DM507 — Algoritmer og datastrukturer

Eksaminatorie-timer uge 9, Forår 2021

Løsninger

Del A

Opgave 1 - Cormen et al. øvelse 6.4-4

Vis at worst-case køretiden af Heapsort er $\Omega(n \lg n)$.

Dette følger direkte af **Sætning 8.1**, da `Heapsort` internt bruger en max-heap til at hive det største element ud af listen, som sorteres. En heap skabes og vedligeholdes udlukkende vha. sammenligninger.

Theorem 8.1: Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

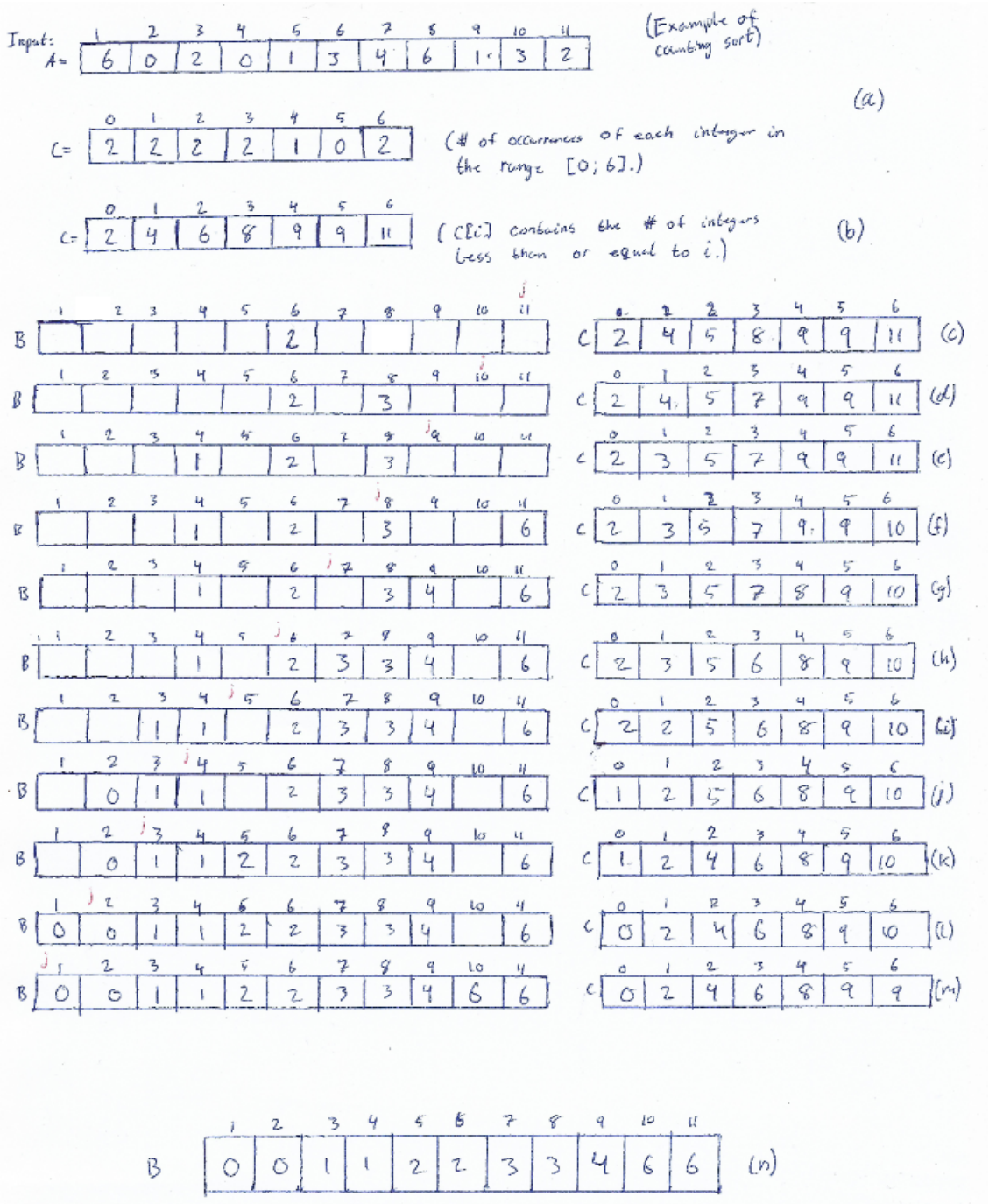
[1, p. 193]

Opgave 2 - Cormen et al. øvelse 8.2-1

Vha. Figur 8.2 (fra Cormen) som en model, illustrer `Couting-Sort` på arrayet

A:	6	0	2	0	1	3	4	6	1	3	2

Se løsningen i figuren under.



