# Oplossingen zeverexamenvragen

Mats Esseldeurs

June 2019

## 1 Geef het algoritme van Prim om een minimaal opspannende boom van een grafe te berekenen, en bespreek en verklaar de tijdscomplexiteit.

In het algoritme van Prim om een MST te vinden is het de bedoeling om vanaf een gegeven knoop uit te breiden tot een MST gevonden is.

- Een knoop kan worden toegevoegd als hij rechtstreeks verbonden kan worden met een eerder verbonden knoop. De boom zal dus in elke stap volledig verbonden zijn.
- Om hiervoor de juiste edge te kiezen zullen de edges verbonden met knopen die al verbonden zijn op een queue gestoken worden gesorteerd op hun 'lengte'.
- Nu zijn er twee verschillende implementaties mogelijk, lazy en eager Prim.
  - In lazy prim zal er gewoon blijven edges op de queue gegooid worden dus ook de edges die mogelijks een lus zullen vormen.
    - Dit zorgt ervoor dat bij deze implementatie er extra ruimte nodig is volgens $\sim E$
    - Ook zorgt dit voor een tijdscomplexiteit komende van de queue volgens $\sim E \log_2(E)$.
  - In eager prim zal elke iteratie kijken welke edges uit de queue mogen verwijderd worden. Hierdoor zullen er minder edges op de queue komen te staan
    - Dit zorgt ervoor dat bij deze implementatie er slechts extra ruimte nodig is volgens $\sim V$
    - Ook zorgt dit voor een tijdscomplexiteit komende van de queue volgens $\sim E \log_2(V)$.

## 2 Bespreek het CountingSort algoritme (Key-Indexed Sorting). Geef een intuïtieve beschrijving, pseudocode, en een analyse van de tijdscomplexiteit.

Beschrijving: CountingSort bestaat erin de lijst af ge gaan en het aantal elementen van een bepaalde soort te tellen. Bijvoorbeeld een lijst met op elke plaats een letter, dan zullen er veel elementen zijn die hetzelfde zijn. CountingSort zal dan een lijst maken die elk mogelijk karakter voorstelt en door de lijst gaan en tellen hoeveel dit karakter voorkomt. De lijst zal vervolgens gereconstrueerd kunnen worden aan de hand van de tellijst.

Pseudocode:

```
CountingSort(list) {
N = length of list
aux = list as lang as the original list
count = list with size the emount of characters
for (i = 0 .. N-1) count[list[i].index()+1]++
for (r = 0 .. R-1) count[r+1] = count[r+1] + count[r]
for (i = 0 .. N-1) aux[count[a[i]]++] = a[i]
for (i = 0 .. N-1) a[i] = aux[i]
}
```

Complexiteit: In de Eerste for-lus worden 2N array accesses gebruikt, in de tweede 3R, de derde 4N en in de vierde 2N. Samen geeft dit $\sim 8N + 3R$

## 3 De algoritmen van Prim en Kruskal maken beiden gebruik van een priority queue om het algoritme te implementeren. Bespreek het gebruik van de queue in beide aanpakken. Op welke manier bepaalt de grootte van de queue de tijdscomplexiteit van beiden?

Prim: In Prims algoritme zullen edges gedeeltelijk op de queue gegooid worden, gesorteerd op de weight van de edge. Ze worden erop gegooid wanneer ze kans maken om de volgende gekozen edge te zijn. Dit kan vanaf een van de 2 kanten van de edge verbonden is met een knoop die al deel uitmaakt van de voorlopig gebouwde MST. Nu zijn er 2 verschillende implementaties van het prim algoritme:

- Lazy Prim: In deze vorm van prim zal het hierbij gelaten worden, er wordt alleen getest of de edge die uit de queue komt wel een deel mag uitmaken van de MST, indien deze edge een lus zou vormen mag hij niet toegevoegd worden. Er zullen dus maximum E (= aantal edges) elementen op de queue zitten.
- Eager Prim: In deze vorm van prim zal er elke keer dat er een edge uit gehaalt wordt (indien het een edge is dat aan de MST mag toegevoegd worden) endus extra edges worden toegevoegd aan de queue ook kijken of er edges uit de queue mogen omdat deze een lus zouden vormen endus toch niet bij de MST mogen horen. Dit bespaart ruimte en zal de maximale lengte van de queue verkleinen tot maximum V (= aantal knopen).

