Informasjon

Du måtte klare 9 av 17 oppgaver for å få øvingen godkjent.

Finner du feil eller mangler eller har forslag til forbedringer, ta gjerne kontakt!

Oppgave 2

«*like elementer i den sorterte listen opptrer i samme rekkefølge som de opptråde i den usorterte listen*» er riktig.

Kommentar:
Definisjonen av en stabil sorteringsalgoritme kan blant annet finnes på side 196 i boken.

Oppgave 3

$\langle(1,5),(3,4),(3,2),(3,3),(5,1),(8,3)\rangle$ er riktig.

Kommentar:
Siden vi kun sorterer på x-verdiene, må man passe på at man ikke også sorterer på y-verdiene. Altså, siden $(3,4)$ opptrer før $(3,2)$ i den originale listen og begge har samme x-verdi, så må de opptre i den samme rekkefølgen i den sorterte listen.

Oppgave 4

Insertion-Sort, Merge-Sort, «Radix-Sort *med* Insertion-Sort *som subrutine*» og Counting-Sort er riktig.

Kommentar:
Partition i Quicksort gjør flere bytter av elementer. Disse byttene gjøres ikke nødvendigvis slik at den originale rekkefølgen av like elementer opprettholdes. Dette gjør at Quicksort ikke er stabil. For eksempel prøv å sortere $A = \langle 2,2,1\rangle$, hvor du holder styr på hvilken som er hvilken av toerene.
Radix-Sort trenger en stabil sorteringsalgoritme som subrutine for å selv være stabil. Siden Quicksort ikke er en stabil sorteringsalgoritme, så kan heller ikke Radix-Sort med Quicksort som subrutine være en stabil sorteringsalgoritme.

Oppgave 5

$\Omega(n \lg n)$ er riktig

Kommentar:
Her vet vi at S er en vilkårlig sammenligningsbasert sorteringsalgoritme. Siden vi ikke vet noe mer om S, er det vanskelig å si noe veldig spesifikt her. Men, vi vet at alle sammenligningsbaserte sorteringsalgoritmer ikke kan ha en bedre verste tilfelle kjøretid enn $\Theta(n \lg n)$. De kan ha en som er verre, men ikke bedre. Derfor kan vi si at denne er $\Omega(n \lg n)$, men ikke noe mer nøyaktig enn dette.

Oppgave 6

«*Kjøretiden til* S *i beste tilfelle kan være* $\Theta(n)$», «*Kjøretiden til* S *i verste tilfelle kan være* $\Theta(n^3)$» og «S *kan være* QUICKSORT» er riktig.

Kommentar:
Ikke alle sammenligningsbaserte sorteringsalgoritmer er stabile (f.eks. QUICKSORT er ikke). Verste tilfelle kjøretiden til en sammenligningsbasert sorteringsalgoritme må være $\Omega(n \lg n)$ og kan derfor ikke være $\Theta(n)$.

Oppgave 7

$\Theta(k)$ er riktig.

Kommentar:
Kjøretiden til COUNTING-SORT er $\Theta(n+k)$. Siden $k \gg n$, domineres n av k og derfor er $\Theta(k)$ også gyldig. Dette er samtidig mer nøyaktig enn $\Theta(n+k)$ i denne sammenhengen, da vi fjerner unødvendige ledd.

Oppgave 8

$O(n^2)$ er riktig.

Kommentar:
Her vil det i snitt havne $n/512$ elementer i hver bøtte. Når n blir stor, så har denne konstanten ikke noe å si, så det vi være $\Theta(n)$ elementer i hver bøtte. Siden INSERTION-SORT brukes til å sortere hver bøtte og har $\Theta(n^2)$ gjennomsnittlig kjøretid, vil også den totale gjennomsnittlige kjøretiden være i samme størrelsesorden.

Her er det greit å merke seg at uansett hva dette konstante antallet bøtter er, så vil det ikke være noen faktisk forskjell i den asymptotiske kjøretiden. Når n blir stor nok, så har ikke denne konstanten noe å si. Asymptotisk kjøretid bryr seg kun om kjøretiden ved slike store verdier av n.

