

Obligatorisk oppgave 1

Teque

Oppgaver:

(a)

Algoritme: Legge til element bakerst i teque, antar array kan utvides

Input: Array A med n elementer og element x som skal legges bakerst i A

Output: Array A med x lagt til bakerst

Procedure push_back(A, x)

```
A[n+1] <- x
```

```
return A
```

Algoritme: Legge til element forrest i teque, oppdaterer alle indekser

Input: Array A med n elementer og element x som skal legges forrest i A

Output: Array A med x lagt til forrest

Procedure push_front(A, x)

```
for i <- n - 1 to 0 do
```

```
    A[i + 1] <- A[i]
```

```
A[0] <- x
```

```
return A
```

Algoritme: Legge til element midterst i teque

Input: Array A med n elementer og element x som skal legges midterst i A

Output: Array A med x lagt til midterst

Procedure push_middle(A, x)

```
m = (n + 1) // 2 # «//» er heltallsdivisjon
```

```
for i <- n - 1 to m do
```

```
    A[i+1] <- A[i]
```

```
A[m] <- x
```

```
return A
```

Algoritme: Returnerer elementet i array A gitt indeks i

Input: Array A med n elementer og element med indeks i

Output: Element med indeks i

Procedure get(A, i)

```
return A[i]
```

- (b) Ligger vedlagt. Jeg var veldig lat og brukte Python sin innebygde liste, så jeg inkorporerte ikke algoritmene som beskrevet oven. ikke finn på å kjør for input_100000
- (c) `push_back` er konstant $O(1)$ da Python har kontroll på siste element.
`push_front` er lineær $O(n)$, går gjennom alle indeksene en gang
`push_middle` er lineær $O(n)$, går gjennom halvparten av indeksene en gang
`get` er lineær hvis den må scrolle gjennom indeksene, men konstant hvis ikke
- (d) Det gir ikke helt mening å snakke om kjørekompleksitet når man har en satt mengde data, kjøretiden vil alltid være konstant.

Korrekthet

```
import merge, insertion

list = [8, 4, 1, 5, 9, 9, 2, 12, 10, 35, 23]

print(merge.sort(list))
print(insertion.sort(list))
```

Sammenligninger, bytter og tid

- Hvordan bytter og sammenligninger er målt:

Eksperimentér

- I hvilken grad stemmer kjøretiden overens med kjøretidsanalysene? (store O) for de ulike algoritmene?

Insertion har $O(N^2)$ mens Merge har $O(n \log(n))$ i store- O som vil være sant på vilkårlig stor data med tilfeldige tall, men hvis listen er nesten-sortert i utgangspunktet vil Insertion ha en fordel. Merge vil uansett og alltid kjøre sin fulle tid fordi den halverer lister og sublister hele veien, og sammenligner alle elementene mens den rekonstruerer. Mens hvis elementene i listen du sender til Insertion er nesten sorterte vil den ikke trenge å gjøre særlig mange sammenligninger, som kutter ned kjøretiden drastisk.

- Hvordan er antall sammenligninger og antall bytter korrelert med kjøretiden?

Som nevnt oven – For merge sort har det lite å si fordi den sammenligner alt uansett, men for insertion har det alt å si. Jo flere ting den må sammenligne jo tregere går det.

- Hvilke sorteringsalgoritmer utmerker seg positivt når n er veldig liten? Og når n er veldig stor?

For veldig små har det ikke veldig mye å si, men Insertion er bra på små og nesten sorterte lister mens Merge vil være overlegen på store.

- Hvilke sorteringsalgoritmer utmerker seg positivt for de ulike inputfilene?

For nesten-sorterte er Insertion utmerket, for tilfeldige er Merge utmerket.

- Har du noen overraskende funn å rapportere?

For meg var det kontraintuitivt å lese at Insertion er veldig bra på små lister og jeg forventet av resultatene å se at det stemte, men i resultatene vi fikk slo Merge den fortsatt ganske greit på små lister.