Informasjon

Du måtte klare 10 av 22 oppgaver for å få øvingen godkjent.

Finner du feil, mangler eller forbedringer, ta gjerne kontakt!

Oppgave 2

PARTITION er riktig.

Kommentar:
Se side 171 i CLRS (læreboka).

Oppgave 3

«QUICKSORT *gjør sorteringsarbeidet før det rekursive kallet, mens* MERGE-SORT *gjør det etterpå.*» er riktig.

Kommentar:
QUICKSORT benytter seg av PARTITION som flytter tallene på rett side av pivot-elementet før den utfører rekursive kall. MERGE-SORT utfører først rekursive kall på hver halvdel av listen man ønsker å sortere, før den flytter tallene i sortert rekkefølge under flettesteget.

Oppgave 4

«*Etter vi har kjørt* PARTITION *er vi garantert at* $A[1] \leqslant A[A.length]$» og «A *vil bli uforandret dersom* $A[A.length]$ *er ekte større enn alle de andre elementene i A før vi kjører* PARTITION» er riktige.

Kommentar:
Etter at vi har kjørt PARTITION er vi garantert at $A[1] \leqslant A[A.length]$, siden listen da vil være delt inn i potensielt tre deler:

1. Alle elementer utenom splittelementet (pivot) som er mindre eller lik splittelementet.
2. Splittelementet
3. Alle elementer som er større enn splittelementet.

Del (1) og (3) er potensielt tomme. Vi vet at elementene i (1) og (2) er mindre eller lik splittelementet og at elementene i (2) og (3) er større eller lik splittelementet. I tillegg vet vi at $A[1]$ er i enten (1) eller (2), samt at $A[A.length]$ er i enten (2) eller (3). Hvis vi kaller splittelementet x får vi dermed:

$$A[1] \leqslant x \leqslant A[A.length] \implies A[1] \leqslant A[A.length]$$

A vil bli uforandret dersom A[A.*length*] er ekte større enn alle de andre elementene i A før vi kjører A[A.*length*]. Legg merke til at elementet som er A[A.*length*] før funksjonen kjøres ender opp med å bli splittelementet. Forløkken vil ikke flytte på noen elementer, siden $i = j$ hver gang algoritmen skal bytte på to elementer. Når splittelementet skal flyttes til slutt skal det flyttes til indeks $i + 1 = (p - 1) + (r - p) + 1 = r = \text{A}.length$, så elementet beholder plassen sin.

A er ikke garantert sortert etter funksjonskallet siden PARTITION ikke sorterer listen, men deler listen i to basert på splittelementet.

Funksjonskallet tar lineær tid uansett hvordan A ser ut før vi kjører funksjonen. Dette er fordi kjøretiden til PARTITION dominieres av forløkken som går gjennom elementene (bortsett fra spittelementet) en gang og dermed tar lineær tid.

Oppgave 5

$\langle 1, 8, 1, 2, 3, 4, 7, 6, 8 \rangle$ er riktig.

Kommentar:
MERGE(A, p, q, r) fletter sammen den delen av A fra og med p til og med q med den delen av A fra og med $q + 1$ til og med r. Det vil si vi skal flette sammen $\langle 3, 4 \rangle$ med $\langle 1, 2, 7 \rangle$. Dette fører til at vi får $\langle 1, 2, 3, 4, 7 \rangle$.

Oppgave 6

«*Begge har kjøretid* $\Theta(n^2)$» er riktig.

Kommentar:
I begge tilfeller får vi rekurrensen $\text{T}(n) = \text{T}(n - 1) + \Theta(n)$ som har løsning $\text{T}(n) = \Theta(n^2)$. I A sitt tilfelle vil PARTITION ikke flytte noen elementer og vi ender opp med å kjøre QUICKSORT på de $n - 1$ første elementene. I B sitt tilfelle vil PARTITION flytte det bakerste tallet først i listen, og vi vil da kjøre QUICKSORT på de $n - 1$ siste elementene.