Opgave 3 - Cormen et al. øvelse 8.2-3

Hvis vi ændre `for`-løkken på linje 10 i Countingsort til

```
for j = 1 to A.length
```

Vis at algoritmen stadig virker korrekt. Er den modificerede algoritme stabil?

- Korrekthed: Det ændrer ikke på korrekthed. Et givent tal i array A placeres stadig efter værdien i array C, som på index i indeholder det index i array B, hvor et element med værdi

i skal placeres.[1]

- Stabil: Den er ikke stabil. Når tager et element k fra array A, så placeres det som det sidste i gruppen af elementer med værdi k i array B. Hvis man har to elementer j og j' med samme værdi, men hvor j kommer før j', så vil j placeres efter j' i array B.
  - Radix sort: Kræver en stabil sorterings algoritme, så i så fald skal Countingsort ikke længere bruges.

Opgave 4 - Cormen et al. øvelse 8.2-4

Beskriv en algoritme som givet n heltal i intervallet $[0, k]$ udfører en *preprocessing* af input, og derefter besvarer enhver *query* omkring hvor mange af de n heltal som er i intervallet $[a, b]$ i køretid $O(1)$. Preprocessing skal tage $\Theta(n + k)$.

- Preprocessing: Modificér Counting-sort, så den ikke udfører det sidste `for`-loop, hvor elementer skrives sorteret ud i array B. Returnér istedet array C, som indholder indekset for det sidste element i en gruppe af elementer med værdi k.
  - Eksempel: Brug evt. C array fra opgave 2.
- Query: Brug det returnerede array C fra preprocessing, og svar:

$$C[b] - C[a-1] = \#\text{ elementer i interval [a, b]}$$

- Note: I tilfælde af at $a = 0$, da returneres bare $C[b]$, da man ellers ville gå ud af array C.
- Note: Ikke `B[b] - B[a]`, da man så ville returnerer hvor mange elementer som falder ind under intervallet $(a, b]$.

Opgave 5 - Cormen et al. øvelse 8.3-1

Vha. Figur 8.3 som model, illustrer `Radix-Sort` på følgende listen af engelske ord (kun på de første 8 ord): COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB

	↓	↓	↓
COW	SEA	TAB	BOX
DOG	MOB	SEA	COW
SEA	TAB	MOB	DOG
RUG	DOG	DOG	MOB
ROW →	RUG →	COW →	ROW
MOB	COW	ROW	RUG
BOX	ROW	BOX	SEA
TAB	BOX	RUG	TAB

[1] Array A er input-array. Array B er output-array.

Del B

Opgave 1 - Eksamen juni 2008, opgave 1a.

Udfør radix sort med radix 10 på tallene

$$747, 765, 544, 754, 431, 231, 222$$

Vis resultatet efter hver iteration.

	↓	↓	↓
747	431	222	222
765	231	431	231
544	222	231	431
754	544	544	544
431 →	754 →	747 →	747
231	765	754	754
222	747	765	765

Opgave 2 - Cormen et al. øvelse 8.3-2

Hvilken af følgende sorterings algoritmer er stabile: insertion sort, merge sort, heap sort, quick sort? Giv et simplet system som gør enhver sorterings algoritme stabil. Hvor meget ekstra tid og pladsforbrug kræver systemet?

Her antages udgaver som givet af [1] og/eller Rolfs slides (bl.a. [2]).