Kruskal: In Kruskals algoritme zullen rechtstreeks alle edges op de queue gegooid, gesorteerd op de weight van de edge. Om daarna elke edge af te gaan en te kijken of deze bij de MST mag horen. Dit zorgt ervoor dat op het begin er E (= aantal edges) elementen op de queue zitten.

Complexiteit: In beide algoritmes zal de complexiteit bepaalt worden door het aantal elementen in de queue. Een element toevoegen of verwijderen uit een queue kost $\sim \log_2(N)$ tijd met N het aantal elementen op de queue. Dit zal E keer gebeuren in beide gevallen met voor elk algoritme zijn eigen maximum N. Dit geeft voor Eager Prim $\sim E \log_2(V)$ door zijn maximum van V elementen en voor lazy Prim en Kruskal $\sim E \log_2(E)$ met hun maximum van E elementen.

## 4 Bespreek verschillende strategieën om collisions in hash-tabellen op te lossen. Geef duidelijk voor- en nadelen aan, en illustreer met relevante voorbeelden indien nodig.

- Seperate chaining: In seperate chaining zullen collisions in hash-tabellen opgelost kunnen worden door de posities als een linked list te bekijken. In dit geval kunnen er meerdere objecten op dezelfde plaats staan.
  - Insert: Bij de insert-operatie zal het object op de juiste positie worden toegevoegd. Indien daar al een element staat zal een link worden gemaakt van het laatste element op deze positie naar het object dat wordt toegevoegd.
  - Search: Bij de search-operatie zal rechtstreeks naar de juiste positie in de hash-tabel gekeken kunnen worden en hier heel de linked list afgaan. De zoektijd is dus nagenoeg constant.
  - Delete: Bij de delete-operatie zal het object eerst gezocht moeten worden, dit kan in constante tijd. Nu het element gevonden is kan het uit de linked list gehaald worden en zal de link die normaal naar het object wees (het vorige object) nu naar het volgende object in de linked list verwezen worden.
  - Voordelen:
    - Simpel te implementeren
    - Hash-tabel geraakt nooit vol
  - Nadelen:
    - Het kost tijd om de linken juist te behandelen
    - Somige posities worden helemaal niet gebruikt
- Lineair probing: In lineair probing zullen collisions in hash-tabellen opgelost worden door als de positie bezet is naar de volgende positie in de array te gaan en die proberen opvullen. In dit geval kan er dus maar 1 element per positie bestaan.
  - Insert: Bij de insert-operatie zal het object op de juiste positie worden toegevoegd. Indien daar al een element staat zal naar de volgende positie in de array gekeken worden en daar proberen toe te voegen.

- Search: Bij de search-operatie zal rechtstreeks naar de juiste positie in de hash-tabel gekeken kunnen worden. Indien dit niet het juiste element is wordt naar het volgende element gekeken worden tot een lege positie in de array gevonden wordt. De zoektijd is dus nagenoeg constant.
- Delete: Bij de delete-operatie zal het object eerst gezocht moeten worden, dit kan in constante tijd. Nu het element gevonden is zal er een soort grafsteen geplaatst worden zodat een search-operatie weet dat hij niet moet stoppen met zoeken. Deze grafstenen zullen echter wel overschreven mogen worden door een insert-operatie
- Voordelen:
  * minder geheugengebruik door geen linken te moeten maken
- Nadelen:
  * Moeilijker te implementeren
  * Last van ophopingen van punten

# 5 Leg het Boyer-Moore algoritme voor substring matching uit.

In het Boyer-Moore algoritme voor substring matching wordt er getracht na te gaan of een substring in een gegeven string zit. Dit zal gebeuren door vanachter aan de substring te beginnen. Nu zijn er 3 opties mogelijk:

