

Voorbeeldvragen Algoritmen & Datastructuren I

Sorteeralgoritmen

Titularis: Prof. Dr. Wolfgang De Meuter

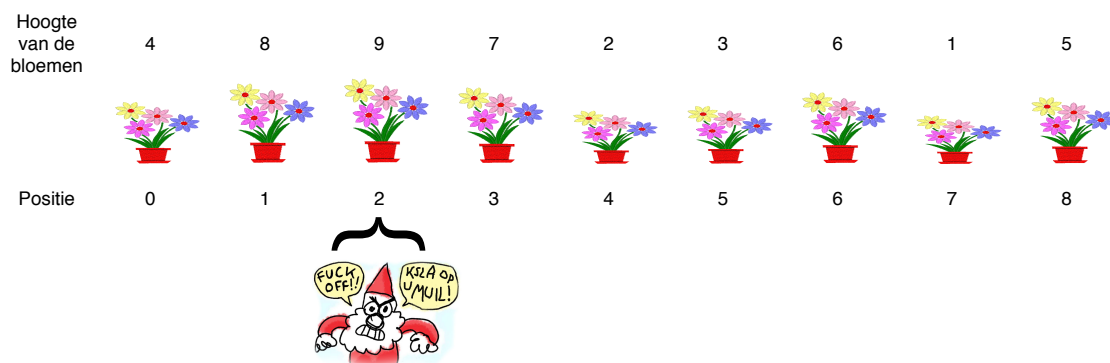
Assistenten: Nathalie Oostvogels, Simon Van de Water

18 december 2015

Vraag 1

Gnome Sort is gebaseerd op de techniek die door kabouters gebruikt wordt om een rij bloempotten te sorteren, bijvoorbeeld volgens de hoogte van de bloemen die zich in de bloempotten bevinden. Stel dat de rij uit n bloempotten bestaat. In dat geval kunnen we elke bloempot een positie toekennen van 0 tot $n - 1$. De kabouter zal telkens de *huidige* bloempot (op positie i) kunnen bekijken, evenals de *vorige* bloempot (op positie $i - 1$) en de *volgende* bloempot (op positie $i + 1$).

De volgende figuur geeft een visuele weergave van een rij van 9 bloempotten. De positie van elke bloempot staat juist onder de bloempot aangeduid, terwijl de hoogte van de bloemen in elke bloempot er boven staat. In de tekening is de *huidige* bloempot de bloempot op positie 2, waaruit volgt dat de *vorige* bloempot zich op positie 1 bevindt, en de *volgende* op positie 3.



De kabouter zal zich, positie per positie, van het begin van de rij (positie 0) naar het einde van de rij (positie $n - 1$) verplaatsen. In iedere stap bekijkt de kabouter de *huidige* bloempot en de *volgende* bloempot. Als deze twee bloempotten in de juiste volgorde staan, verplaatst de kabouter zich naar de *volgende* bloempot (waardoor die de *huidige* wordt). Als de twee bloempotten echter *niet* in de juiste volgorde staan, verwisselt de kabouter de twee bloempotten en verplaatst hij zich naar de *vorige* bloempot (waardoor die de *huidige* wordt).

Er zijn twee randvoorwaarden: als de kabouter zich niet naar een vorige bloempot kan verplaatsen omdat hij aan het begin van de rij staat, zal hij zich naar de volgende bloempot verplaatsen; als de kabouter zich niet naar een volgende bloempot kan verplaatsen omdat hij aan het einde van de rij staat, is de rij bloempotten gesorteerd.

- Gnome Sort vertoont **gelijkenissen** met twee sorteeralgoritmen uit de cursus. Welke, en waarom?
- Implementeer** Gnome Sort voor vectoren van getallen. Schrijf een procedure *wesley* zodat (*wesley* (vector 4 8 9 7 2 3 6 1 5)) de gesorteerde vector #(1 2 3 4 5 6 7 8 9) teruggeeft.
- Bepaal de **tijdscomplexiteit** (in grote O notatie) van Gnome Sort voor het beste, gemiddelde en slechtste geval. Zou de tijdscomplexiteit beïnvloed worden als je in plaats van vectoren enkel gelinkte lijsten of dubbel gelinkte lijsten zou sorteren? Waarom wel/niet?

Vraag 2

Implementeer **selection sort** voor gewone Scheme-lijsten op functionele manier. D.w.z. dat je algoritme een nieuwe gesorteerde lijst moet teruggeven i.p.v. de originele lijst aan te tasten bij het sorteren. Let erop dat je algoritme generisch is (dus dat je alle soorten elementen in de lijst kan sorteren)!