- Insertion sort: Stabil. I det indre loop byttes to elementer kun hvis der er en inversion. Hvis to elementer er ens, så er der ikke en inversion – derfor vil elementer have samme relative orden i både input og ouput array.
- Merge sort: Stabil. Man tager altid fra "venstre"-listen før højre liste i merge-proceduren. Derfor vil man aldrig bytte rundt på to ens elementer – altså bevares den relativt indbyrdes orden.
- Heap sort: Ikke stabil. Modeksempel: Prøv at sortere

2	$2'$	$2''$

  med heapsort. Dette vil give output

$2'$	$2''$	2

- Quick sort: Ikke stabil. I partition flyttes elementer rundt, når et element er $>$ pivot x, og dette kan gøre quicksort ikke-stabil. Prøv at bruge partion på

4	42	$42'$	8

  Efter partition vil output være

4	8	$42'$	42

Delproblemerne skal sorteres rekursivt, men der vil ikke ske flere ændringer ift. ovenstående. Hvis elementers nøgler udvides, så den består af en tupel (original *key*, indeks), vil et array

4	42	42	8

blive

$(4,1)$	$(42,2)$	$(42,3)$	$(8,4)$

Altså er $(42,2) \neq (42,3)$ selvom de har samme originale key (altså $42 = 42$).

- Fremgangsmåde: Først sorterer man på elementers originale *key*, og hvis de er ens, så bruger man deres orignale indeks.
- Køretid: Kræver $\Theta(n)$ mere tid, da man skal udvide hvert af de n elementers *key*. Under algoritmen kræver det stadig konstant arbejde at sammenligne to elementer.
- Pladsforbrug: Kræver $\Theta(n)$ mere plads, da hvert af de n elementer også skal gemme indeks (ikke bare deres originale *key*).

Opgave 3 - Cormen et al. øvelse 8.3-4

Vis hvordan man kan sortere n heltal i intervallet $[0, n^3 - 1]$ i $O(n)$ køretid.

Hvad er "problemet" – kan vi ikke bare bruge Counting sort?

- Nej. Der er n elementer, som kan have en værdi i intervallet $[0; n^3 - 1]$. Altså, så vil der være $k = (n^3 - 1) + 1 = n^3$ forskellige værdier.
- Køretid: Hermed er køretiden $\underbrace{O(n+k)}_{\text{Counting sort køretid}} = O(n^3)$.
- Radix sort: Samme problem gælder her med mindre...(a good idea will follow!)

Den gode idé: Se hvert tal x i radix n (base n).

- Antal cifre: x i radix n kræver 3 cifre at repræsentere. Hvorfor?[2]

  Det største mulige tal er $n^3 - 1$, som kan konverteres til radix n på følgende måde:

Heltalsdivision	Kvotient	Rest	Ciffer
$\frac{n^3-1}{n}$	$n^2 - 1$	$n - 1$	d_0
$\frac{n^2-1}{n}$	$n^1 - 1$	$n - 1$	d_1
$\frac{n^1-1}{n}$	0	$n - 1$	d_2

  Altså, så kan man repræsentere x i base n vha. de tre cifre $d_2 d_1 d_0$.

- Observation: Da d_2, d_1 og d_0 er base n cifre, så kan de antage de n forskellige værdier i intervallet $[0, n - 1]$.

[2] Anden måde: Der er n^3 mulige tal og $\log_n(n^3) = 3$, så der skal bruges 3 cifre.

- Radix sort: Benyttes Radix sort (med Counting sort) på de n tal i radix n, så bliver køretiden $O(n)$. Hvorfor? Hvert tal x kan repræsenteres med højst 3 cifre i radix n, så Radix sort fortager $d = 3$ gennemløb. I hvert gennemløb benyttes Counting sort på de i'te cifre fra højre. Dette tager $O(n+k)$, da der er n tal og $k = n$, da hvert ciffer er et heltal i intervallet $[0, n-1]$. I alt bliver køretiden $O(d \cdot (n+k)) = O(3 \cdot (n+n)) = O(6n) = O(n)$.