- Het element matched: Indien het element van de substring matched met het element in de string dan zal er gekeken worden naar het element links van het huidige element. Indien dit het eerste element is dan is de hele string overlappend en is er een match.
- Het element is een element dat niet voorkomt in de substring: Indien het element van de string dat bekeken wordt niet overeenkomt en sterker nog niet eens in de substring zit dan kan er zeker gezegt worden dat dit element nooit tot de substring zal kunnen behoren en kan er geskipt worden voor heel de string. De string wordt dus achter dit element geplaatst en er wordt opnieuw op het einde van de substring gezocht.
- Het element matched niet maar komt wel voor in de substring: Indien het element niet matched maar wel voorkomt in de substring kan niet zomaar heel de string opgeschoven worden naar voor dit karakter. Nu zal naar de meest rechtse keer dat dit karakter voorkomt in de substring moeten gekeken worden (voor lepel is de e het meest rechts op positie 3 indien vanaf 0 getelt wordt). Nu zijn er opnieuw 2 opties:
  - Dit element bevind zich rechts van het huidig gecheckte element: Indien dit element zich rechts bevind dan kan mits een kleine opschuiving niet opnieuw een mogelijke configuratie gevonden worden en zal dus gedaan worden alsof het element helemaal niet voorkwam in de substring (zie puntje 2)
  - Dit element bevind zich links van het huidig gecheckte element: Indien dit element zich links bevind kan er wel een mogelijke configuratie ontstaan door op te schuiven. De string wordt dus opgeschoven tot dit meest rechtse karakter overeenkomt met het huidige gecheckte karakter in de string en wordt opnieuw helemaal rechts gekeken of het een match is.

# 6 Leg uit: Huffman codering.

In het Hoffman algoritme wordt gezocht naar een efficiente manier om aan datacompressie te doen. gegeven een string maakt dit algoritme de optimale codering. Dit wordt in een aantal stappen gedaan:

Stap 1: Tel elk karakter in de string.

Stap 2: Neem de 2 karakters die het minste voorkomen, deze vormen samen een tak van een boom, del de frequenties op en beschouw de boom als een nieuw karakter met frequentie de som van de 2 frequenties.

Stap 3: Herhaal tot elk element gebruikt is. Dit doen maakt het algoritme greedy.

Stap 4: Elke splitsing naar links of naar rechts in de boom staat voor een 0 of een 1 in de codering. Elk element kan dus met een combinatie van 0 en 1 tjes voorgesteld worden. Deze codering is ook prefix-vrij.

Dat dit een optimale code geeft kan ook bewezen worden:

Lemma 1: Elke optimale code bestaat uit een complete boom.
Indien de optimale boom niet uit een complete boom bestaat zou het mogelijk zijn om de boom wel compleet te maken door het element dat alleen staat omhoog te duwen. Dit resulteert in een optimalere code endus is het lemma bewezen.

Lemma 2: Er bestaat een optimale boom waarbij de elementen met de laagste frequentie siblings van elkaar zijn. Indien dit niet het geval was zijn er twee opties: ze liggen op dezelfde diepte in de boom of een van de twee ligt hoger dan de andere. Indien ze op dezelfde hoogte liggen kan er geschoven worden met elementen zodanig dat deze twee elementen wel siblings zijn. Indien een van de twee op een andere hoogte ligt zal de verwisseling tussen de sibling van het laagste element en het hoger gelegen element een optimalere code geven. Dit is dus geen optie.

Inductie: De rest van het bewijs is gegeven per inductie:

Basisstap: Voor 2 karakters is de optimale boom triviaal, een van de twee is 0 en de andere 1. Dit is ook de boom gevonden door het Huffman algoritme.

Inductiestap: GEEN ZIN IN

# 7 Wat betekent het als een sorteeralgoritme "stabiel" (stable) wordt genoemd? Welk sorteeralgoritme maakt expliciet gebruik van stabiliteit om correct te kunnen werken? Verklaar.

- Een stabiel algoritme een algoritme dat structuur die in de data zit behoud. Een voorbeeld hiervan is indien gekeken wordt naar een lijst van personen die al alfabetisch is gesorteerd en men gaat deze lijst opnieuw sorteren met een stabiel algoritme op leeftijd. Dan zal binnen elke leeftijdsklasse de lijst steeds alfabetisch gesorteerd zijn.
- Stabiele algoritmes: LSD, InsertionSort, MergeSort, CountingSort ...
- Niet stabiele algoritmes: SelectionSort, QuickSort ...
- Een algoritme dat expliciet gebruik maakt van stabiliteit van algoritmes is LSD. Dit algoritme zal gebruik maken van CountingSort dat stabiel is. Een lijst zal gesorteerd worden door te beginnen bij het laatste karakter en dit te sorteren. Hierna het voorlaatste enzovoort. Stabiliteit komt hier in plaats doordat binnen dezelfde karakters ervanuit kan gegaanworden dat de volgende karakters al gesorteerd zijn.
- GEEF EEN VOORBEELD

# 8 Bespreek lineair probing om collisions in een hash-tabel op te lossen.

In lineair probing zullen collisions in hash-tabellen opgelost worden door als de positie bezet is naar de volgende positie in de array te gaan en die proberen opvullen. In dit geval kan er dus maar 1 element per positie bestaan.