Oppgave 9

«*Dette er nødvendig for at algoritmen skal være en stabil sorteringsalgoritme.*» er riktig.

Kommentar:
Counting-Sort fungerer helt fint hvis man itererer motsatt vei og man ville da utført akkurat like mange operasjoner og dermed fått samme kjøretid. Det er altså bare en nødvendighet for at algoritmen skal være stabil.

Oppgave 10

```
# Implementasjon av pseudokoden til Counting-Sort i CLRS.
def counting_sort(A, B):
    # Vi vet at k er mindre eller lik 2047. Kan derfor ha 2048 verdier.
    values = [0] * 2048
    # Teller hvor mange det er av de forskjellige verdiene.
    for element in A:
        values[element] += 1
    for i in range(1, 2048):
        values[i] += values[i - 1]
    # Itererer baklengs over A for å ha en stabil sorteringsalgoritme.
    for i in range(len(A) - 1, -1, -1):
        element = A[i]
        values[element] -= 1
        B[values[element]] = element
```

Oppgave 11

«*Det er mulig å oppnå en kjøretid på* $O(n)$ *i verste tilfelle.*» er riktig.

Kommentar:
For medianen i en liste med et odde antall elementer trenger vi å finne elementet som ville ligget i midten av listen hvis den er sortert. For et partall antall elementer må vi finne de to elementene som ville ligget i midten av listen hvis den er sortert. Vi kjenner alltid lengden på listen og kan derfor regne ut indeksen til de midterste elementene. Select kan dermed brukes til å hente ut disse verdiene i lineær tidskompleksitet i verste tilfelle. Siden vi har et konstant antall elementer vi skal hente ut, får vi en kjøretid på $O(cn)$ i verste tilfelle, hvor $c \in 1, 2$. Dermed har vi en lineær tidskompleksitet i verste tilfelle.

Oppgave 12

Counting-Sort er riktig.

Kommentar:
Siden vi har et stort antall verdier i intervallet [1, 6] er det veldig effektivt å bruke Counting-Sort. Her kunne man tenkt at Radix-Sort med Counting-Sort kunne vært det mest effektive. Men, da d er så liten og vi bruker Counting-Sort internt, skal det mye til at dette er mer effektivt enn å bare bruke Counting-Sort.
Bucket-Sort kunne også vært et godt alternativ her, men her ender vi fort opp med å måtte utføre masse ekstra arbeid, som Counting-Sort slipper. Dette da vi må lage n forskjellige bøtter og iterere over disse en del ganger.

Oppgave 13

Insertion-Sort er riktig.

Kommentar:
Her vet vi ingenting om elementene i listen, og det er derfor vanskelig/umulig å kunne bruke ikke-sammenligningsbaserte sorteringsalgoritmer. Siden kun to elementer har byttet plass i listen, slipper man å flytte elementene veldig langt i Insertion-Sort. Man kan bevise at kjøretiden til Insertion-Sort i verste tilfelle for denne typen situasjon vil være $O(n)$. Dette er bedre enn både den gjennomsnittlige kjøretiden til Merge-Sort og kjøretiden til Merge-Sort i det beste tilfelle.

Oppgave 14

«Radix-Sort *med* Counting-Sort» er riktig.

Kommentar:
Her har vi ingen idé om distribusjonen til brukernavnene, men de er nok ikke uniformt distribuert. Dermed vil nok ikke Bucket-Sort gjøre det så bra. Vi kan dessuten regne ut at Radix-Sort bør gjøre det bedre enn Merge-Sort og Insertion-Sort med dette antallet elementer (ved å se på antall operasjoner). Her vil det også være i hvert fall $26^8 \approx 10^{11}$ forskjellige mulige verdier for brukernavnene, så Counting-Sort vil være veldig tregt.

Oppgave 15

«*Når $n \ll k$ (n mye mindre enn k)*» er riktig.