Vigtig pointe: At sammenligne lexiografisk og ordensmæssigt er det samme. Sammenlignes 141 og 123, så er sammenligningen $123 < 141$ det samme som er sammenligne lexiografisk:

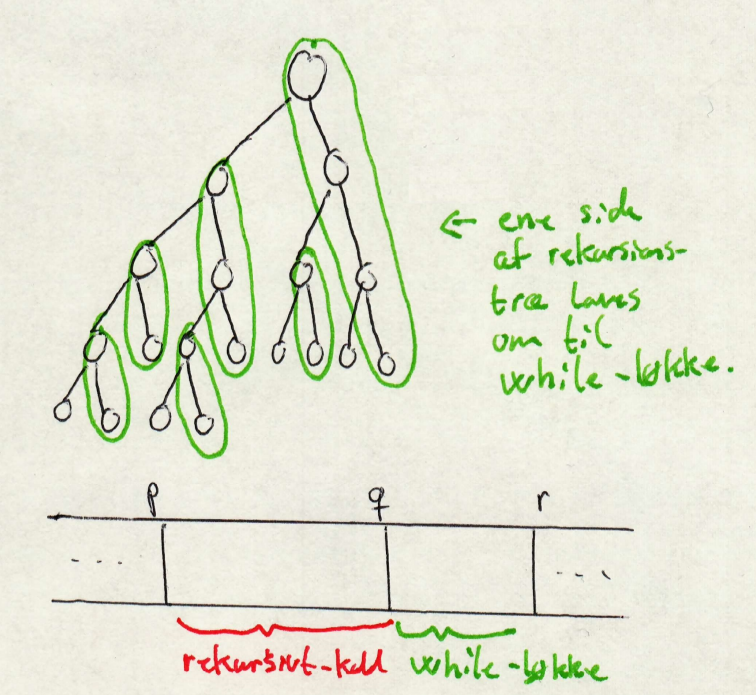
$$\begin{array}{ccc} 1 & 2 & 3 \\ = & < & \\ 1 & 4 & 1 \end{array}$$

Note: Konvertere x til base n? Dette kan man gøre før, og det tager $O(k \cdot n)$ tid, hvor n er antal tal og k er antallet af cifre i basen, der konverteres til. I dette tilfælde er $k = 3$, så køretiden er $O(n)$

Opgave 4 - Cormen et al. problem 7-4

a) Argumenter for at `TAIL-RECURSIVE-QUICKSORT(A, 1, A.length)` sorterer array A korrekt.

- Den eneste forskel er bare, at der ikke foretages to rekursive kald efter en partition, men at der foretages et rekursivt kald for venstre delproblem og man bruger et while loop for højre delproblem.
- Fordel: Dermed bruges "den samme" funktion til at sortere højredelen af partitionen fremfor at lave et rekursivt kald (skabe et nyt stack frame).
- Illutration: Rekursionstræets højresider laves i princippet om til while-løkker.



b) Beskriv et senarie hvor `TAIL-RECURSIVE-QUICKSORT` stack-dybde er $\Theta(n)$ på et input array af størrelse n.

- Eksempel: Brug et array sorteret i ikke-faldende orden.
- Problem: Det dårlige split laver to delproblemer af størrelse $n-1$ og 0. Det er delproblemet af størrelse $n-1$, der bruges i det rekursive kald. Dvs. højre delproblem er altid tom.

c) Modificer koden for `TAIL-RECURSIVE-QUICKSORT`, så worst-case stack-dybden er $\Theta(\lg n)$. Oprethold $O(n \lg n)$ forventet køretid af algoritmen.

- Kode: Se nedenunder
- Stack-dybde: $\Theta(\lg n)$. Hvert rekursivt kald foretages på et subarray af størrelse højst $\frac{n}{2}, \frac{n}{4}, \ldots, 1$. Altså, antallet af rekursive kald, der foretages, er $\Theta(\lg n)$.
- Køretid: Samme forventede køretid, da vi arbejder på samme delproblemer som ordinær quicksort, men vi vælger bare klogt, så vi altid foretager et rekursivt kald på det mindste delproblem.

```
Tail-Recursive-Quicksort'(A, p, r):
  while p < r:
    q = Partition(A, p, r)
    // Checks left-side > right-side
    if (q-p) < (r-q)
      Tail-Recursive-Quicksort'(A, p, q - 1)
      p = q + 1
    else
      Tail-Recursive-Quicksort'(A, q+1, r)
      r = q - 1
```

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

[2] Rolf Fagerberg. Sortering. URL `https://imada.sdu.dk/~rolf/Edu/DM507/F20/sortingSlides.pdf`, 2020.