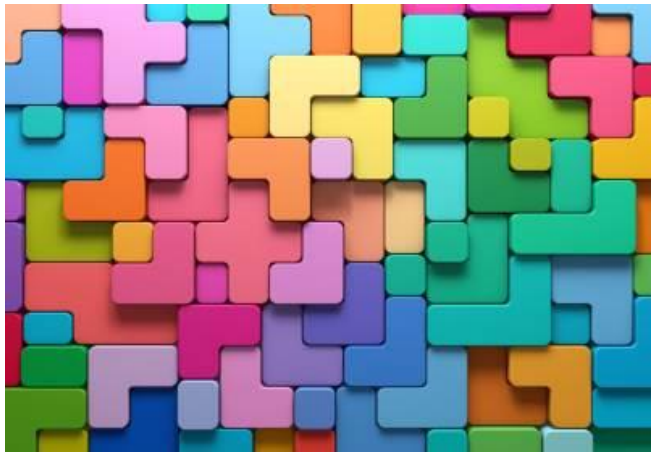
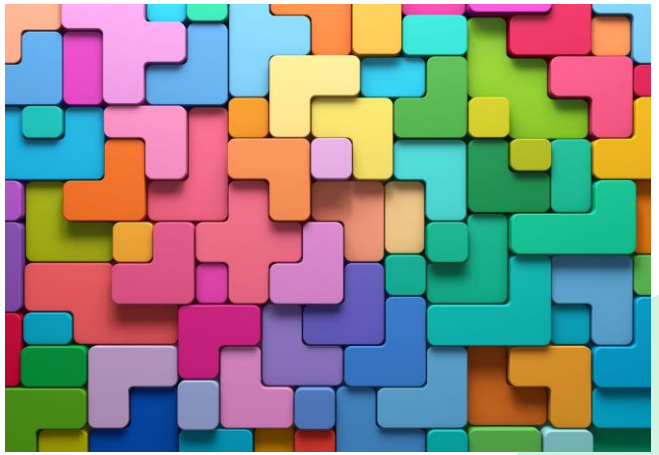
LENKEDE LISTER OG ALGORITMEANALYSE

FORELESNING 12

ONSDAG 2/10

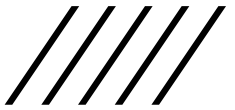




(bilder generert av bing image creator)

Ekstramateriale om C++

- Gratis lærebok (gjennom UiO)
- Ikke pensum, men nyttig om man ønsker en ekstra kilde



International English Spelling Chart

United States	Canada	United Kingdom	Australia
color	colour	colour	colour
center	centre	centre	centre
realize	realize	realise/ize	realise
analyze	analyze	analyse	analyse
traveling	travelling	travelling	travelling
defense	defence	defence	defence
computer program, concert program	computer program, concert program	computer program, concert programme	computer program, concert program
gray	grey	grey	grey
fulfill	fulfil(l)	fulfil	fulfil
aging	ag(e)ing	ageing	ag(e)ing

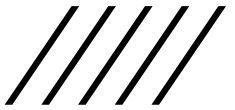
Kilde: lumida.ca

Til prosjekt 2

- Bruk **-std=c++14** for å bruke C++14-standarden (fra 2014) når man kompilerer
- Husk å teste at koden kompilerer og kjører på en IFI-maskin – det holder ikke at det funker på *din* maskin!
- Enkleste måte er å bruke **scp** til å kopiere kildekoden og **ssh** til å logge inn og kompilere / kjøre etterpå
- (Dette "livekodes" i forelesningen – se evt. opptak)

Livekoding over flere forlesninger

- Fortsett der vi slapp i forelesning 11
- Hvis du ikke var til stede eller ikke finner koden:
  - Lag en fork av dette repo'et (public)
  - Direkte link: https://github.uio.no/oddps/IN1910-live/fork



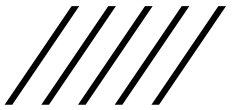
LIVEKODING:

SINGLY LINKED LIST

(FORTS.)

