Prof. G. Zachmann
David Mainzer

TU Clausthal
Institut für Informatik
22. Mai 2010

Sommersemester 2010

Übungen zu Informatik II - Blatt 7

Abgabe in der Übung am 1. 6. / 2. 6. 2010

Bitte beachten Sie, dass die Programmieraufgaben von Ihnen in der Übung vorgeführt und erklärt werden müssen. Zusätzlich senden Sie die Lösung, unter Angabe ihres Namens, an **dm@tu-clausthal.de**.

Aufgabe 1 (Counting-Sort, *2 Punkte*)

Gehen Sie im folgenden von der Counting-Sort-Implementierung aus der Vorlesung aus

Angenommen der Kopf der letzten ***for***-Schleife in Counting-Sort wird geändert zu

```
for j in range( 0, len(A) ):
```

Arbeitet das modifizierte Verfahren korrekt? Ist es stabil? Begründen Sie.

Aufgabe 2 (LSD-Radix-Sort, *2+2 Punkte*)

a) LSD-Radix-Sort sortiert die Keys, indem alle Keys zunächst bzgl. des *niederwertigsten* Digits, danach bzgl. des zweit-niederwertigsten, etc. sortiert werden. Warum würde LSD-Radix-Sort mit der umgekehrten Reihenfolge, also beim höchstwertigen Digit zu beginnen, nicht funktionieren?

b) Geben Sie die Platz-Komplexität des LSD-Radix-Sort an, d.h., geben Sie eine Funktion $g(n)$ an, so daß für den Platzbedarf $P(n)$ des Algorithmus gilt:

$$P(n) \in \mathcal{O}(g(n))$$

wobei n die Größe des Eingabearray's ist. Begründen Sie diese.

Aufgabe 3 (Mergesort, *12 Punkte*)

Der "klassische" Mergesort ist ein rekursives *top-down*-Verfahren, denn er startet mit dem ganzen Array, zerlegt dieses rekursiv in immer kleinere Teile, und fügt diese dann in mehreren Merge-Schritten wieder zusammen.

Wenn man sich Integer-Arrays mit zufällig gewählten Elementen anschaut, so stellt man fest, dass es schon im unsortierten Array Teilfolgen gibt, die in sich sortiert sind. Diese nennt man auch *Runs*. Im folgenden Beispiel-Array sind diese eingezeichnet:

$$\underbrace{1 \quad 10 \quad 11}\ \underbrace{7 \quad 13 \quad 19 \quad 47}\ \underbrace{17 \quad 99}\ \underbrace{2}$$

Der klassische Mergesort ignoriert diese und erzeugt gewissermaßen per "brute force" solche Runs, um diese dann zu mergen.

Ein ähnlicher Algorithmus nutzt die schon existierenden Runs jedoch folgendermaßen:

```
sei A das Eingabe-Array
erzeuge ein gleich großes Hilfs-Array B
while forever do
    bestimme zunächst die ersten beiden Runs in A
    if es gibt nur einen Run then liefere A zurück
    merge diese beiden Runs und speichere den neuen (längeren Run) in B
    repeat
        bestimme fortlaufend die nächsten beiden Runs und merge diese nach B
    until rechtes Ende von A erreicht (0 oder 1 Run ist übrig)
    vertausche A und B und beginne von vorne im (neuen) Array A
end
```

Das o.g. Array würde also nach dem ersten Vertauschen von A und B (d.h., nach dem ersten Durchlauf der äußeren Schleife) in obigem Algorithmus so aussehen:

$$\underbrace{1 \quad 7 \quad 10 \quad 11 \quad 13 \quad 19 \quad 47}\ \underbrace{2 \quad 17 \quad 99}$$

Dieser Algorithmus heißt *Natural Merge Sort*. Er ist eine iterative *bottom-up*-Variante das klassischen Mergesort.

Implementieren Sie diesen Algorithmus im Framework `sort.py` als Funktion `natural_merge_sort`.

