



Uitwerkingen

1. Pseudo-code van insertion sort van slides 6:

```
Algorithm insertionSort( $A, n$ ):  
  for  $j := 1$  to  $n - 1$  do  
     $key := A[j]$   
     $i := j - 1$   
    while  $i \geq 0$  and  $A[i] > key$  do  
       $A[i + 1] := A[i]$   
       $i := i - 1$   
     $A[i + 1] := key$ 
```

Best-case input-rijtje: een rijtje dat al gesorteerd is, bijvoorbeeld [1, 2, 3, 4, 5, 6, 7, 8, 9, 10].

Worst-case input-rijtje: een rijtje dat andersom gesorteerd is, bijvoorbeeld [10, 9, 8, 7, 6, 5, 4, 3, 2, 1].

2. Insertion sort (zoals op de slide) is een in-place sorteeralgoritme want we gebruiken behalve de ruimte voor het input-array alleen iets extra's voor bijvoorbeeld *key*.
3. Pseudo-code voor aflopende insertion sort:

```
Algorithm insertionSortOmgekeerd( $A, n$ ):  
  for  $j := 1$  to  $n - 1$  do  
     $key := A[j]$   
     $i := j - 1$   
    while  $i \geq 0$  and  $A[i] < key$  do  
       $A[i + 1] := A[i]$   
       $i := i - 1$   
     $A[i + 1] := key$ 
```

We gaan nu de correctheid van insertionSortOmgekeerd aantonen. We doen dat door gebruik te maken van een *invariant*, dat wil zeggen, een eigen-

schap die behouden blijft onder het uitvoeren van een stukje van het algoritme. Zie eventueel ook de link bij extra materiaal aan het eind van slides 6.

In dit geval gebruiken we de volgende invariant:

Na k iteraties van de for-loop is het sub-array $A[0 \dots k - 1]$ een (aflopend) gesorteerde permutatie van datzelfde stuk van het input-array.

We bewijzen de invariant met inductie. Basisgeval: $k = 0$. Zonder een enkele for-loop te hebben uitgevoerd is het sub-array $A[0]$ inderdaad een gesorteerde permutatie van het input-array.

Inductiestap: stel de invariant geldt voor k . Voordat we de $k + 1$ ste iteratie van de for-loop ingaan, is $A[0 \dots k]$ dus een gesorteerde permutatie van dat stuk van het input-array. Als $A[k + 1]$ groter is dan alle elementen in $A[0 \dots k]$ dan wordt de while-loop voor de laatste keer uitgevoerd met $i = 0$. Dan wordt wat voor de loop op $A[k + 1]$ stond, oftewel *key*, vooraan geplaatst; hetgeen inderdaad een gesorteerd aflopend sub-array $0[k + 1]$ levert. Als de while-loop voor de laatste keer wordt uitgevoerd omdat $A[i] \geq key$ voor zekere i , dan is alles in $A[i + 1 \dots j - 1]$ één plaats naar rechts opgeschoven. Dan kunnen we de *key*, dus wat voor de loop op $A[j]$ stond, nu op $A[i + 1]$ zetten. Alles rechts ervan is kleiner dan *key* en (oplopend) gesorteerd, en alles links ervan is groter dan *key* en (oplopend) gesorteerd. Het geheel $A[0 \dots k + 1]$ is dus ook (oplopend) gesorteerd.

4. Insertion sort als recursieve procedure:

Algorithm insertionSortRec(A):

```
 $n := A.size()$ 
if  $n = 0$  then
    return  $A$ 
if  $n > 0$  then
     $B := \text{insertionSortRec}(A[0 \dots (n - 2)])$ 
    return insert( $B, A[n - 1]$ )
```

met als hulp-procedure:

Algorithm insert(A, x):

```
 $i := A.size()$ 
while  $i \geq 0$  and  $A[i] > x$  do
     $A[i + 1] := A[i]$ 
     $i := i - 1$ 
 $A[i + 1] := x$ 
```

Hier is A een gesorteerd array, en x is wat we op de goede plek willen toevoegen aan A . Dus `insert` is eigenlijk een stukje van de code van `insertionSort`. De `for`-loop in `insertionSort` is nu vervangen door recursie.