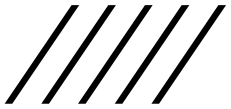
Viktig forskjell på variabel og peker

- En variabel (**Node ny_node**) ryddes opp når den ikke lenger er definert (f.eks. funksjonen der den defineres er ferdig)
- En peker (**Node* ny_node**) ryddes ikke opp av seg selv
- Hvis vi fortsetter å bruke en peker med adressen til en variabel som ikke lenger er definert, kan det være et helt annet objekt der enn vi tror (minnet gjenbrukes når det er ledig!)
- Kompilatoren vil ofte advare oss hvis vi prøver på dette



Minnelekkasje

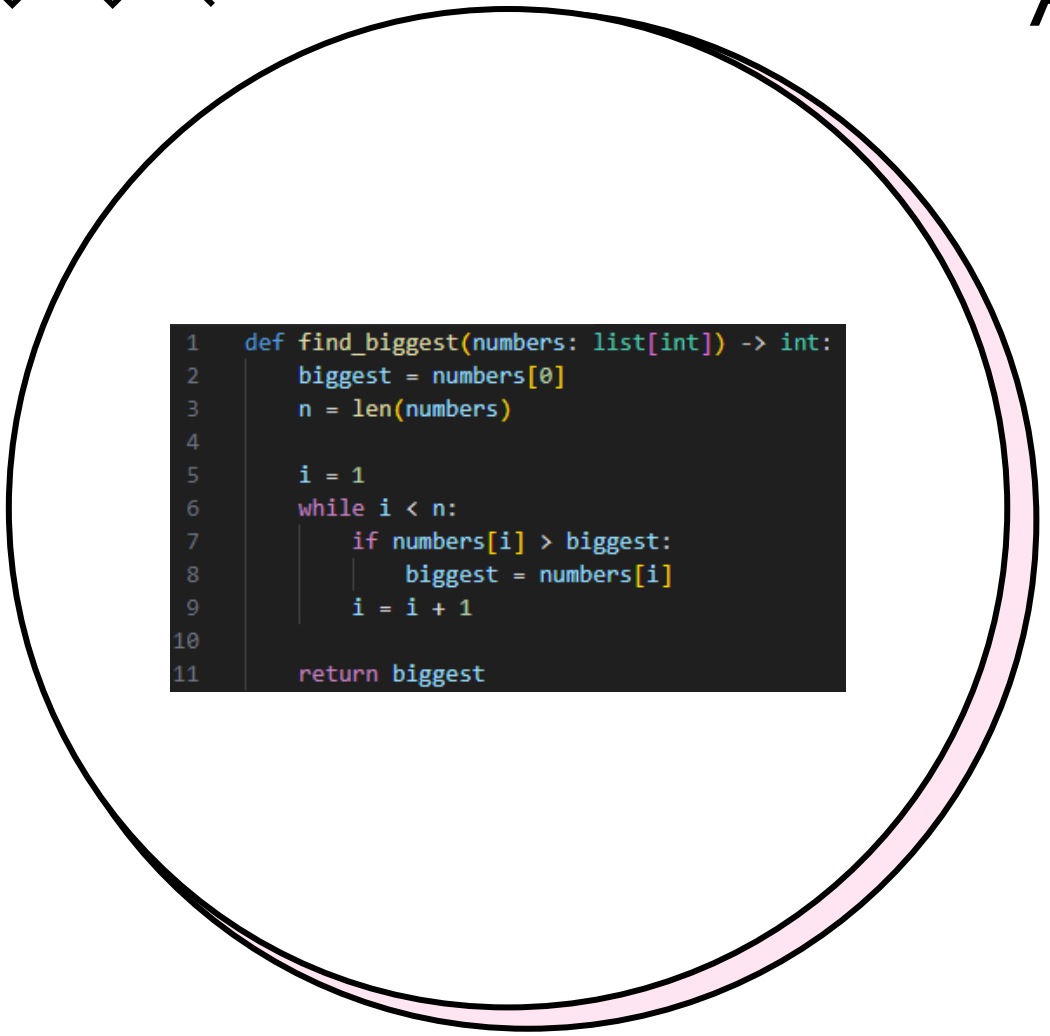
- Vi kan lage et objekt direkte med en peker isteden:
  Node* ny_node = new Node();
- Men da må vi rydde opp selv: **delete ny_node**
- Hvis vi ikke rydder opp, kan vi få en *memory leak* – vi reserverer mer og mer minne (med **new**) uten å gi beskjed om at disse minneadressene ikke trengs lenger
- (Livekodet eksempel til skrekk og advarsel)