- Insert: Bij de insert-operatie zal het object op de juiste positie worden toegevoegd. Indien daar al een element staat zal naar de volgende positie in de array gekeken worden en daar proberen toe te voegen.
- Search: Bij de search-operatie zal rechtstreeks naar de juiste positie in de hash-tabel gekeken kunnen worden. Indien dit niet het juiste element is wordt naar het volgende element gekeken worden tot een lege positie in de array gevonden wordt. De zoektijd is dus nagenoeg constant.
- Delete: Bij de delete-operatie zal het object eerst gezocht moeten worden, dit kan in constante tijd. Nu het element gevonden is zal er een soort grafsteen geplaatst worden zodat een search-operatie weet dat hij niet moet stoppen met zoeken. Deze grafstenen zullen echter wel overschreven mogen worden door een insert-operatie
- Voordelen:
  - minder geheugengebruik door geen linken te moeten maken
- Nadelen:
  - Moeilijker te implementeren
  - Last van ophopingen van punten
- Complexiteit: Voor de search operatie zal de complexiteit gegeven worden door $\sim \frac{1}{2}(1 + \frac{1}{1-\alpha})$ voor een operatie met een hit en $\sim \frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$ voor een operatie zonder hit. Dit komt door het feit dat de kans op een collision $\frac{1}{1-\alpha}$ is met $\alpha = \frac{M}{N}$ met M het aantal plaatsen in de tabel en N het aantal elementen in de tabel.

## 9 Bespreek het Least-Significant-Digit (LSD) sorteeralgoritme. Geef een intuïtieve beschrijving, pseudocode, en een analyse van de tijdscomplexiteit.

Beschrijving: LSD is een algoritme dat gebaseerd is op de eigenschap stabiliteit van andere algoritmes. In de meeste implementaties zal gebruik gemaakt worden van CountingSort omdat dit algoritme deze eigenschap heeft.

- Een stabiel algoritme betekend dat als een lijst een bepaalde eigenschap heeft bv al alfabetisch is gesorteerd dat indien men de lijst op een andere eigenschap zal sorteren bv lengte van een woord dat binnen dezelfde lengtes de lijst nogsteeds alfabetisch gesorteerd is.
- De exacte uitwerking van LSD bestaat erin dat men eerst op de minst belangrijke digit zal sorteren (meestal het laatste karakter van een lijst strings die even lang zijn). Vervolgen zal het voorlaatste karakter gesorteerd worden en zo verder tot en met het eerste karakter. Nu is de lijst op elk karakter alfabetisch gesorteerd met het eerste karakter het meest prioritair.

Pseudocode: LSD(list){

W = length of elements in list

for (i = W-1 .. 0) CountingSort(list, i) }

Complexiteit: Dit algoritme zal W keer CountingSort oproepen. Dit algoritme heeft een tijdscomplexiteit van $\sim 8N + 3R$ met N het aantal elementen in de lijst en R het aantal mogelijke karakters. Dit resulteert dus rechtstreeks in een complexiteit van $\sim 8WN + 3WR$.

## 10 Bespreek het algoritme van Kruskal om een minimaal opspannende boom van een grafe te berekenen, en verklaar de tijdscomplexiteit.

- In kruskals algoritme om de minimale opgespanne boom van een grafe te berekenen zal er een priority queue gebruikt worden gevuld met edges die gesorteerd zijn op hun weight om gebruikt te worden.
  - Er zal dus een queue ontstaan met een maximum van E (= aantal edges in de grafe) elementen.
  - Uit deze queue zullen stuk voor stuk elementen uitgehaalt worden en nagekeken worden of ztabtoe aan de MST mogen voldoen. Dit is niet het geval indien er een lus zou ontstaan in de voorlopig opgebouwde boom.
- Dit geeft dus aanleiding tot het regelmatig gebruiken van de insert en extract minimum operaties van de queue. Deze hebben beide complexiteit van $\sim \log_2(N)$ met N het aantal elementen op de queue. Dit zal voor E elementen gedaan worden met een maximum van E elementen op de queue resulterend in een complexiteit van $\sim E \log_2(E)$.

