GEGEVENSSTRUCTUREN EN ALGORITMES
EXAM 2022-2023
Prof P. Dutré

Liam Luys

KU Leuven

Veel succes met het examen!
Ik heb geprobeerd een samenvatting te maken die alle info bevat uit de cursustekst/slides.
Geen garantie dat alles er in staat en/of juist is, doe zeker de boeken ook open.
Foutje gevonden? Laat het zeker weten, dan pas ik het aan!

Les 1 - Inleiding

vrijdag 17 februari 2023 10:40

Liam Luys – r0934449 - 1e BA Informatica
Gegevensstructuren en Algoritmes

KU LEUVEN

1. Fundamenten Algoritmes

1.1 Basis van Algoritmes

Wat is een algoritme?
- Een reeks instructies
- Steeds hetzelfde resultaat voor dezelfde input?
- Procedurale Kennis

Eigenschappen Algoritmes
- Betrouwbaarheid / Correctheid
- Eindigheid / Snelheid
- Efficiënt in tijd en operaties
- Leesbaar

Declaratieve kennis
- Waarneembare feiten
- Weten wat iets is

Procedurale kennis
- Hoe kan je bepaalde kennis uitrekenen?
- Waten hoe en wanneer feiten, definities en concepten worden toegepast

Equivalente Problemen

Convex Omhullende
- Voorbeeld van algoritme
- Eerst onderste punt zoeken, dan alle punten sorteren volgens hoek t.o.v. onderste punt
- Elk punt aflopen en bypassen als ze bepaalde hoek maken

- Als we optimaal algoritme vinden voor het sorteren van de hoeken, hebben we een optimaal algoritme voor de convex omhullende

Equivalente problemen
- Kunnen door eenzelfde algoritme opgelost worden
- Oplossing voor de ene is oplossing voor de andere
- Hebben dezelfde efficiëntie, ondergrens, …
- vb: sorteren en convex omhullende

Studie Algoritmes

Waarom algoritmes bestuderen?
- Intellectuele stimulatie
- Om een betere ontwikkelaar/programmeur te worden
- Om de geheimen van het universum te ontdekken
- Slimme oplossingen te vinden
- Verband tussen algoritme en grootte van input
- Problemen tot elkaar reduceren en zelfde algoritme gebruiken

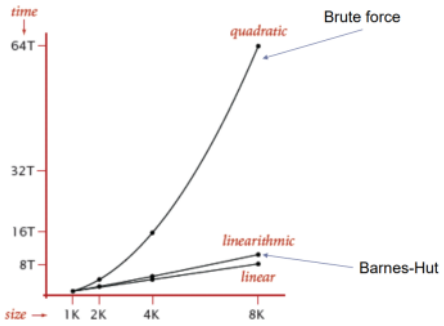
Beter algoritme is efficiënter en slimmere keuze dan een snellere computer!

1.4 Analyse van Algoritmes

Waarom Algoritmes Analyseren?

- Voorspellen van gedrag, tijd, …
- Vergelijken van algoritmes
- Garanties kunnen geven "mag max zoveel tijd duren"
- Theoretische kennis opbouwen van hoe algoritmes werken

N-Body Probleem



Hypothese van tijdsduur

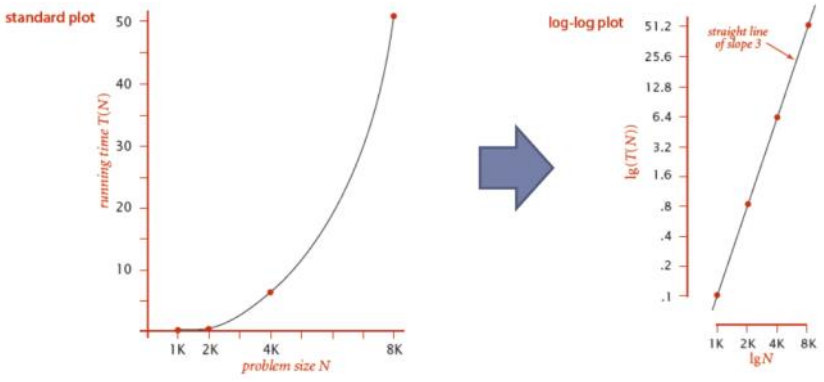
Hoe hypotheses maken over tijd van algoritme?