Dokumentasjon: Doxygen

- Lastes ned her
- (Hvordan bruke? Livekodes i forelesning)

Algoritmeanalyse



```python
def find_biggest(numbers: list[int]) -> int:
    biggest = numbers[0]
    n = len(numbers)

    i = 1
    while i < n:
        if numbers[i] > biggest:
            biggest = numbers[i]
        i = i + 1

    return biggest
```

- Hvordan skalerer algoritmen med antall elementer?
- Se på antall operasjoner:
  - Indeksering […]
  - Tilordning =
  - Sammenligning >, <
  - Addisjon +
  - len(…)
  - Returverdi

n = 1 element

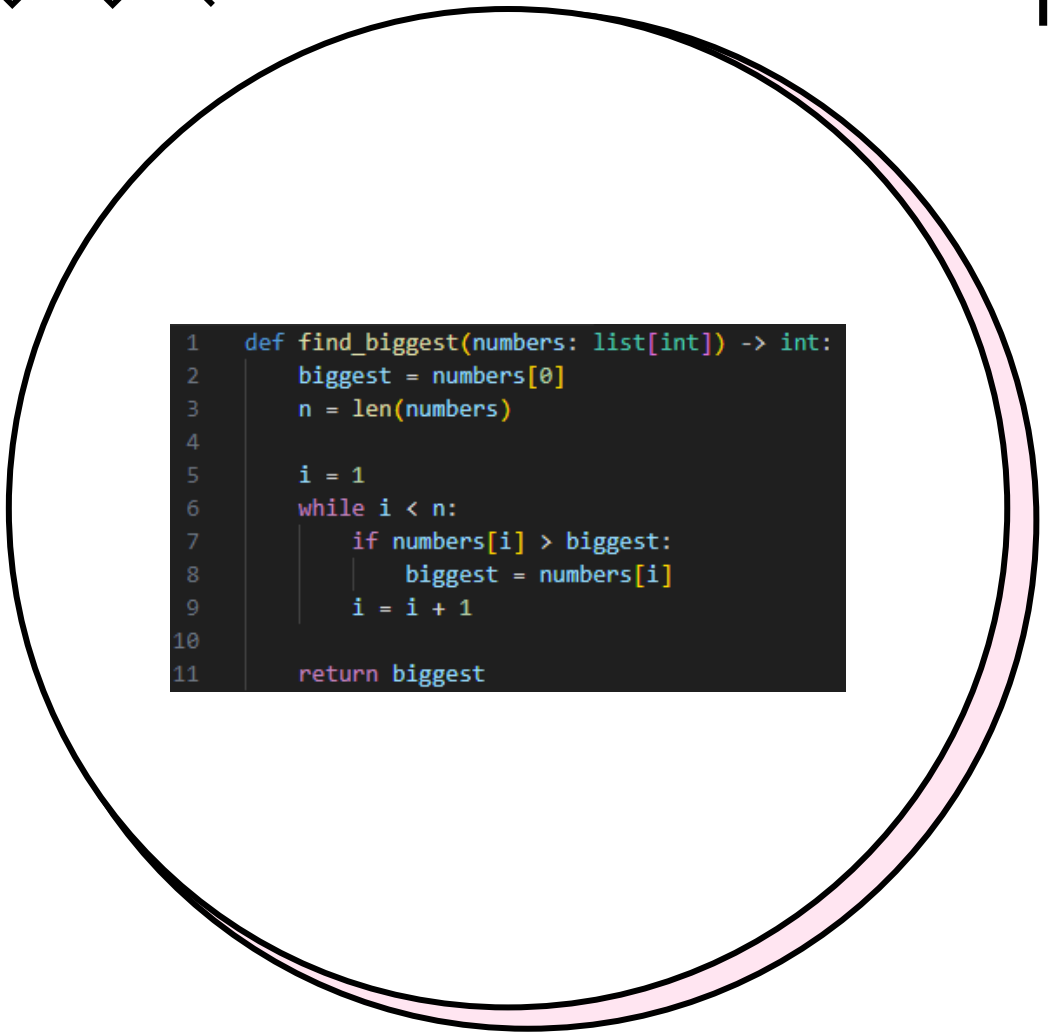
```python
def find_biggest(numbers: list[int]) -> int:
    biggest = numbers[0]
    n = len(numbers)

    i = 1
    while i < n:
        if numbers[i] > biggest:
            biggest = numbers[i]
        i = i + 1

    return biggest
```