Is je algoritme in-place of niet? Is je algoritme stabiel? Leg uit.

Hint: het is makkelijker om van nul te vertrekken dan om te proberen de selection sort voor vectoren aan te passen.

Vraag 3

De originele plaats van de elementen in een lijst speelt een grote rol in de performantie van **bubble sort**. Grote elementen in het begin van een lijst komen snel op de juiste positie achteraan de lijst terecht. Kleine elementen achteraan een lijst komen echter slechts zeer traag op de juiste positie vooraan de lijst terecht. Deze soorten van elementen worden daarom soms respectievelijk “hazen” en “schildpadden” genoemd. Een mogelijke optimalisatie van bubble sort die het voorkomen van schildpadden reduceert, heet **cocktail sort**. Dit algoritme zal alternerend het grootste element van links naar rechts verschuiven (t.t.z. het standaard gedrag van bubble sort) en het kleinste element van rechts naar links verschuiven. (De naam van het algoritme is geïnspireerd door de op- en neergaande beweging van een cocktailshaker.) Voor een illustratie van de werking van cocktail sort vs. die van bubble sort, beschouw de volgende voorbeelden. Het getal 1 is in het voorbeeld van bubble sort een schildpad.

Bubble sort

Invoer : 2 3 4 5 1
1ste pass : 2 3 4 1 5
2de pass : 2 3 1 4 5
3de pass : 2 1 3 4 5
4de pass : 1 2 3 4 5

Cocktail sort

Invoer : 2 3 4 5 1
1ste pass (links naar rechts) : 2 3 4 1 5
2de pass (rechts naar links) : 1 2 3 4 5

Opgave Implementeer cocktail sort voor Scheme vectoren **of** lijsten. Motiveer je keuze. Zorg dat je procedure generiek is, en dus niet enkel voor getallen kan gebruikt worden. Is je procedure stabiel en/of in place? Leg uit. Bereken de worst case performantiekarakteristiek van je procedure. Is deze beter dan die van bubble sort? Waarom wel/niet?

Vraag 4

Implementeer het *merge sort* algoritme voor Scheme lijsten op een functionele manier. Is je implementatie stabiel? In place? Wat is de performantiekarakteristiek van je implementatie (in grote O notatie)?

Vraag 5

De merge-sort implementatie die we in de cursus bestudeerd hebben is de zogenaamde top-down implementatie van merge-sort: de vector wordt recursief in twee gedeeld en tijdens het backtracken gebeurt de eigenlijke merge operatie.

In deze vraag is het de bedoeling de zogenaamde bottom-up implementatie van merge-sort te implementeren. Deze werkt het eenvoudigst op lijsten (i.p.v. vectoren). De bottom-up implementatie begint door de lijst op te delen in een lijst van lijstjes van lengte 1. Dit is het werk gedaan door de procedure `distribute`:

```
> (distribute (list 5 7 1 9 8 2 4 6))  
((5) (7) (1) (9) (8) (2) (4) (6))
```

Je nieuwe implementatie van merge-sort krijgt zo'n lijst van lijsten als invoer en blijft deze twee-aan-twee mergen tot er een gesorteerde lijst overblijft.

