# Overzicht sorteeralgoritmen

Jelle De Bock

schakelprogramma industrieel ingenieur informatica 2016

## Insertion sort

### Werkwijze

- Er bestaat steeds een gesorteerd en een ongesorteerd deel
- Per iteratie wordt het eerste niet gesorteerde element genomen en wordt deze correct in het gesorteerde deel ingevoegd

### Code

```
void insertion_sort(vector<T> &v){
    for(int i=0; i<v.size()-1; i++){
        T help;
        //Voeg element i toe aan het gesorteerde deel
        //(de elementen voor i staan in volgorde)
        help = std::move(v[i]);
        int j = i-1;
        while(j>=0 && help<v[j]){
            //help is kleiner dan het voorgaande element dus
            //laat element op j plaats maken (schuif op)
            v[j+1] = std::move(v[j]);
            j--;
        }
        //na de voorgaande lus hebben we de positie van help gevonden
        //merk op: j+1 we zijn namelijk 1tje te ver gegaan in de while lus
        v[j+1] = std::move(help);
    }
}
```

### Tijdscomplexiteit

- **Slechtste geval** : de array staat helemaal omgekeerd gesorteerd
  $\Theta(n^2)$
- **Gemiddelde geval** : we moeten steeds ongeveer half de reeds gesorteerde array doorlopen
  $\Theta(n^2)$
- **Beste geval** : de array is reeds gesorteerd, we voeren slechts de buitenlus uit.
  $\Theta(n)$

## Shellsort

### Werkwijze

- Pas het principe van insertion sort toe, maar sorteer "deelarrays" bestaande uit elementen die een x aantal posities uit elkaar verwijderd liggen.
- Verminder systematisch de afstand x tot de posities, totdat je in principe een "klassieke insertion sort" toepast
- Vermits vele getallen niet maar 1 enkele positie moeten opgeschoven worden is het dus "slim" om deze meteen verder te laten opschuiven.

De redenering van deze werkwijze is dat de array bij de laatste iteratie al quasi gesorteerd is en de binnenste lus dus een pak minder zal uitgevoerd worden.

**De increments**
Over de increments is menig inkt gevloeid, maar onderstaande increments zijn veruit de belangrijkste:

- Sedgewick

$$k_i = 9 \cdot 2^i - 9 \cdot 2^{\frac{i}{2}} + 1, even, i \geq 0$$

$$k_i = 8 \cdot 2^i - 6 \cdot 2^{\frac{i+1}{2}} + 1, oneven, i \geq 0$$

- Tokuda

$$\lceil 9\frac{\frac{9}{4}^i - 4}{5} \rceil; \frac{9}{4} \cdot k_i < n$$

### Code

```
void shellsort(vector<T> &v){
    //Dalend: shellreeks, tokuda, sedgewick, ...
```

```
    std::stack<int> reeks = ...;
    int incr = reeks.pop();

    while(incr>=1){
        for(int i = incr; i<v.size(); i++){
            //sorteer reeksen van i tot en met size met
            //increment plaatsen tussen de elementen

            //Identiek aan insertion sort
            T help = v[i];
            //we springen telkens incr positie(s) ipv 1 positie
            int j = i-incr;
            while(j>=0 && help < v[j]){
                //schuif j incr posities naar achter
                v[j+incr] = std::move(v[j]);
                j-=incr;
            }
            //plaats het element op zijn juist plaats
            v[j+incr]=std::move(help);
        }
        incr = reeks.pop();
    }
}
```

### Tijdscomplexiteit

Tot op heden blijft de volledige tijdscomplexiteit van Shellsort nog steeds een mysterie. De optimale reeks is trouwens ook nog steeds niet gevonden.

- Shellreeks
$$\{\lfloor\frac{n}{2}\rfloor, \cdots, k_i, k_{i-1}\}$$
(deel telkens voorgaande door 2), is de complexiteit nog steeds $\Theta(n^2)$.
- Sedgewick reeks
Slechtste reeks: $\Theta(n^{\frac{4}{3}})$, gemiddeld $\Theta(n^{\frac{7}{6}})$
- Tokuda reeks nog iets beter ... (! zie cursus, staat daar zo letterlijk ;-) )

## Selection sort

### Principe

- Ook hier bestaat er steeds een gesorteerd en ongesorteerd deel.
- Zoek het laagste element

- Swap het laagste element met het eerste niet gesorteerde element in de array.
- Herhaal tot er geen ongesorteerde elementen meer zijn.

## Code

```
void selection_sort(vector<T> &v){
    //we gaan per iteratie op zoek naar het maximum
    //ipv het minimum
    for(int i=v.size()-1;i>0;i--){
        int max_index = i;
        //tweede loop om op zoek te gaan naar het maximum
        for(int j=0; j<i; j++){
            if(v[j] > v[max_index]){
                //markeer element j als max_index
                max_index = j;
            }
        }
        //verwissel het element dat op i staat
        //met het gevonden maximum (op positie max_index)
        T help = std::move(v[i]);
        v[i] = std::move(v[max_index]);
        v[max_index] = std::move(help);
    }
}
```

## Tijdscomplexiteit

Dit algoritme is verre van efficiënt. $\Theta(n^2)$ (tijdscomplexiteit) en $O(n^2)$ (aantal verplaatsingen).
Het algoritme wordt pas efficiënt van zodra je een betere selectieprocedure gaat gebruiken. (zie: heapsort)