- biggest = numbers[0]
  2 operasjoner totalt
- n = len(numbers)
  1 operasjon totalt
- i = 1
  1 operasjon totalt
- while i < n:
  1 operasjon totalt
- if numbers[i] > biggest:
  0 operasjoner totalt
- biggest = numbers[i]
  0 operasjoner totalt
- i = i + 1
  0 operasjoner totalt
- return biggest
  1 operasjon totalt
- Sum: **6**

n = 2 elementer



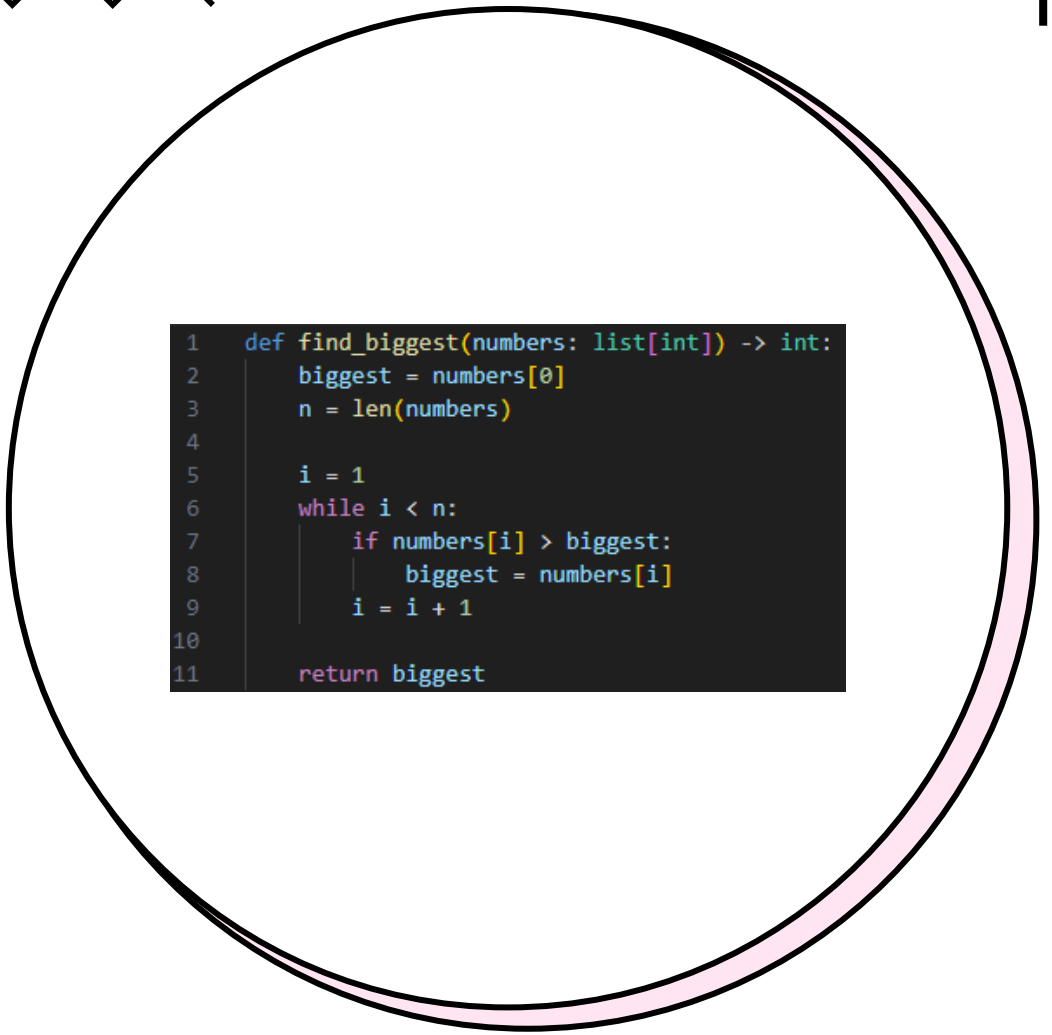
```python
def find_biggest(numbers: list[int]) -> int:
    biggest = numbers[0]
    n = len(numbers)

    i = 1
    while i < n:
        if numbers[i] > biggest:
            biggest = numbers[i]
        i = i + 1

    return biggest
```