Oppgave 7

«QUICKSORT *er en inplace sorteringsalgoritme*» er riktig

Kommentar:
QUICKSORT bytter kun plass på elementene i listen, og lagrer dermed ikke mer enn et konstant antall elementer utenfor listen. (I motsetning til f. eks. MERGE-SORT, som lager nye lister der man setter inn elementene og lagrer dermed et lineært antall elementer utenfor listen)

MERGE-SORT er som sagt ikke in-place.

QUICKSORT har en kjøretid på $\Theta(n^2)$ i verste tilfelle, som forekommer f. eks. om listen er sortert.

MERGE-SORT kan ikke få $\Theta(n^2)$ kjøretid, siden den er $\Theta(n \lg n)$ uansett input.

Oppgave 8

«*En sortert liste vil ikke alltid ha kjøretid* $\Theta(n^2)$» er riktig.

Kommentar:
En sortert liste vil ikke alltid ha kjøretid $\Theta(n^2)$ siden splittelementet velges tilfeldig. Dermed vil man (sannsynligvis) ofte unngå å velge et av de største eller et av de minste elementene å partisjonere på, og dermed unngå kjøretid på $\Theta(n^2)$, som er kjøretiden i verste tilfelle, og heller ende på $\Theta(n \lg n)$.

Splittelementet bytter plass med det bakerste (og ikke det første) elementet i listen.

RANDOMIZED-QUICKSORT er $\Theta(n^2)$ i verste fall. En enkel måte å se det på er at man oppnår dette om man har en sortert liste og algoritmen tilfeldigvis velger bakerste element som splittelement hver gang (samme splittelement som vanlig QUICKSORT).

Det miderste element ender ikke nødvendigvis opp som splittelementet, siden algoritmen velger et tilfeldig element som splittelement.

Oppgave 9

```
def merge(a, p, q, r):
    n_1 = q - p + 1
    n_2 = r - q
    l_arr = (n_1 + 1) * [None]
    r_arr = (n_2 + 1) * [None]
    for i in range(n_1):
        l_arr[i] = a[p + i]
    for j in range(n_2):
        r_arr[j] = a[q + j + 1]
    l_arr[n_1] = float("inf")
    r_arr[n_2] = float("inf")
    i = 0
    j = 0
    for k in range(p, r + 1):
        if l_arr[i] <= r_arr[j]:
            a[k] = l_arr[i]
            i = i + 1
        else:
            a[k] = r_arr[j]
            j = j + 1

def merge_sort(a, p, r):
    if p < r:
        q = (p + r)//2
        merge_sort(a, p, q)
```

```
        merge_sort(a, q + 1, r)
        merge(a, p, q, r)
```

Kommentar:
Nesten rett avskrift fra CLRS (læreboka) s. 31 og 34. Må passe på at listene her er 0-indeksert og ikke 1-indeksert som i CLRS.

Oppgave 10

FIND-MAXIMUM-SUBARRAY er riktig.

Kommentar:
En mulig løsning hadde vært å slå opp algoritmene som var alternativ og sett hvilken som passet.

Oppgave 11

A_2 og A_3 er riktig.

Kommentar:
Her er det viktig å huske hva induksjon er, og hvordan det fungerer. Hypotesen vår var at

$$T(n) \geqslant c \cdot n^3$$

for en $n < n'$. For at induksjonssteget skal være gyldig må vi vise at dersom vi antar dette er også

$$T(n') \geqslant c \cdot (n')^3$$

For $T_1(n)$ har vi at

$$T_1(n) \geqslant c \cdot n^3 - 5n^2 \not\Longrightarrow T_1(n) \geqslant c \cdot n^3$$

siden

$$c \cdot n^3 - 5n^2 < c \cdot n^3$$

for $n \in Z^+$. Dermed har vi ikke bevist dette for $T_1(n)$
For $T_2(n)$ har vi at

$$T_2(n) \geqslant c \cdot n^4 + 5n^2 \implies T_2(n) \geqslant c \cdot n^3$$

siden

$$c \cdot n^4 + 5n^2 \geqslant c \cdot n^3$$

for $n \in Z^+$. Dermed har vi bevist dette for $T_2(n)$
For $T_3(n)$ har vi at

$$T_3(n) \geqslant c \cdot n^3 + n \implies T_3(n) \geqslant c \cdot n^3$$

siden

$$c \cdot n^3 + n \geqslant c \cdot n^3$$

for $n \in Z^+$. Dermed har vi bevist dette for $T_3(n)$
For $T_4(n)$ har vi at

$$T_4(n) \geqslant c \cdot n^3 - 1 \not\Longrightarrow T_4(n) \geqslant c \cdot n^3$$