- Grafiek maken en functie opstellen adv observaties
- Verdubbelingsexperiment
- Wiskundig model: Operaties tellen

Hypothese maken: Grafiek methode

Grafiek-methode

- De gegeven waarden plotten in een grafiek
- De helling van de log-log plot is de exponent (1 bij lineair, 2 bij kwadratisch, 3, …) bij de invoergrootte N
- Formule opstellen a.d.h.v. gevonden waardes
- Hypothese doen over grotere waardes



Hypothese maken: Verdubbelingsmethode

Verdubbeling-methode

- Ga er van uit dat je een afhankelijkheid van a * N^b
- b berekenen door verhouding te nemen van experimenten
    - $T(N) = a * N^b$

    - $T(2N) = a * (2N)^b = a * 2^b * N^b$

    - $T(N) / T(2N) = 2^b$

- b kan je berekenen door binair logaritme te nemen
- Formule opstellen voor tijdsduur van algoritme a.d.h.v. invoergrootte
- a berekenen/schatten door b in te vullen in metingen

Afhankelijkheid van systeem

Systeem-onafhankelijke factoren
- Algoritme
- Input data

- ***Geeft aan hoe de tijd groeit met de grootte van de input***

Systeem-afhankelijke factoren
- Hardware: CPU, geheugen, cache, …
- Software: Compiler, interpreter, garbage collector, …
- Systeem: Besturingssysteem, netwerk, achtergrondprocessen, …

- ***Geeft de absolute snelheid van het algoritme op dat specifieke toestel***

Parameters verdubbelings-experiment *($a * N^b$)*
- Parameter b
    - Systeem-onafhankelijke parameter
    - Geeft aan hoe de tijd groeit, verhouding

- Parameter a
    - Systeem-afhankelijke parameter
    - Geeft de absolute tijd weer op een specifieke computer

Hypothese maken: Wiskundig model

Hypothese maken via wiskundig model
- Tellen van operaties die worden uitgevoerd in algoritme
- De kosten van alle operaties afzonderlijk optellen

Vergemakkelijking bij hypothese
- Enkel kijken naar de significante operaties
- Enkel kijken naar grootte waardes van N, grootte invoer van data
    - Asymptotisch gedrag onderzoeken, niet op kleinere schaal
    - Tilde notatie

Enkel tellen van significante operaties
- Anders wordt het veel te complex
- Is hetgeen wat 'het echte werk' doet in het algoritme

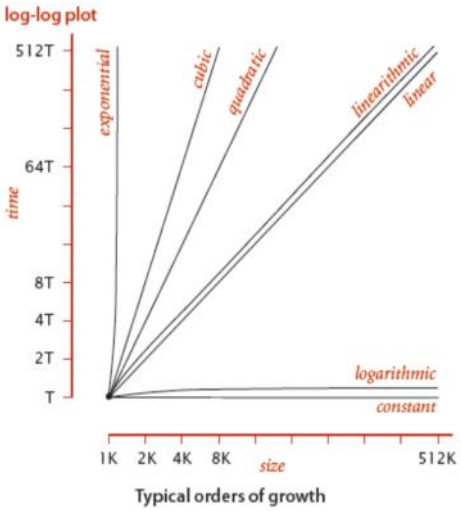
- Kijken naar de body van de lussen
- Alan Turing

Enkel grootte invoergroottes gebruiken

- Gebruik van asymptotisch gedrag
- Onderzoek met tilde-notatie
    - Specifieker dan grote O-notatie
    - F = ~G als G/F gaat naar 1