Wat betreft de tijdscomplexiteit: het werk zit in `insert`. De tijdscomplexiteit van `insert` wordt bepaald door de lengte van het input-array A . De methode `insert` wordt aangeroepen voor arrays met lengte $1, \dots, n$, met n de lengte van het input-array A van `insertionSortRec`. De tijdscomplexiteit van die laatste methode is wordt dus bepaald door $1 + \dots + n$. Dus `insertionSortRec` $\in \mathcal{O}(n^2)$.

Of met een recurrente betrekking voor de tijdscomplexiteit:

$$T(n) = \begin{cases} 1 & \text{als } n = 0 \\ T(n-1) + n & \text{als } n > 0 \end{cases}$$

5. (a) De inversies van $[2, 3, 8, 6, 1]$: $(2, 1), (3, 1), (8, 1), (8, 6), (6, 1)$.
 (b) Een permutatie van $[1, \dots, 10]$ met zoveel mogelijk inversies: $[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]$. Het maximum aantal inversies van $[1, \dots, n]$ is $(n-1) + (n-2) + \dots + (n-(n-1)) = \frac{n(n-1)}{2}$, en is zelfs in $\Omega(n^2)$.
 (c) Zie bijvoorbeeld het array $[2, 3, 8, 6, 1]$. Dit bevat 5 inversions. Elke inversion correspondeert met een opschuif-actie in de uitvoering van `insertionSort`. Dit geldt ook in het algemeen: bij het uitvoeren van `insertionSort` op een array met k inversions moet er k keer een $A[i]$ één plaats opgeschoven worden.
6. Een naïef algoritme in $\mathcal{O}(n^2)$: vergelijk de eerste met alle andere, vergelijk de tweede met alle andere rechts ervan, etcetera totdat we uiteindelijk de een-na-laatste met de laatste vergelijken. Dit is in $\mathcal{O}(n^2)$ omdat we $n-1 + \dots + (n-(n-1))$ vergelijkingen maken.

Een iets minder naïef algoritme in $\mathcal{O}(n \log(n))$: we gebruiken merge sort, maar met een iets opgevoerde versie van `merge`. We voegen namelijk een globale variable toe voor een teller die bijhoudt hoe vaak het geval voorkomt dat het eerste element van A groter is dan het eerste element van B (in de notatie van slides 6), dus het ‘else’ geval.

7. De pseudo-code van selection sort zoals op slides 6:

Algorithm `selectionSort(A, n)`:

```

for  $i := 0$  to  $n - 2$  do
     $m := i$ 
    for  $j = i$  to  $n - 1$  do
        if  $A[j] < A[m]$  then  $m := j$ 
     $x := A[m]$ 
     $A[m] := A[i]$ 
     $A[i] := x$ 

```

De $n - 2$ is correct, want als je de $n - 1$ kleinste elementen uit het array van n elementen al op de eerste $n - 1$ posities hebt geplaatst, staat het laatste, en grootste, element op positie $n - 1$ op de goede plaats.

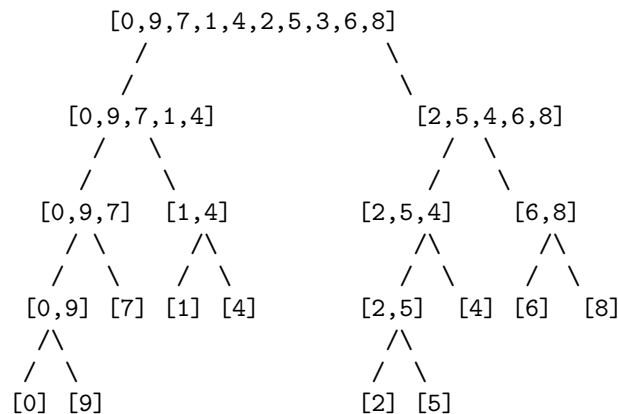
8. Eigenlijk maakt het voor selection sort niet zoveel uit. Als het input-rijtje aflopend is, dan is het aantal aanpassingen van de waarde van m maximaal. Maar voor het aantal keren dat de loops worden doorlopen doet de vorm van de input er niet toe.

