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)

Heap sort

Principe

Een **heap** is een minimale/maximale binaire boom (=maximaal 2 opvolgers) die zodanig geordend is dat elk kindelement kleiner/groter is dan zijn ouderelement.

- **Een kind** (ook blad) is een knoop die geen opvolgers heeft
- **De wortel** is de eerste knoop bovenaan in de hierarchie van de boom
- Het **niveau** wordt geteld van boven naar beneden (wortel is niveau 0)

Bewerkingen op een heap

Zie code heap.h (https://github.com/jelledebock/algo/blob/master/labo2/src/heap.h)

Het sorteeralgoritme

Het principe van heapsort is dat je steeds de wortel uit de boom haalt. Dit is het kleinste/grootste element van de dataset. Vervolgens voeg je laatste bladelement in de wortel in en laat je deze "omlaag **bubbelen**".

```
void heapsort(vector<T> tree){
    while(aantal>0)
    {
        std::swap(tree[aantal-1],v[0]);
        aantal--;
        bubble_down(0);
    }
}
```

Tijdscomplexiteit

- Constructie $O(n)$
- Sorteren $O(n \cdot \log n)$

Merge sort

Principe

Tracht om verschillende gesorteerde deellijsten samen te voegen.

Code

```
void merge_sort(vector<T> &v, int l, int r, vector<v> &hulp){
    //rangschik deel vector v[l]...v[r-1] door samenvoegen
    //gebruik hulp als hulpvector
    if(l < r-1){
        //bereken midden van het interval (l...r)
        int midden = l+(r-1)/2;
        //roep recursief op werkend op 2 deelarrays
        merge_sort(v,l,midden,hulp);
        merge_sort(v,midden,r,hulp);
        merge(v,l,midden,r,hulp);
    }
}

void merge_sort(vector<T> & v){
    //gebruik een vector als "hulpje"
    vector<T> help(v.size());
    merge_sort(v,0,v.size(),help);
}

void merge(vector<T> &v, int l, int m, int r, vector<T>& hulp){
    int w1 = l, w2 = m+1, hw = 0;
    while(w1<=m && w2 <= r){
        hulp[hw++] = (v[w1]<v[w2]?v[w1++]:v[w2++]);
    }
    //plaats de overgebleven elementen bij
    if(w1 > m){
        while(w2<=r){
            hulp[hw++] = v[w2++];
        }
    }
    else{
        while(w1 <= m){
            hulprij[hw++] = v[w1++];
        }
    }
    //Zet in originele tabel
    for(hw=l;hw<=r;hw++){
        v[hw]=hulp[hw];
    }
}
```

Tijdscomplexiteit

- $O(n \log n)$

Quick sort

Principe

- Beste keuze wanneer je intern moet sorteren
- Recursief algoritme
- Een *willekeurig* element wordt uit de gegevens gekozen en dient als **spil (pivot)**
- Aan de hand van deze *pivot* worden de gegens in twee delen opgesplitst
  - De elementen kleiner dan de pivot
  - De elementen groter dan de pivot
- Nadat deze partitionering plaats vond zijn we van de positie van de spil reeds zeker...
- Herhaal dit totdat alle elementen gesorteerd zijn.

Code

```
//Implementatie met als pivot het eerste element (left pivot)
void quick_sort(vector<T> &v, int l, int r){
    //Rangschik deelvector v[l]..[r]
    if(l<r-1){
        T pivot = v[l];
        int i = l, j = r-1;
        while(v[j]>pivot)
            j--;
        while(i<j){
            std::swap(v[i],v[j]);
            i++;

            while(v[i] < pivot)
                i++;
            j--;
            while(v[j] > pivot)
                j--;
        }
    }
    //Roep recursief quicksort op
    quick_sort(v,1,j);
```

```
    quick_sort(v,j+1,r);
}

void quicksort(vector<T> &v){
    quick_sort(v,0,v.size()-1);
}
```

Opmerking: keuze van de spil

Er zijn verschillende mogelijkheden om een spil te kiezen, de keuze van de spil beïnvloedt wel degelijk de uitvoeringstijd van het algoritme. Idealiter zou de spil de rij opdelen in twee delen met gelijke grootte, en dat voor elke stap in de recursie.

- Eerste element in de rij (zie bovenstaande code): werkt prima als de rij willekeurig geordend is. Wanneer ze daarentegen gesorteerd is, dan is deze spilkeuze zeer slecht.
- Het middelste element is dan weer een prima keuze wanneer de rij reeds gesorteerd is.
- **Mediaan van drie** staat gelijk aan een gemiddeld goede spil kiezen. Dat wil zeggen een spil die voor alle mogelijke ordeningen rijen gemiddeld gezien het beste gaat presteren. Een mediaan berekenen kost ook vaak tijd en dit is vaak iets wat een performant algoritme kan missen. Hierdoor wordt een schatting gemaakt door een steekproef uit de dataset te nemen (linkse, rechtse en middelste element) en berekent hiervan de mediaan. ##Tijdscomplexiteit
- **Gemiddeld** $\Theta(n \log n)$
- **Slechtst** $\Theta(n^2)$