NOOIT GROTE O GEBRUIKEN OP EXAMEN

growth rate	name	typical code framework	description	example	T(2N) / T(N)
1	constant	`a = b + c;`	statement	add two numbers	1
log N	logarithmic	`while (N > 1) { N = N / 2; ... }`	divide in half	binary search	~ 1
N	linear	`for (int i = 0; i < N; i++) { ... }`	loop	find the maximum	2
N log N	linearithmic	[see mergesort lecture]	divide and conquer	mergesort	~ 2
N^2	quadratic	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }`	double loop	check all pairs	4
N^3	cubic	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }`	triple loop	check all triples	8
2^N	exponential	[see combinatorial search lecture]	exhaustive search	check all subsets	T(N)



Groeipatronen van algoritmes

- Constant
    - Tijdsduur hangt niet af van invoergrootte

- Logaritmisch
    - Vaak algoritmes waarin recursie optreedt: binary search
    - Is zeer goed! Ergens een log in krijgen is goed voor tijd!
    - Tijdsduur volgens log(N)

- Lineair
    - Tijdsduur volgens N

- Linearitmisch
    - Tijdsduur volgens N*log(N)

- Kwadratisch
    - vb 2 geneste for-loops
    - Tijdsduur volgens N^2

- Kubisch
    - vb 3 geneste for-loops
    - Tijdsduur volgens N^3

- Exponentieel
    - Tijdsduur volgens b^N met b > 1

Logaritmisch Algoritme

- Traag stijgend logaritme
- Willen we als informatici, logaritmisch algoritme willen we gebruiken!
- Komt vaak uit recursief algoritme

Tijd Notaties

Grote O-Notatie

- Duid enkel een bovengrens aan

  f(x) element van O(g(x))
  - $\forall x >= x_0: \exists c: f(x) \underline{<=} c * g(x)$

  vb: $50x$ is element van $O(x^2)$, ook $10x^2$ of $1000x^2$ behoren tot $O(x^2)$
  - $O(x^2)$ is de bovengrens!

Grote Omega-Notatie

- Duid enkel een ondergrens aan

  f(x) element van Ω(x)
  - $\forall x >= x_0: \exists c: f(x) \underline{>=} c * g(x)$

  vb:

Grote Teta-Notatie

- Indien zowel de Omega en grote O dezelfde functie zijn

Gemakkelijkste om zowel boven- als ondergrens zo specifiek mogelijk te definiëren

Tilde Notatie

f(x) ~ g(x)
- Als limiet (x->oo) van f(x) / g(x) = 1

$10x^2 + 5x + 3$ --> $\sim 10x^2$ --> $O(x^2)$
$10x^2 + 100x - 100^2$ --> $\sim 10x^2$ --> $O(x^2)$

$10^{-10}x^2 + \ldots$ --> $\sim 10^{-10} x^2$ --> $O(x^2)$

<u>In dit vak enkel tilde notaties, geen grote O! Veel preciezer en duidelijker</u>

Nagekeken en vervolledigd op 31-05-2023

Les 2 - Sorting

vrijdag 24 februari 2023 10:35

Liam Luys – r0934449 - 1e BA Informatica
Gegevensstructuren en Algoritmes

KU LEUVEN

2. Sorting Algorithms

Verschillende aspecten
- Soms makkelijk vergelijken en moeilijk verplaatsen (dieren)
- Anders moeilijk vergelijken en makkelijk verplaatsen (arrays)

Selection Sort

Werking Selection Sort
- Kleinste element zoeken en vooraan zetten
- Telkens het kleinste element uit de niet-gesorteerde lijst zoeken en vooraan zetten

Hoeveel vergelijkingen om lijst 6 elementen te sorteren?
- Eerste keer 5, dan 4, 3, 2, 1 ==> Samen 15 vergelijkingen
- (n-1) + (n-2) + …. + 1 = n(n-1)/2 met n = 6

- Als asymptotische benadering beschreven door $\sim n^2/2$ *(tilde-notatie)*, $O(n^2)$ *(O-notatie)*