Tips zur Implementierung: es führt höchstwahrscheinlich schneller zum Ziel, wenn Sie schrittweise folgende Funktionen implementieren:

1. Eine Funktion, die von einem Startindex aus das Ende eines Runs sucht.
2. Eine Funktion, die zwei Runs "merget".
3. Die abschließende Funktion `natural_merge_sort`.

Testen Sie jede Funktion gründlich, bevor Sie die nächsthöhere implementieren. Denken Sie auch an Runs, die nur aus einem Element bestehen, oder an den Fall, dass das ganze Array schon zu Beginn nur einen Run hat.

Sie brauchen keine optimierte Implementierung zu schreiben.[1] Wichtig ist, dass Ihr Code keine Bugs enthält und gut strukturiert und kommentiert ist.

Tips zum Debugging für diejenigen, die mit einem ASCII-Editor ihre Programme schreiben, und in der Shell ausführen: An der Stelle, an der man das Programm anhalten möchte, fügt man die Zeile `import pdb; pdb.set_trace()` ein. Dies erzeugt einen sog. *Breakpoint*. Dann startet man das Programm normal vom Terminal aus. Wenn die Ausführung am Breakpoint angekommen ist, hält das Programm an, und man bekommt eine sog. Debugger-Shell. Hier kann man verschiedene Debugging-Kommandos eingeben, um z.B. Variablen zu inspizieren, oder schrittweise durch das Programm zu gehen. Die wichtigsten Kommandos sind `print` und `next`. Weitere Infos finden Sie unter `http://docs.python.org/library/pdb.html`.

Generelle Tips zum Debugging, falls das Programm nicht auf Anhieb funktioniert:

[1] Falls es Ihnen gelingt, eine Implementierung zu schreiben, die schneller als der klassische Mergesort in dieser Implementierung `http://en.literateprograms.org/Merge_sort_(Python)` ist, bekommen Sie 4 Extra-Punkte.

- Wählen Sie kleine Array-Größen, z.B. `N = 10` oder noch kleiner.
- Wenn Sie eine Array-Belegung (eine sog. *Problem-Instanz*) gefunden haben, bei der etwas schief geht, dann ist es oft hilfreich, diese sich ausgeben zu lassen, zu kopieren, und im Programm als Belegung für `a` vorübergehend "fest zu verdrahten".
- Am besten versucht man zunächst, eine minimale Probleminstanz zu finden, bei der der Bug auftritt.
- Man kann besser nachvollziehen, was der Computer macht, wenn man zunächst "kleine" Zahlen verwendet (ersetzen Sie `sys.maxint` z.B. durch 100).
- Wichtig: vergessen Sie nicht, am Schluss alle Modifikationen des Programm, die Sie zum Debugging eingebaut haben, wieder rückgängig zu machen. Manchmal ist es gar nicht so einfach, den Überblick zu behalten. Für mich hat es sich bewährt, jede solche Stelle mit einem kleinen Kommentar zu versehen, nach dem man leicht suchen kann (z.B. `# DEBUG`).

Übrigens: der ganz normale Entwicklungszyklus ist `:-)`

1. Programm(-teil) schreiben;
2. Es compiliert nicht (d.h., es sind noch Syntax-Fehler drin);
3. Es compiliert, aber es crasht sofort;
4. Es crasht nicht mehr, aber der Output ist trotzdem Müll;
5. Debugging (s.o.) ist angesagt ...
6. Ich glaube, ich habe den Bug gefunden!
7. Programm reparieren ...
8. Nun crasht es wieder sofort ...
9. OK, jetzt ist es schon besser, und das Programm scheint sogar den korrekten Output zu produzieren ...
10. Mist, es gibt *eine* Probleminstanz, bei der das Programm das falsche Ergebnis liefert! Also wieder Debugging ...
11. ...
12. Heureka: nun sind alle Bugs beseitigt (... oder?)