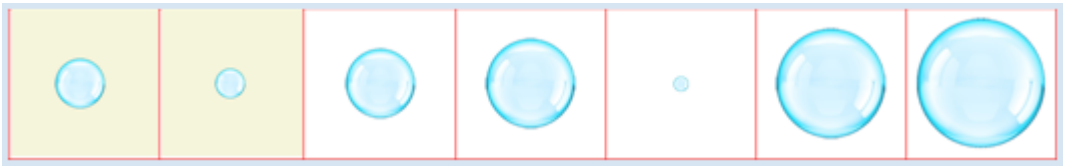
MEMO

Dieses Sortierverfahren heisst **Sortieren durch Einfügen** (**insertion sort**). Die benötigte Anzahl Schritt hängt dabei von der Reihenfolge der Karten im aufgenommen Kartenstapel ab. Am meisten Schritt braucht es, wenn der Kartenstapel zufälligerweise gerade umgekehrt geordnet ist. Man kann sic überlegen oder mit einer Computersimulation herausfinden, dass die Zahl der Schritte im Mittel (für gross n) mit $n^2 / 4$ zunimmt, die Komplexität im Mittel also wie beim Chidren Sort ebenfalls $O(n^2)$ ist.

SORTIEREN MIT LUFTBLASEN: BUBBLE SORT

Eine bekannte Art, Objekt in einer Liste zu sortieren, besteht darin, die Liste von links nach recht mehrmals zu durchlaufen und immer zwei nebeneinander liegende Elemente zu vertauschen, falls sie i falscher Reihenfolge liegen.

Mit diesem Verfahren, bewegt sich zuerst das grösste Element sukzessive von links nach rechts, bis es do angekommen ist. Im nächsten Durchlauf beginnst du wieder links, gehst aber nur bis zum zweitletzte Element, da sich ja das grösste bereits an der richtigen Stelle befindet. Bei diesem Verfahren ist kein zweite Liste nötig [mehr...].



```
from gamegrid import *
from random import shuffle

def bubbleSize(bubble):
    return bubble.getImage().getHeight()

def updateGrid():
   removeAllActors()
   for i in range(len(li)):
       addActor(li[i], Location(i, 0))

def exchange(i, j):
    temp = li[i]
    li[i] = li[j]
    li[j] = temp

n = 7
li = []

makeGameGrid(n, 1, 150, Color.red, False)
setBgColor(Color.white)
show()
for i in range(0 , n):
    bubble = Actor("sprites/bubble" + str(i) + ".png")
    li.append(bubble)
shuffle(li)
updateGrid()
setTitle("Bubble Sort. Press <SPACE> for next step...")
k = n - 1
i = 0
count = 0
while not isDisposed() and k > 0:
    getBg().fillCell(Location(i, 0), makeColor("beige"))
    getBg().fillCell(Location(i + 1, 0), makeColor("beige"))
    refresh()
    c = getKeyCodeWait()
    if c == 32:
        count += 1
        bubble1 = li[i]
        bubble2 = li[i + 1]
        refresh()
        if bubbleSize(bubble1) > bubbleSize(bubble2):
            exchange(i, i + 1)
            setTitle("Last Action: Exchange. Count: " + str(count))
        else:
            setTitle("Last Action: No Exchange. Count: " + str(count))
        getBg().clear()
        updateGrid()
        if i == k - 1:
            k = k - 1
            i = 0
        else:
            i += 1
getBg().clear()
```

```
refresh()
setTitle("All done. Count: " + str(count))
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die grösseren Elemente bewegen sich dabei nach rechts (sozusagen wie Luftblasen in Wasser nach oben Aus diesem Grund heisst dieser Sortieralgorithmus **Bubble Sort**. Wie du überlegen kannst oder a eingebauten Schrittzähler siehst, ist seine Komplexität unabhängig von der Anordnung der Element in d vorgegebenen Liste wieder von der Ordnung $O(n^2)$.

Zur Bereicherung der Demonstration werden die beiden Zellen, deren Blasen als letztes verglichen wurde mit der Background-Methode *fillCell()* farbig hinterlegt. Mit *getBg().clear()* kann die Hintergrundfarbe wied entfernt werden. Der Aufruf von *refresh()* ist nötig, damit das Bild neu auf dem Bildschirm gerendert wird.

MIT BIBLIOTHEKSROUTINEN SORTIEREN: TIMSORT

Da das Sortieren zu den wichtigsten Algorithmen gehört, stellen alle höheren Programmiersprache eingebaute Bibliotheksfunktionen zum Sortieren zur Verfügung. In Python handelt es sich um die Funktio *sorted(liste, cmp)*, die sogar zu den eingebauten Funktionen gehört, also ohne import verwendet werde kann. Du kannst dir also deinen selbstgeschriebenen Sortieralgorithmus ersparen, dafür musst du ab lernen, wie die Bibliotheksfunktion verwendet wird. Offensichtlich benötigt sie als Parameter die z sortierende Liste. Du musst ihr aber auch noch mit einem zweiten Parameter die Information mitgeb nach welchem Ordnungskriterium sie die Objekte sortieren soll.

Das Sortierkriterium legst du in einer Funktion fest, die du hier mit *compare()* bezeichnest. Diese muss a Parameter die zwei Objekte erhalten und drei Werte 1, 0 und -1 zurückgeben, je nachdem ob das erst Objekt grösser, gleich oder kleiner dem zweiten Objekt ist. Der Bibliotheksfunktion *sorted()* übergibst d den frei gewählten Funktionsnamen als zweiten Parameter oder mit dem benannten Parameter *cmp*.