Kommentar:
Counting-Sort har kjøretiden $O(n + k)$, mens denne varianten av Radix-Sort har kjøretiden $O(d(n + k_0))$, hvor d er antall siffer det er i det største heltallet, k, og k_0 er antall mulige verdier et siffer kan ta (f.eks. 10 i 10-tallssytemet). Når $k < n$ dominerer n kjøretiden til både Counting-Sort og Radix-Sort, og kjøretidene blir da $O(n)$ og $O(dn)$. Når k er litt større enn n vil man være i et område hvor de to algoritmene er omtrentlig like raske. Når $k > dn$ får man kjøretidene $O(k)$ og $O(dn + dk_0)$. dk_0 er av definisjon alltid mindre eller lik k, så dermed må

RADIX-SORT være best i dette tilfellet. Dette skjer når $k \gg n$.

Denne oppgaven er inspirert av oppgave 3 i eksamen fra august 1995.

Oppgave 16

```python
def flexradix(A, d):
    # Deler opp elementene basert på lengde
    length_buckets = [[] for _ in range(d + 1)]
    for string in A:
        length_buckets[len(string)].append(string)

    # Starter med de lengste strengene
    current_min_length = d
    strings = []
    while current_min_length:
        if length_buckets[current_min_length]:
            # Legg til de kortere strengene først
            strings = length_buckets[current_min_length] + strings
        # Bare utfør sort hvis vi har mer enn en streng
        if len(strings) > 1:
            strings = sort(strings, current_min_length - 1)
        current_min_length -= 1
    return strings

# Implementasjon av en miks mellom Bucket-Sort og Counting-Sort
def sort(A, i):
    buckets = [[] for _ in range(26)]
    for string in A:
        buckets[char_to_int(string[i])].append(string)

    # Python triks for å pakke ut lister av lister
    return [string for bucket in buckets for string in bucket]
```

Kommentar:
Her er det viktig at man ikke utfører noen padding eller lignende på tekststrengene. Da ender man opp med en tidskompleksitet på $\Theta(nd)$ og ikke $\Theta(n+d)$. Det man må gjøre, er å dele opp strengene etter lengde. Da kan man begynne med å sortere de lengste basert på de siste tegnene, for så å legge til de kortere strengene etterhvert som man beveger seg lengre frem i de lange strengene. Man må legge til de kortere strengene på starten av listen, da man ellers ikke vil kunne få sortert i leksikalsk rekkefølge.

I koden over er det anvendt en sorteringsalgoritme som ikke er gjennomgått i pensum. Denne utnytter egentlig bare det at strengene allerede er sortert basert på leksikalsk rekkefølge i alle de senere tegnene og dermed kan vi bare fordele strengene i 26 bøtter for så å sette disse sammen. Dette ligner på BUCKET-SORT, men da uten sorteringen innad i bøttene.

Oppgave 17

«SELECT *bruker en strategi for å velge et element som er nær medianen som pivot, mens* RANDOMIZED-SELECT *velger et tilfeldig element.*» er riktig.

Kommentar:
SELECT anvender en strategi som kalles «median of medians», hvor man deler opp listen i mange små lister og velger median av median til disse listene som pivot. Da treffer man som regel i nærheten av den faktiske medianen og får delt delen av listen man ser på i to. Merk at dette tar en del mer tid enn det å bare velge en tilfeldig pivot, men garanterer en bedre kjøretid i verste tilfellet.

Oppgave 18

```
from random import randint

# Implementasjon av Partition fra CLRS
def partition(A, p, r):
    x = A[r]
    i = p - 1
    for j in range(p, r):
        if A[j] <= x:
            i = i + 1
            A[i], A[j] = A[j], A[i]
    A[i + 1], A[r] = A[r], A[i + 1]
    return i + 1

# Implementasjon av Randomized-Partition fra CLRS
def randomized_partition(A, p, r):
    i = randint(p, r)
    A[r], A[i] = A[i], A[r]
    return partition(A, p, r)

# Variant av Randomized-Select fra CLRS
def randomized_k_largest(A, p, r, i):
    if p == r:
        return A[p:]
    q = randomized_partition(A, p, r)
    if i == q:
        return A[q:]
    elif i < q:
        return randomized_k_largest(A, p, q - 1, i)
    return randomized_k_largest(A, q + 1, r, i)

def k_largest(A, k):
    if k == 0:
        return []
    return randomized_k_largest(A, 0, len(A) - 1, len(A) - k)
```