siden

$$c \cdot n^3 - 1 < c \cdot n^3$$

for $n \in \mathrm{Z}^+$, Dermed har vi ikke bevist dette for $\mathrm{T}_4(n)$

Oppgave 12

```
def find_maximum(arr):
    return _find_maximum(arr, 0, len(arr) - 1)

# Finner maksimum i området [low, high]
def _find_maximum(arr, low, high):
    # Grunntilfelle
    if high - low < 2:
        return max(arr[low], arr[high])

    mid = (low + high) // 2

    # Low ligger på toppen av listen
    if arr[low] > max(arr[low - 1], arr[low + 1]):
        return arr[low]

    elif arr[low - 1] < arr[low] < arr[low + 1]:
        if arr[mid] < arr[mid + 1] and arr[mid] > arr[low]:
            # Se (1) under
            return _find_maximum(arr, mid, high)
        # Se (2) under
        return _find_maximum(arr, low, mid)
    elif arr[mid] < arr[low] or arr[mid] < arr[mid + 1]:
        # Se (3) under
        return _find_maximum(arr, mid, high)
    # Se (4) under
    return _find_maximum(arr, low, mid)
```

Kommentar:
For de fire tilfellene i hintet så har man følgende:

1. Listen stiger ved *low* og når du kommer til *mid* stiger listen fortsatt, og siden $arr[mid] > arr[low]$ så kan ikke listen ha synket mellom *low* og *mid*, og dermed må maksimum være mellom *mid* og *high*.
2. Listen stiger ved *low* og når du kommer til *mid* har enten listen begynt å synke igjen, eller så har den sunket og steget igjen. Uansett må maksimum ligge mellom *low* og *mid*.
3. Listen synker ved *low* (kan ikke stige siden da ville vi nådd (1) eller (2)). Enten har ikke listen kommet seg over det nivået den var ved *low* når den kommer seg til *mid*, noe som betyr at maksimum må ligge mellom *mid* og *high*, eller så stiger listen ved *mid* og da må også maksimum ligge mellom *mid* og *high*, siden listen kan ikke endre om den går opp eller ned tre ganger mellom *low* og *mid*.
4. Nå må listen være synkende ved *low* (evt. *low* er bunnen). Siden forrige if-setning ikke ble `True`, så må $arr[mid] > arr[low]$ og $arr[mid] > arr[mid + 1]$. Altså listen synker ved *mid*,

men den er høyere enn *low*, som betyr at listen må ha økt mellom *low* og *mid* for å så begynne å synke igjen, så maksimum ligger mellom *low* og *mid*.

Oppgave 13

«*Case 2*» er riktig.

Kommentar:

$$\mathrm{T}(n) = 64 \cdot \mathrm{T}(n/4) + 3n^3 + 7n \implies \mathrm{T}(n) = 64 \cdot \mathrm{T}(n/4) + \Theta(n^3)$$

Masterteoremet går ut på å ha en rekurrens på formen

$$\mathrm{T}(n) = a \cdot \mathrm{T}(n/b) + \Theta(n^c)$$

og sammenligne $\log_b(a)$ med c. Her er $\log_b(a) = \log_4(64) = 3 = c$ så dette er case 2.

Oppgave 14

$\Theta(n^3 \lg n)$ er riktig.

Kommentar:
Følger av at det er case 2 i masterteoremet (se oppgaven over).

Oppgave 15

$\Theta(n^{\log_3(2)})$ er riktig.