## 11 Bespreek de insert en delete-maximum operaties in een heap-structuur.

In een heap-structuur is een boom waarvan in elk punt ervan uit gegaan worden dat beide kinderen kleiner zijn dan het huidige punt. Een heap is ook een complete boom, dit wilt zeggen dat enkel de onderste diepte niet volledig opgevuld is en elk element op de laagste diepte helemaal naar links is geduwd, VOORBEELD. Met deze gegevens worden volgende operaties besproken:

Insert: In deze operatie komt er een nieuw element binnen maar de boom moet zijn structuur behouden. Dit zal gedaan worden door het element helemaal vanachter in de boom te steken en het element dan te laten 'zwemmen'.

- Zwemmen betekend dat naar dit element gekeken wordt en naar zijn ouder en indien de ouder groter is wordt hij verwisselt en zal men naar de ouder daarvan gaan vergelijken tot het toegevoegde element op de juiste plaats bevind.
- De complexiteit van deze operatie is $\sim \log_2(N)$ aangezien er nooit meer compares zullen zijn dan helemaal naarboven te zwemmen. De diepte van de boom is ook gegeven door deze $\log_2(N)$

delete: In deze operatie zal het bovenste element verwijderd worden aangezien dit het grootste element is. Dit kan gezegd worden omdat elk element onder hem zeker kleiner is. Maar nu is er een lege plaats in de boom. Het laatste element van de boom zal dus verplaatst worden om deze op te vullen alhoewel deze hier niet perse thuis hoort. er zal dus een 'sink' operatie uitgevoerd worden.

- De sink operatie zal dit element laten zinken. Hij zal dit doen door naar zijn kinderen te kijken en te zien welk element het grootste is van de twee en indien deze groter is dan het te zinken element worden ze verwisselt. Nu kan het zijn dat zijn kinderen nogsteeds groter zijn dus de swim operatie herhaalt zichzelf tabtotot het element op een juiste plaats zich bevind.
- De complexiteit van deze operatie is $\sim 2\log_2(N)$ aangezien er voor elke sink operatie 2 kinderen zijn die vergeleken moeten worden en een met het element zelf. De $\log_2(N)$ komt opneuw van de diepte van de boom.

## 12 Omschrijf het Longest Common Subsequence probleem (langste gemeenschappelijke subsequentie van 2 strings) zoals gezien in de les (ter herinnering: we hebben dit probleem toegepast op DNA-strings). Geef tevens een oplossing die gebruikt maakt van dynamisch programmeren om LCS op te lossen.

Dit probleem is een voorbeeld van dynamisch programmeren. Het probleem zal recursief opgelost worden maar er zal voor gezorgd worden dat er geen dubbel werk gedaan wordt. Er zal een soort lookup table gemaakt worden waarin eerder gemaakte bewerkingen in opgeslagen worden. Dit is een matrix van grootte MxN met M de lengte van de ene string en N de lengte van de andere string. Deze matrix zal gevuld worden met volgende rekenregels:

$$L[i][j] = L[i-1][j-1] + 1 \; als \; X[i] == Y[j] \tag{1}$$

$$L[i][j] = max\{L[i-1][j], L[i][j-1]\} \; als \; X[i] != Y[j] \tag{2}$$

- Rekenregel 1 is afkomstig van het feit dat indien het laatste element van X overeen komt dat van Y, dit zeker behoort tot de LCS endus het aantal in de LCS van de strings waar deze elementen geen deel van uit maken plus 1.
- Rekenregel 2 is afkomstig van het feit dat indien de laatste elementen van X niet overeen komt met dat van Y, een van deze elementen zeker niet tot de LCS behoort, er zal dus gekeken worden naar het maximum van wanneer van X een element afgetrokken wordt en wanneer van Y een element afgetrokken wordt.
- Dit algoritme zal op deze manier een matrix vullen om op positie L[M][N] het antwoord op het probleem te vinden. Deze matrix opvullen zal dus overeen komen met een complexiteit van $\sim M \cdot N$.

## 13 Is de volgende uitspraak waar of niet waar? "Er bestaat een algoritme, gebaseerd op onderlinge vergelijkingen van elementen, dat 5 willekeurige elementen steeds kan sorteren door gebruik te maken van maximaal 6 vergelijkingen." Verklaar tevens je antwoord.