Kommentar:
Her implementeres en variant av RANDOMIZED-SELECT hvor i stedet for å returnere det i-te minste elementet, så returneres heller dette og alle elementene som er større enn dette. Det fungerer siden RANDOMIZED-PARTITION garanterer at alle elementene på høyresiden av pivoten er større enn pivoten. Dette holder for oss, da vi ikke trenger å sortere de k største verdiene.

Oppgaver som handler om å finne de k største eller minste tallene i en liste forekommer stadig vekk på eksamen. Senest under konten sommeren 2020 ble det gitt en slik oppgave.

Oppgave 19

Trikset her er å se at hvis man ser på tallene med base n får man maksimalt 3 siffere per tall. Med dette kan man lage en algoritme med følgende steg:

1. Gjør om tallene til base n.
2. Sorter tallene ved hjelp av RADIX-SORT med COUNTING-SORT som subrutine.
3. Gjør om tallene tilbake til den originale basen.

Steg 1. og 3. tar begge $O(n)$ tid. Steg 2. har en tidskompleksitet på $O(3n) = O(n)$, da denne varianten av RADIX-SORT har tidskompleksitet på $O(d(n+k_0))$, hvor $d = 3$ og $k_0 = n$ i dette tilfellet.

Denne oppgaven er inspirert av oppgave 2 fra eksamen i januar 1994.

Oppgave 20

For at to av disse operasjonene skal ha sublineær tidskompleksitet, så trenger vi kun å opprettholde en sortert liste. Da kan vi sette inn et nytt element i $\Theta(n)$ tid, ved å iterere over listen for å finne ut hvor elementet skal legges inn. Så kan vi forskyve elementene til høyre for denne indeksen et hakk mot høyre og plassere inn det nye elementet. Da vil uthenting av den største verdien ha tidskompleksitet $O(1)$, fordi vi kan finne det siste elementet i listen i konstant tid. Sletting av det største elementet vil også kunne utføres i konstant tid, da vi ikke trenger å gjøre noe mer enn å korte ned lengden til listen med en.

Det er også mulig å få til at alle tre operasjonene har sublineær tidskompleksitet. En *max-heap* er en datastruktur som gjør akkurat dette. Denne oppnår en tidskompleksitet på $O(\lg n)$ for både innsettelse av et element og sletting av det største elementet, samt $O(1)$ tidskompleksitet for å hente ut det største elementet. Kapittel 6 i CLRS og forelesning 5 går igjennom denne datastrukturen i detalj.

Oppgave 21

Dette er generelt ikke mulig. For å se hvorfor kan vi anta at dette er mulig. Da kunne vi med hjelp av denne datastrukturen sortert en vilkårlig liste med elementer i $O(n)$ tid. Dette kan gjøres ved å sette inn alle n elementene fra listen i datastrukturen. Dette har en tidskompleksitet på $O(n)$. Vi kan så hente ut elementene fra størst til minst og sette dem inn på riktig plass i listen

med en tidskompleksitet på O(1) per element, eller da O(n) for alle elementene. Dette gir en tidskompleksitet på O(n) for sortering, noe som vi vet ikke er mulig for en liste med vilkårlige elementer uten noen restriksjon på verdiene til elementene.

Oppgave 23

For eksamensoppgavene, se løsningsforslaget til den gitte eksamenen.

For oppgaver i CLRS, så finnes det mange ressurser på nettet som har fullstendige eller nesten fullstendige løsningsforslag på alle oppgavene i boken. Det er ikke garantert at disse er 100% korrekte, men de kan gi en god indikasjon på om du har fått til oppgaven. Det er selvfølgelig også mulig å spørre studassene om hjelp med disse oppgavene.

Et eksempel på et ganske greit løsningsforslag på CLRS, laget av en tidligere doktorgradsstudent ved Rutgers, finnes her.