Kommentar:
Vi starter med å sette opp rekurrensene $\mathrm{T}_1(n)$ for FUNCTION-A(n) og $\mathrm{T}_2(n)$ for FUNCTION-B(n):

$$\mathrm{T}_1(n) = \mathrm{T}_1(n/3) + \mathrm{T}_2(n-2) + \Theta(\sqrt{n})$$

$$\mathrm{T}_2(n) = \mathrm{T}_1(n/3) + \Theta(\sqrt{n})$$

Hvis vi gjør som hintet sier og setter inn for $\mathrm{T}_2(n)$ i rekurrensen for $\mathrm{T}_1(n)$ får vi:

$$\mathrm{T}_1(n) = \mathrm{T}_1(n/3) + [\mathrm{T}_1((n-2)/3) + \Theta(\sqrt{n})] + \Theta(\sqrt{n})$$

$$\mathrm{T}_1(n) = \mathrm{T}_1(n/3) + \mathrm{T}_1((n-2)/3) + \Theta(\sqrt{n})$$

Siden

$$\lim_{n\to\infty} \frac{(n-2)/3}{n} = \frac{1}{3} < 1$$

vil argumentet til $\mathrm{T}_1((n-2)/3)$-leddet synke tilnærmet geometrisk og ikke aritmetrisk for store n (dette er tilsvarende fenomenet i matematisk analyse, der eksponensiell vekst vinner over lineær

vekst når x er stor), og vi kan dermed bytte ut $\mathrm{T}_1((n-2)/3)$ med $\mathrm{T}_1(n/3)$ og få

$$\mathrm{T}_1(n) = 2 \cdot \mathrm{T}_1(n/3) + \Theta(\sqrt{n})$$

som etter masterteoremet er $\Theta(n^{\log_3(2)})$

Merk: *I teorien vil forenklinger på formen* $\mathrm{T}_1((n-2)/3)$ *til* $\mathrm{T}_1(n/3)$ *kunne føre til at svaret ikke blir helt riktig. I praksis skal det en del til for at dette skjer, men for å være helt sikker kan man vise ved induksjon at svaret man har kommet frem til faktisk stemmer.*

Oppgave 16

$\mathrm{T}(n) = \Theta(n)$ er riktig.

Kommentar:
Her får vi rekurrensen

$$\mathrm{T}(n) = \mathrm{T}(n-42) + \Theta(1)$$
$$\mathrm{T}(n' \leqslant 42) = \Theta(1)$$

Siden $f(n) = \Theta(1)$ (arbeidet per iterasjon) trenger vi kun å se på hvor mange ganger vi trenger å utvide $\mathrm{T}(n-42)$ før $n' - 42 \leqslant 42$. La oss kalle dette antallet k. Da følger det at

$$n - 42k = 42 \implies k = \tfrac{n}{42} - 1 \implies k = \Theta(n)$$

Dermed må $\mathrm{T}(n) = \Theta(n)$.

Oppgave 17

$\Theta(n^2 \lg n)$ er riktig.

Kommentar:
Først kan vi finne ut hvor mye arbeid den første løkken utfører.

$$n + (n-1) + (n-2) + ... + 2 + 1 = \sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2} = \Theta(n^2)$$

Dermed gjør den første løkken $\Theta(n^2)$ arbeid.
Neste løkke kaller så FUNCTION-E$(n/6)$ 36 ganger.
Vi får dermed rekurrensen

$$\mathrm{T}(n) = 36 \cdot \mathrm{T}(n/6) + \Theta(n^2)$$

som kan løses ved masterteoremet case 2. Vi får dermed $\mathrm{T}(n) = \Theta(n^2 \lg n)$

Oppgave 18

$\mathrm{T}(n) = \Theta(n^2 \lg n)$ er riktig.

Kommentar:
Med samme argument som da vi løste rekurrensen for FUNCTION-A(n) kan vi også her se bort i fra små konstantledd i argumentet til funksjonene. Dermed får vi rekurrensen

$$\mathrm{T}(n) = 4\mathrm{T}(n/2) + \Theta(n^2)$$

Vi kan da bruke masterteoremet case 2, og får at $\mathrm{T}(n) = \Theta(n^2 \lg n)$.

Merk: *I teorien vil forenklinger på formen* $\mathrm{T}(n/2 - 1)$ *til* $\mathrm{T}(n/2)$ *kunne føre til at svaret ikke blir helt riktig. I praksis skal det en del til for at dette skjer, men for å være helt sikker kan man vise ved induksjon at svaret man har kommet frem til faktisk stemmer.*

Oppgave 19

$\Theta(\lg n)$ er riktig.