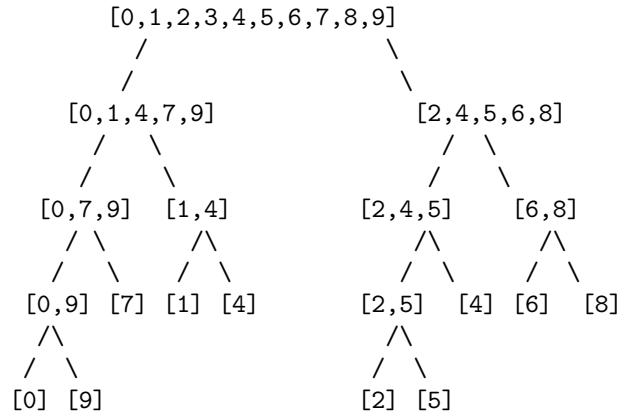
9. We tonen aan: na k iteraties staan de kleinste k elementen van het input-array gesorteerd op indexen $0, \dots, k - 1$.

Voor $k = 1$: De index van het kleinste element wordt opgeslagen in de variabele m , en vervolgens wordt $A[0]$ verwisseld met $A[m]$. Dus de loop-invariant geldt na één keer uitvoeren van de loop.

Voor $k > 1$: In de tweede for-loop worden de elementen vanaf index $k - 1$ tot en met het einde bekeken. De kleinste daarvan wordt verwisseld met $A[k - 1]$. Daarmee staan de kleinste k elementen van de input-array gesorteerd in $A[0 \dots (k - 1)]$.

10. Bepalend voor de tijdscomplexiteit zijn de twee geneste for-loops. We gaan zoals gebruikelijk uit van elementaire operaties in $\mathcal{O}(1)$. Dan wordt de orde van grootte bepaald door de som $(n - 1) + \dots + 1$: de eerste uitvoering van de tweede for-loop heeft $n - 1$ stappen, de tweede $n - 2$, etctera. Daarmee is de worst-case tijdscomplexiteit in $\mathcal{O}(n^2)$.
11. De merge-sort-boom voor het sorteren van de rij $[0, 9, 7, 1, 4, 2, 5, 3, 6, 8]$: eerst een boom met naar beneden de splitsing, daarna een boom waarin het samenvoegen met merge naar boven loopt. Dus notatie zoals in het boek dit keer.





12. Er zijn ten minste 10 en ten hoogste 29 vergelijkingen nodig.
 13. We bekijken de recurrente betrekking voor merge sort:

$$T(n) = \begin{cases} 1 & \text{als } n = 1 \\ 2T(\frac{n}{2}) + n & \text{als } n > 1 \end{cases}$$

We lossen deze recurrente betrekking op met de ‘substitutie methode’.
 Voor grote n :

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + n \\ &= 2(2T(\frac{n}{4}) + \frac{n}{2}) + n \\ &= 4T(\frac{n}{4}) + 2n \\ &= 4(2T(\frac{n}{8}) + \frac{n}{4}) + 2n \\ &= 8T(\frac{n}{8}) + 3n \\ &= \dots \\ &= 2^i T(\frac{n}{2^i}) + in \end{aligned}$$

Substitueer: $i = \log(n)$. Dan vinden we:

$$T(n) = n + n \log(n)$$

14. De definitie van $T(n)$ is eigenlijk niet precies goed voor oneven n . We kunnen T dan wel als volgt geven:

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + n$$

Voor elke n die geen 2-macht is geldt: er is een k met $2^{k-1} < n < 2^k$. Dan geldt $T(2^{k-1}) < T(n) < T(2^k)$ want T is een continue, stijgende, functie. Daarmee vinden we: $T(n) \in \mathcal{O}(n \log(n))$, en eigenlijk zelfs $T(n) \in \Theta(n \log(n))$.

15. Vergelijk met slides 1.

```
Algorithm arrayMax( $A, n$ ):  
  if  $n = 0$  then  
    return error  
  if  $n = 1$  then  
    return  $A[0]$   
  if  $n > 1$  then  
     $m :=$  arrayMax( $A[0 \dots (n - 2)], n - 1$ )  
    if  $m > A[n - 1]$  then  
      return  $m$   
    else  
      return  $A[n - 1]$ 
```

De tijdscomplexiteit van dit algoritme kan beschreven worden met de functie T gegeven met de volgende recurrente betrekking:

$$T(n) = \begin{cases} 1 & \text{als } n = 1 \\ T(n-1) + 1 & \text{als } n > 1 \end{cases}$$

We lossen deze recurrente betrekking op met de ‘substitutie methode’.
Voor grote n :