```
from gamegrid import *
from random import shuffle

def bodyHeight(dwarf):
    return dwarf.getImage().getHeight()

def compare(dwarf1, dwarf2):
    if bodyHeight(dwarf1) < bodyHeight(dwarf2):
        return -1
    elif bodyHeight(dwarf1) > bodyHeight(dwarf2):
        return 1
    else:
        return 0

def updateGrid():
   removeAllActors()
   for i in range(len(li)):
       addActor(li[i], Location(i, 0))

n = 7
li = []

makeGameGrid(n, 1, 170, Color.red, False)
setBgColor(Color.white)
show()
for i in range(0 , n):
    dwarf = Actor("sprites/dwarf" + str(i) + ".png")
    li.append(dwarf)
shuffle(li)
```

```
updateGrid()
setTitle("Timsort. Press any key to get result...")
getKeyCodeWait()
li = sorted(li, cmp = compare)
updateGrid()
setTitle("All done.")
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Willst du Bibliotheksfunktionen zum Sortieren verwenden, so muss du mit einer Vergleichsfunktio festlegen, wie zwei Elemente auf *grösser, gleich* und *kleiner* verglichen werden [mehr...].

Der in Python verwendete Algorithmus wurde erst 2002 von Tim Peters erfunden und heisst darum *Timsor* Er hat (im Mittel) die Ordnung O(nlog(n)). Es sind also beispielsweise für $n = 10^6$ nur rund 10^7 Operatione nötig, statt der rund 10^{12} bei einem Sortieralgorithmus mit der Ordnung $O(n^2)$.

AUFGABEN

1. Sortiere die 7 Zwerge mit einem Bubble Sort.

2. Füge das Spritebild *snowwhite.png* von Schneewittchen, das dieselbe Grösse wie der grösste Zwe besitzt, in den Bubble Sort von Aufgabe 1 hinzu. Zeige, dass die Reihenfolge von Schneewittchen un dem grössten Zwerg immer ihrer Reihenfolge in der Startliste einspricht. (Einen solche Sortieralgorithmus nennt man **stabil**.)

3. Mit *row = range(n)* und nachfolgendem *random.shuffle(row)* kannst du sehr einfach lange unsortier Zahlenlisten erzeugen. Messe die Laufzeit für das Sortieren mit dem internen Sortieralgorithmu (*Timsort*) für verschiedene Werte von n und zeige, dass die Komplexität wesentlich besser als $O(n^2$ ist. Anleitung: Um eine Zeitdifferenz zu messen, importierst du das Modul *time* und bildest die Differer von zwei Aufrufen von *time.clock()*.

ZUSATZSTOFF

ÜBERLADEN DER VERGLEICHSOPERATOREN

Das Vergleichen von zwei Objekten ist eine wichtige Operation. Für Zahlen kannst du dazu die Vergleichsoperationen <, <=, ==, >, = > verwenden. In Python ist es möglich, diese Operatoren auch fü irgend einen anderen Datentyp einzusetzen, also beispielsweise für Zwerge. Dadurch gewinnt de Programmcode an Eleganz und Übersichtlichkeit.

Du gehst wie folgt vor:

Definiere in der Klassendefinition deines Datentyps die Methoden *__lt__(), __le__(), __eq__(), __ge__(__gt__(),* die den booleschen Werte der Vergleichsoperation *less, less-and-equal, equal, greater-and-equa greater* zurückgeben. Zusätzlich kannst du noch die Methode *__str()__* definieren, die beim Aufruf der *str* Funktion verwendet wird. Dies hat allerdings mit dem Sortieren nichts zu tun.

In der Klasse *Dwarf, d*ie von *Actor* abgeleitet ist, speicherst du als Instanzvariable zusätzlich noch de Namen des Zwergs, den du bei *updateGrid()* als *TextActor* ausschreibst.

```
from gamegrid import *
from random import shuffle

class Dwarf(Actor):
    def __init__(self, name, size):
        Actor.__init__(self, "sprites/dwarf" + str(size) + ".png")
        self.name = name
        self.size = size
    def __eq__(self, a):  # ==
        return self.size == a.size
    def __ne__(self, a): # !=
        return self.size != a.size
    def __gt__(self, a): # >
        return self.size > a.size
    def __lt__(self, a): # <
        return self.size < a.size
    def __ge__(self, a): # >=
        return self.size >= a.size
    def __le__(self, a): # <=
        return self.size <= a.size
    def __str__(self):  # str() function
        return self.name

def compare(dwarf1, dwarf2):
    if dwarf1 < dwarf2:
        return -1
    elif dwarf1 > dwarf2:
        return 1
    else:
        return 0

def updateGrid():
   removeAllActors()
   for i in range(len(row)):
       addActor(row[i], Location(i, 0))
       addActor(TextActor(str(row[i])), Location(i, 0))

n = 7
row = []
names = ["Monday", "Tuesday", "Wednesday", "Thursday",
         "Friday", "Saturday", "Sunday"]

makeGameGrid(n, 1, 170, Color.red, False)
setBgColor(Color.white)
show()
for i in range(0 , n):
    dwarf = Dwarf(names[i], i)
    row.append(dwarf)
shuffle(row)
updateGrid()
setTitle("Press any key to get result...")
getKeyCodeWait()
row = sorted(row, cmp = compare)
updateGrid()
setTitle("All done.")
```


MEMO

Die Verwendung der Vergleichsoperatoren für beliebige Datentypen ist zwar nicht zwingend, aber elegan

Man sagt, dass man die Operatoren dabei **überladet** (operator overloading).