Kommentar:
$\mathrm{T}(n/\pi)$ leddet er det leddet der argumentet synker saktest, så det er dette som avgjør treets høyde. I første iterasjon vil man ha n. I andre iterasjon vil man ha n/π. Tredje iterasjon vil ha n/π^2 osv. Grunntilfellet i rekurrensen treffer vi når $n = 1$, noe som skjer når

$$n/\pi^k = 1 \implies n = \pi^k \implies \log_\pi(n) = k$$

der k er hvilket nivå i rekursjonstreet vi befinner oss i. Når vi befinner oss nederst er $k = \log_\pi(n) = \Theta(\lg n)$, så $\Theta(\lg n)$ er høyden på treet.

Oppgave 20

$\Theta(\lg \lg n)$ er riktig.

Kommentar:
Siden oppgaven spør etter høyden på rekursjonstreet og ikke mengden arbeid som utføres kan vi sette opp en ny rekurrens som teller antall nivå i treet som følger:

$$\mathrm{H}(n) = \mathrm{H}(\sqrt{n}) + 1, \qquad \mathrm{H}(1) = 1.$$

Denne kan løses ved å bruke substutisjonen $n = 2^k$, som insatt gir

$$\mathrm{H}(2^k) = \mathrm{H}(2^{k/2}) + 1.$$

Om vi så definerer $\mathrm{S}(k) = \mathrm{H}(2^k)$ får vi da

$$\mathrm{S}(k) = \mathrm{S}(k/2) + 1.$$

På samme måte som i forrige oppgave kan vi da regne ut at høyden på rekursjonstreet til S er $\lg k$. Siden $n = 2^k$ er $k = \lg n$, og ved å ta logaritmen på begge sider får vi at rekursjonshøyden til treet blir $\lg k = \lg \lg n = \Theta(\lg \lg n)$.
En alternativ løsning på oppgaven er å se at etter k iterasjoner vil det rekursive kallet til T være gitt ved $\mathrm{T}(n^{1/2^k})$. En kan da finne hvor mange iterasjoner som kreves for å komme til gruntilfellet $n = 2$ ved å løse $n^{1/2^k} = 2$ for k og komme frem til samme resultat.

Oppgave 21

$\Theta((\lg(n))^2(\lg \lg n))$ er riktig.

Kommentar:
Vi starter med

$$\mathrm{T}(n) = 4 \cdot \mathrm{T}(\sqrt{n}) + (\lg n)^2$$

Deretter bruker vi substitusjonen $n = 2^k$:

$$\mathrm{T}(n) = \mathrm{T}(2^k) = 4 \cdot \mathrm{T}(\sqrt{2^k}) + (\lg(2^k))^2 = 4 \cdot \mathrm{T}(2^{k \cdot 1/2}) + k^2 = 4 \cdot \mathrm{T}(2^{k/2}) + k^2$$

Nå kan vi bruke substitusjonen $\mathrm{S}(k) = \mathrm{T}(2^k)$:

$$\mathrm{S}(k) = \mathrm{T}(2^k) = 4 \cdot \mathrm{T}(2^{k/2}) + k^2 = 4 \cdot \mathrm{S}(k/2) + k^2$$

Bruker man masterteoremet på $\mathrm{S}(k)$ får man at $\mathrm{S}(k) = \Theta(k^2 \lg k)$. Siden $\mathrm{T}(2^k) = \mathrm{S}(k)$ er også $\mathrm{T}(2^k) = \Theta(k^2 \lg k)$. Videre har vi at

$$n = 2^k \implies k = \lg n$$

. Hvis man setter inn dette i den opprinnelige rekurrensen får man

$$\mathrm{T}(n) = \mathrm{T}(2^{\lg n}) = \Theta((\lg n)^2 \cdot \lg \lg n)$$

Oppgave 22

$\mathrm{T}(n) = \Theta((\lg n)^{\log_{\pi/e} 2})$ er riktig.