- Er kan voor een algoritme dat gebaseerd is op onderlinge vergelijkingen van elementen bewijzen dat het niet sneller zal kunnen opgelost worden dan $\sim \log_2(N!)$. Voor grote N zal deze formule door de Stirling-benadering geschreven kunnen worden als $\sim N\log_2(N)$ maar voor dit geval wordt niet met grote N gewerkt.
- Voor dit geval geeft deze formule ingevuld $\log_2(N!) = \log_2(6!) = \log_2(120) \in [6, 7]$ endus kan besloten worden dat er minstens 7 vergelijkingen nodig zijn. De uitspraak is dus fout.

## 14 Becommentarieer de volgende uitspraak: "HeapSort kan gebruikt worden i.p.v. Counting Sort (key-indexed sorting), als de basis voor Least Significant Digit (LSD) sortering."

- LSD is een algoritme dat gebaseerd is op de eigenschap stabiliteit van het gebruikte onderliggende algoritme. Stabiele algoritmes zijn algoritmes die vorige structuren behouden. Een voorbeeld hiervan is neem een lijst waar gegevens van mensen in zitten, deze lijst is alfabetisch gesorteerd. Indien deze lijst gesorteerd zal worden met een stabiel algortime op bijvoorbeeld leeftijd zal binnen elke leeftijd de lijst nogsteeds alfabetisch gesorteerd zijn. HeapSort is niet stabiel endus kan dit niet toegepast worden op LSD.

- Er is wel de optie om HeapSort wel stabiel te maken door de oorspronkelijke positie van elementen in rekening te brengen bij het sorteren maar dit kost extra tijd en is dus niet gewenst. HeapSort heeft oorspronkelijk een complexiteit van $\sim Nlog_2(N)$ terwijl CountingSort een complexiteit heeft van $\sim 8N + 3R$ wat sneller is dan linearitmische tijd. Het heeft dus geen voordeel om HeapSort te gebruiken.

## 15 Stel dat je een groot aantal gegevens gegevens over personenwagens wil bijhouden in een database, waarbij je wagens wil opzoeken a.d.h.v. hun nummerplaat. We willen enkel gegevens opzoeken, dus niets toevoegen of verwijderen. Bovendien past de volledige database in het geheugen van de computer. Voor welke datastructuur uit onderstaande mogelijkheden zou je opteren, en waarom? Van welke andere factoren hangt de keuze eventueel af? Keuze uit: Gesorteerde array – Gelinkte lijst – Binaire zoekboom – Queue – Hashtabel

- Een gesorteerde array en een gelinkte lijst hebben een lineaire tijd om een zoek-operatie toe te passen, deze structuur is dus niet gewenst aangezien dit traag zal zijn.
- Een binaire zoekboom en een Queue hebben logaritmische tijd. Dit is sneller als lineair maar nog niet constant. De volledige database past in het geheugen van de computer dus indien dit ook het maximum is zullen deze structuren wel gehanteerd worden.
- Een HashTabel heeft constante tijd om een zoek-operatie te doen. Dit is ideaal maar vraagt wel extra geheugen. Zowel bij Lineair pobing als Seperate chaining zijn er lege plaatsen die geheugen innemen. Indien er dus geen extra geheugen over is zal eerder voor een binaire zoekboom of een queue gekozen worden.

## 16 Bespreek de partitionering van Lomuto voor het partitioneren van een array bij het QuickSort algoritme. Geef een intuïtieve beschrijving, pseudocode, bespreek eventuele voor- en nadelen, alsook de tijdscomplexiteit.

Beschrijving: In Lomutos partitionering voor QuickSort zal er in de array die met de povot onderzocht wordt van links naar rechts gescand worden. Terwijl dit gebeurt worden alle elementen die kleiner zijn als de pivot links gezet, er wordt dus ook bijgehouden hoeveel elementen er al links staan. Verder zal er naar rechts gegaan worden en telkens als het gescande element kleiner is dan de pivot wordt het verwisselt met het element 1 rechts van de lijst elementen dat kleiner is. Als dit gebeurt is wordt de pivot er midden in geplaatst.

Pseudocode:

```
Lomuto-Partition(A, p, r) {
x = A[r]
i = p - 1
for j = p to r - 1
   if A[j] ≤ x
      i = i + 1
      swap( A[i], A[j] )
swap( A[i +1], A[r] )
return i + 1
}
```

Complexiteit: In dit algoritme wordt er vanaf p tot r - 1 afgegaan en telkens 1 vergelijking gemaakt. Dit resulteert dus voor een rij van N elementen in een complexiteit van $\sim N - 1$.