- biggest = numbers[0]
  2 operasjoner totalt
- n = len(numbers)
  1 operasjon totalt
- i = 1
  1 operasjon totalt
- while i < n:
  2 operasjoner totalt
- if numbers[i] > biggest:
  2 operasjoner totalt
- biggest = numbers[i]
  0 - 2 operasjoner totalt
- i = i + 1
  2 operasjoner totalt
- return biggest
  1 operasjon totalt
- Sum: **11 – 13**

n = 3 elementer



```python
def find_biggest(numbers: list[int]) -> int:
    biggest = numbers[0]
    n = len(numbers)

    i = 1
    while i < n:
        if numbers[i] > biggest:
            biggest = numbers[i]
        i = i + 1

    return biggest
```

- biggest = numbers[0]
  2 operasjoner totalt
- n = len(numbers)
  1 operasjon totalt
- i = 1
  1 operasjon totalt
- while i < n:
  3 operasjoner totalt
- if numbers[i] > biggest:
  4 operasjoner totalt
- biggest = numbers[i]
  0 - 4 operasjoner totalt
- i = i + 1
  4 operasjoner totalt
- return biggest
  1 operasjon totalt
- Sum: **16 – 20**

Hvordan scaler dette med n?

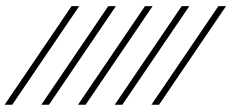
- n = 1 → 6 - 6
- n = 2 → 11 - 13
- n = 3 → 16 - 20
- n = 4 → 21 - 27

```python
def find_biggest(numbers: list[int]) -> int:
    biggest = numbers[0]
    n = len(numbers)

    i = 1
    while i < n:
        if numbers[i] > biggest:
            biggest = numbers[i]
        i = i + 1

    return biggest
```

- "Best case": $6 + 5(n-1) = 5n + 1$
- "Worst case": $6 + 7(n-1) = 7n - 1$

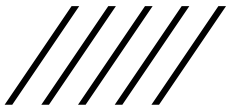


Det går an å lese dette mer direkte

```
1   def find_biggest(numbers: list[int]) -> int:
2       biggest = numbers[0]
3       n = len(numbers)
4
5       i = 1
6       while i < n:
7           if numbers[i] > biggest:
8               biggest = numbers[i]
9           i = i + 1
10
11      return biggest
```

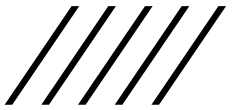
- 2
- 1
- 1
- n
- 2(n-1)
- 0 – 2(n-1)
- 2(n-1)
- 1

- "Best case": $5 + n + 4(n-1) = 5n + 1$
- "Worst case": $5 + n + 6(n-1) = 7n - 1$



O-notasjon

- Når kjøretiden er $an + b$ har vi **lineær** scaling med n
- Leddet som dominerer når n blir stort er det vi fokuserer på
- Vi sier at skaleringen er av orden $O(n)$ for denne algoritmen



Big O	Name
$\mathcal{O}(1)$	constant
$\mathcal{O}(\log n)$	logarithmic
$\mathcal{O}(n)$	linear
$\mathcal{O}(n \log n)$	loglinear/quasilinear
$\mathcal{O}(n^2)$	Quadratic
$\mathcal{O}(n^3)$	Cubic
$\mathcal{O}(n^k)$	Polynomial
$\mathcal{O}(e^n)$	Exponential
$\mathcal{O}(n!)$	Factorial

Note that an algorithm can also have fractional exponent scaling, such as $\mathcal{O}(n^{1.5})$, but this is rare.

Etter forelesningen

- Repo for prosjekt 2 er ute
- Husk å kopiere filer du trenger (fra annet repo) før du/dere begynner å kode (se oppgavetekst for detaljer)
- Hvis du er plassert alene på gruppe og ikke ønsker dette, ta kontakt på e-post