Kommentar:
Vi starter med rekurrensen:

$$\sqrt{\lg n} + 2 \cdot \mathrm{T}(\sqrt[\pi]{n}) = \mathrm{T}(\sqrt[e]{n}) - \lg \lg n$$

Vi ønsker å få den på formen $\mathrm{T}(n) = a \cdot \mathrm{T}(f(n)) + \ldots$, der $f(n) < n$ siden argumentet i rekurrensen skal synke slik at det treffer grunntilfellet til slutt. Siden $\sqrt[\pi]{n} < \sqrt[e]{n}$ velger vi å isolere $\mathrm{T}(\sqrt[e]{n})$-leddet på venstresiden:

$$\mathrm{T}(\sqrt[e]{n}) = 2 \cdot \mathrm{T}(\sqrt[\pi]{n}) + \sqrt{\lg n} + \lg \lg n = 2 \cdot \mathrm{T}(\sqrt[\pi]{n}) + \Theta(\sqrt{\lg n})$$

Vi ønsker å få det på formen $\mathrm{T}(n) = \ldots$, så vi opphøyer n i e på begge sider.

$$\mathrm{T}(n) = 2 \cdot \mathrm{T}(n^{e/\pi}) + \Theta(\sqrt{\lg n^e}) = 2 \cdot \mathrm{T}(n^{e/\pi}) + \Theta(\sqrt{e \cdot \lg n})$$

$$= 2 \cdot \mathrm{T}(n^{e/\pi}) + \Theta(\sqrt{e} \cdot \sqrt{\lg n}) = 2 \cdot \mathrm{T}(n^{e/\pi}) + \Theta(\sqrt{\lg n})$$

Her kommer den klassiske substitusjonen $n = 2^k$ og $\mathrm{T}(n) = \mathrm{T}(2^k) = \mathrm{S}(k)$ som gir:

$$\mathrm{T}(2^k) = \mathrm{S}(k) = 2 \cdot \mathrm{T}(2^{k \cdot e/\pi}) + \Theta(\sqrt{\lg 2^k}) = 2 \cdot \mathrm{S}(e/\pi \cdot k) + \Theta(\sqrt{k})$$

Hvis vi bruker masterteoremet på $\mathrm{S}(k)$ får vi at $\log_{\pi/e}(2) > 1/2$, så dette er case 3, og vi får at $\mathrm{S}(k) = \mathrm{T}(2^k) = \Theta(k^{\log_{\pi/e}(2)})$, som gir $\mathrm{T}(n) = \mathrm{T}(2^{\lg n}) = \Theta((\lg n)^{\log_{\pi/e}(2)})$

Oppgave 23

```
# Hjelpefunksjon som finner minste verdi i rektangelet utspent av (x0, y0)
    og (x1, y1)
def minimum(x, x0, y0, x1, y1):
    minimal = float("inf")
    minimal_index = (-1, -1)
    for i in range(x0, x1+1):
        for j in range(y0, y1+1):
            if x[i][j] < minimal:
                minimal_index = (i, j)
                minimal = x[i][j]
    return minimal, minimal_index


# Største kiste i rektangelet utspent av (x0, y0) og (x1, y1)
# O(n^4) i gjennomsnittlig kjøretid
def _largest_cuboid(x, x0, y0, x1, y1):
    if (x0, y0) == (x1, y1):
        return x[x0][y0]
    minimal, minimal_index = minimum(x, x0, y0, x1, y1)

    # Volumet til det største prismet som spenner hele arealet
    maximal = (x1 - x0 + 1) * (y1 - y0 + 1) * minimal

    # Hvis ikke det største prismet spenner hele arealet, må det største
    prismet
    # enten være over, under, til venstre eller til høyre for det grunneste
    punktet
    if minimal_index[0] > x0:
        maximal = max(maximal, _largest_cuboid(x, x0, y0, minimal_index[0] -
     1, y1))

    if minimal_index[0] < x1:
        maximal = max(maximal, _largest_cuboid(x, minimal_index[0] + 1, y0,
    x1, y1))

    if minimal_index[1] > y0:
        maximal = max(maximal, _largest_cuboid(x, x0, y0, x1, minimal_index
    [1] - 1))
```

```
    if minimal_index[1] < y1:
        maximal = max(maximal, _largest_cuboid(x, x0, minimal_index[1] + 1,
    x1, y1))

    return maximal


def largest_cuboid(x):
    return _largest_cuboid(x, 0, 0, len(x) - 1, len(x) - 1)
```