- a) Implementeer deze bottom-up versie.
- b) Is je implementatie stabiel? Argumenteer!
- c) Is je implementatie in-place? Argumenteer!

```

(define (bubble-sort vector <=?)
  (define (bubble-swap vector idx1 idx2)
    (let ((keep (vector-ref vector idx1)))
      (vector-set! vector idx1 (vector-ref vector idx2))
      (vector-set! vector idx2 keep)
      #t))
  (let outer-loop
    ((unsorted-idx (- (vector-length vector) 2)))
    (if (>= unsorted-idx 0)
      (if (let inner-loop
            ((inner-idx 0)
             (has-changed? #f))
            (if (> inner-idx unsorted-idx)
                has-changed?
                (inner-loop (+ inner-idx 1)
                            (if (<=? (vector-ref vector (+ inner-idx 1))
                                     (vector-ref vector inner-idx))
                                (bubble-swap vector inner-idx (+ inner-idx 1))
                                has-changed?))))
          (outer-loop (- unsorted-idx 1))))))

(define (insertion-sort vector <=?)
  (define (>=? x y) (not (<=? x y)))
  (let outer-loop
    ((outer-idx (- (vector-length vector) 2)))
    (let ((current (vector-ref vector outer-idx)))
      (vector-set!
       vector
       (let inner-loop
         ((inner-idx (+ 1 outer-idx)))
         (cond
          ((or (>= inner-idx (vector-length vector))
               (>=? (vector-ref vector inner-idx) current))
           (- inner-idx 1))
          (else
           (vector-set! vector (- inner-idx 1) (vector-ref vector inner-idx))
           (inner-loop (+ inner-idx 1))))))
      current)
      (if (> outer-idx 0)
        (outer-loop (- outer-idx 1))))))

(define (selection-sort vector <=?)
  (define (swap vector i j)
    (let ((keep (vector-ref vector i)))
      (vector-set! vector i (vector-ref vector j))
      (vector-set! vector j keep)))
  (let outer-loop
    ((outer-idx 0))
    (swap vector
     outer-idx
     (let inner-loop
       ((inner-idx (+ outer-idx 1))
        (smallest-idx outer-idx))
       (cond
        ((>= inner-idx (vector-length vector))
         smallest-idx)
        ((<=? (vector-ref vector inner-idx)
              (vector-ref vector smallest-idx))
         (inner-loop (+ inner-idx 1) inner-idx))
        (else
         (inner-loop (+ inner-idx 1) smallest-idx))))))
    (if (< outer-idx (- (vector-length vector) 1))
      (outer-loop (+ outer-idx 1))))))

(define (heapsort vector <=?)
  (define >=? (lambda (x y) (not (<=? x y))))
  (define heap (from-scheme-vector vector >=?))
  (define (extract idx)
    (vector-set! vector idx (delete! heap))
    (if (> idx 0)
      (extract (- idx 1))))
  (extract (- (vector-length vector) 1)))

```

```

(define (quicksort vector <=?)
  (define (swap i j)
    (let ((keep (vector-ref vector i)))
      (vector-set! vector i (vector-ref vector j))
      (vector-set! vector j keep)))
  (define (shift-to-right i x)
    (if (<=? (vector-ref vector i) x)
      (shift-to-right (+ i 1) x)
      i))
  (define (shift-to-left j x)
    (if (<=? x (vector-ref vector j))
      (shift-to-left (- j 1) x)
      j))
  (define (partition pivot i j)
    (let ((shifted-i (shift-to-right i pivot))
          (shifted-j (shift-to-left j pivot)))
      (cond ((< shifted-i shifted-j)
             (swap shifted-i shifted-j)
             (partition pivot shifted-i (- shifted-j 1)))
            (else
             shifted-j))))
  (define (quicksort-main l r)
    (if (< l r)
      (begin
       (if (<=? (vector-ref vector r)
               (vector-ref vector l))
         (swap l r))
       (let ((m (partition (vector-ref vector l) (+ l 1) (- r 1))))
         (swap l m)
         (quicksort-main l (- m 1))
         (quicksort-main (+ m 1) r))))))
  (quicksort-main 0 (- (vector-length vector) 1)))

(define (mergesort vector <=?)
  (define (merge vector p q r)
    (let ((working-vector (make-vector (+ (- r p) 1))))
      (define (copy-back a b)
        (vector-set! vector b (vector-ref working-vector a))
        (if (< a (- (vector-length working-vector) 1))
            (copy-back (+ a 1) (+ b 1))))
      (define (flush-remaining k i until)
        (vector-set! working-vector k (vector-ref vector i))
        (if (< i until)
            (flush-remaining (+ k 1) (+ i 1) until)
            (copy-back 0 p))))
      (define (merge-iter k i j)
        (cond ((and (<= i q) (<= j r))
              (let ((low1 (vector-ref vector i))
                    (low2 (vector-ref vector j)))
                (if (<=? low1 low2)
                    (begin
                     (vector-set! working-vector k low1)
                     (merge-iter (+ k 1) (+ i 1) j))
                    (begin
                     (vector-set! working-vector k low2)
                     (merge-iter (+ k 1) i (+ j 1))))))
              ((<= i q)
               (flush-remaining k i q))
              (else
               (flush-remaining k j r))))
      (merge-iter 0 p (+ q 1))))
  (define (merge-sort-rec vector p r)
    (if (< p r)
      (let ((q (div (+ r p) 2)))
        (merge-sort-rec vector p q)
        (merge-sort-rec vector (+ q 1) r)
        (merge vector p q r)))
      (merge-sort-rec vector 0 (- (vector-length vector) 1)
        vector)))

```