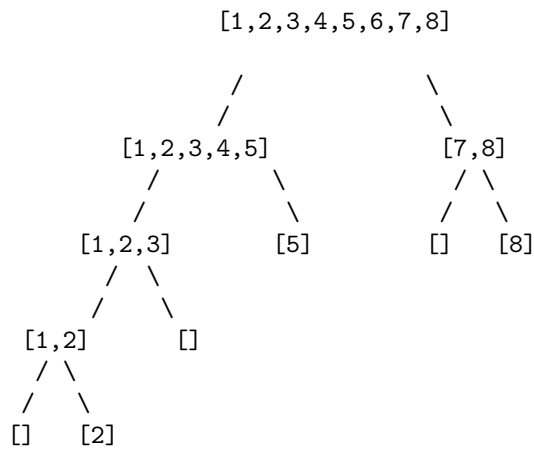
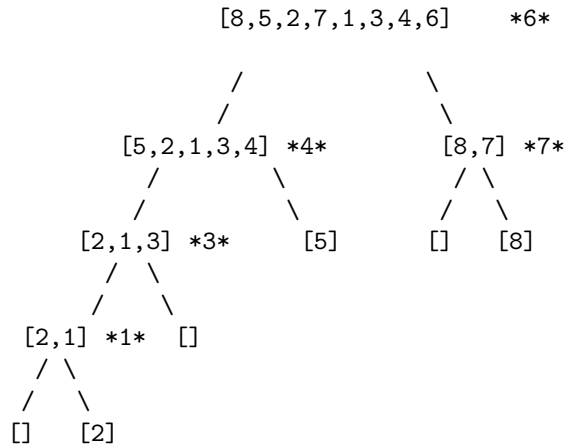
$$\begin{aligned} T(n) &= T(n-1) + 1 \\ &= T(n-2) + 1 + 1 \\ &= T(n-3) + 3 \\ &= \dots \\ &= T(n-i) + i \end{aligned}$$

We substitueren $i = n - 1$. Dan vinden we:

$$T(n) = 1 + n - 1 = n$$

Dus de tijdscomplexiteit van arrayMax is in $\mathcal{O}(n)$.

16. De quick sort boom: eerst een boom met naar beneden de partities, daarna een boom waarin het samenvoegen naar boven loopt. Dus notatie zoals in het boek dit keer.



17. Een voorbeeld van een worst-case input-rijtje ter lengte 7 voor quick sort waarbij als pivot het middelste element wordt gekozen: $[6, 2, 4, 1, 5, 3, 7]$.

In het algemeen geldt dat ook in het geval dat de pivot (min of meer) het middelste element van het rijtje is, de worst-case tijdscomplexiteit van quick sort in $\mathcal{O}(n)$ is. Want ook is de diepte van de quick sort boom van orde n , in het worst-case geval.

18. Pseudo-code voor het omdraaien van een sub-array:

Algorithm reverseArray(A, i, j):

```
left := i
right := j
while left < right do
    t := A[left]
    A[left] := A[right]
    A[right] := t
    left := left + 1
    right := right - 1
```

Je kunt nog een error melden als $i > j$, of $i < 0$, of $j > A.size() - 1$.

Toepassen op $[4, 3, 6, 2, 5]$ levert:

```
[4, 3, 6, 2, 5]
[5, 3, 6, 2, 4]
[5, 2, 6, 3, 4]
```

19. Voor merge sort maakt het niet uit dat onze input deze speciale vorm heeft; het algoritme blijft in $\mathcal{O}(n \log(n))$.

Met quick sort kiezen we als pivot eerst 0 of 1, en dan in de volgende laag van de quick sort boom de andere. De quick sort boom is dus maar 2 lagen diep.

Alternatief: loop door het array heen en tel de 1-en. Plaats vervolgens het juiste aantal 0-en aan het begin en het juiste aantal 1-en erachter. Dit is in $\mathcal{O}(n)$.

20. Als het input-rijtje gesorteerd is, en de pivot min of meer het midden van het rijtje is, dan is quick sort in $\mathcal{O}(n \log(n))$. Dit komt omdat een pivot in het midden een gesorteerd rijtje mooi in twee min of meer gelijke delen splitst. Daardoor is de diepte van de quick sort boom in dit geval van orde $\log(n)$.

21. Een gesorteerd rijtje.

22. Eerste stap: sorteer het input-rijtje; dit kan in $\mathcal{O}(n \log n)$. Tweede stap: loop door het gesorteerde rijtje heen en haal dubbele weg.

Pseudo-code nog toevoegen.