Denne var ikke rask nok til å havne på highscore-listen, så jeg skal forklare en algoritme som faktisk var rask nok. Den er ikke basert på splitt-og-hersk og det går utenfor pensum, så det er kun for de som eventuelt er nysgjerrige.

https://www.sciencedirect.com/science/article/pii/0166218X84901240 sier at i en $n \times n$-binærmatrise er antallet potensielle rektangler (rektangler som bare består av enere og ikke kan utvides i lengden eller høyden) $O(n \lg n)$ i forventning. Vi starter med en liste over potensielle rektangler (som i starten kun er hele matrisen) og så itererer vi gjennom filsene i stigende rekkefølge og oppdaterer de potensielle rektanglene underveis, slik at man alltid har en liste over potensielle rektangler, der de potensielle rektanglene ikke inkluderer fliser vi allerede har iterert gjennom. Da trenger vi bare for hvert flis å sjekke hvilke potensielle rektangler den er i, og oppdatere maksimalt volum ut fra det.

Kjøretiden domineres av at vi må sjekke alle rektangler for hver flis, noe som gir en kjøretid på $\Theta(n^2 \cdot n \lg n) = \Theta(n^3 \lg n)$ i forventning. Siden antallet potensielle rektangler i verste tilfelle er $\Theta(n^2)$ får vi en verste kjøretid på $\Theta(n^4)$

En implementasjon av algoritmen:

```
def largest_cuboid(x):
    n = len(x)
    potential_rectangles = {((0, 0), (n - 1, n - 1))}
    coordinates = [(i, j) for i in range(n) for j in range(n)]
    coordinates = sorted(coordinates, key=lambda c: x[c[0]][c[1]])
    max_volume = 0
    for c in coordinates:
        to_split = []
        for rectangle in potential_rectangles:
            # Punktet ligger i rektangelet
            if rectangle[0][0] <= c[0] <= rectangle[1][0] and rectangle
    [0][1] <= c[1] <= rectangle[1][1]:
                to_split.append(rectangle)
                max_volume = max(max_volume, (rectangle[1][0] - rectangle
    [0][0] + 1) * (rectangle[1][1] - rectangle[0][1] + 1) * x[c[0]][c[1]])
        # Splitter rektangelet i potensielt fire deler: over, under,
        # til venstre og til høyre for flisa vi ser på.
        # Disse vil da også være potensielle rektangler.
        for rectangle in to_split:
            potential_rectangles.remove(rectangle)
            if c[0] > rectangle[0][0]:
                potential_rectangles.add(((rectangle[0][0], rectangle[0][1])
    , (c[0] - 1, rectangle[1][1])))
```

```
            if c[1] > rectangle[0][1]:
                potential_rectangles.add(((rectangle[0][0], rectangle[0][1])
, (rectangle[1][0], c[1] - 1)))
            if c[0] < rectangle[1][0]:
                potential_rectangles.add(((c[0] + 1, rectangle[0][1]), (
rectangle[1][0], rectangle[1][1])))
            if c[1] < rectangle[1][1]:
                potential_rectangles.add(((rectangle[0][0], c[1] + 1), (
rectangle[1][0], rectangle[1][1])))
    return max_volume
```

Oppgave 25

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.

Løsning på Kattis-oppgavene:

- Guess:
  Standard binærsøk

  ```
  import sys

  low = 1
  high = 1001

  while True:
      mid = (low + high) // 2
      print(str(mid), flush=True)
      response = sys.stdin.readline().strip()
      if response == "correct":
          break
      if response == "lower":
          high = mid
      else:
          low = mid
  ```

- Frosh Week:
  Modifisert mergesort. Se f. eks. https://www.geeksforgeeks.org/counting-inversions/
- Factor free:
  Splitt og hersk. Se https://people.bath.ac.uk/masjhd/NWERC/news/nwerc2017slides.pdf

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