Hoeveel verplaatsingen?
- n-1 aantal verplaatsingen
- ~n

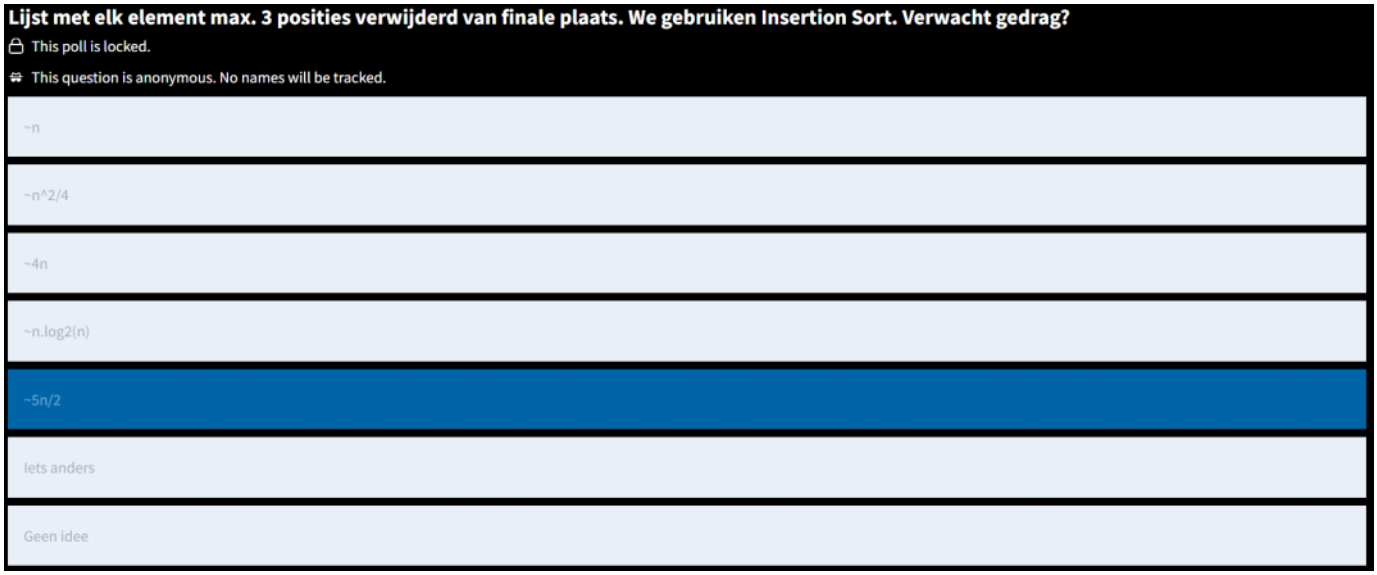
Insertion Sort

Werking Insertion Sort
- In het niet gesorteerde gedeelte het eerste element verplaatsen naar links
- Zo lang de voorganger kleiner is blijven verplaatsen tot het element op de juiste plaats staat.

Hoeveel vergelijkingen/verwisselingen?
- Best Case
  - Elk element links is kleiner dan het element (rij is gesorteerd)
  - n vergelijkingen nodig --> ~n
  - Geen enkele verwisseling

- Worst Case
  - Lijst is omgekeerd gesorteerd
  - 1+2+3+4+…+(n-1) vergelijkingen nodig --> $\sim n^2/2$
  - Aantal verwisselingen evenveel --> $\sim n^2/2$

- Gemiddelde
  - Gemiddeld gedrag
  - Alle plaatsen even veel kans, gemiddeld zal je dan in de helft eindigen
  - Aantal vergelijkingen en verwisselingen: $\sim n^2/4$



Element kan maximaal 3 plaatsen opschuiven

- Mogelijk aantal vergelijkingen: 1 of 2 of 3 of 4, elk met gelijke kans
- Gemiddeld 10/4 vergelijkingen bij asymptotisch gedrag = 5/2 vergelijkingen
- ~5/2

- Worst Case = ~4n
- Best Case = ~n

Merge Sort

Werking Merge Sort

- Lijsten in twee delen en beide delen sorteren
- Daarna twee halve delen in elkaar schuiven

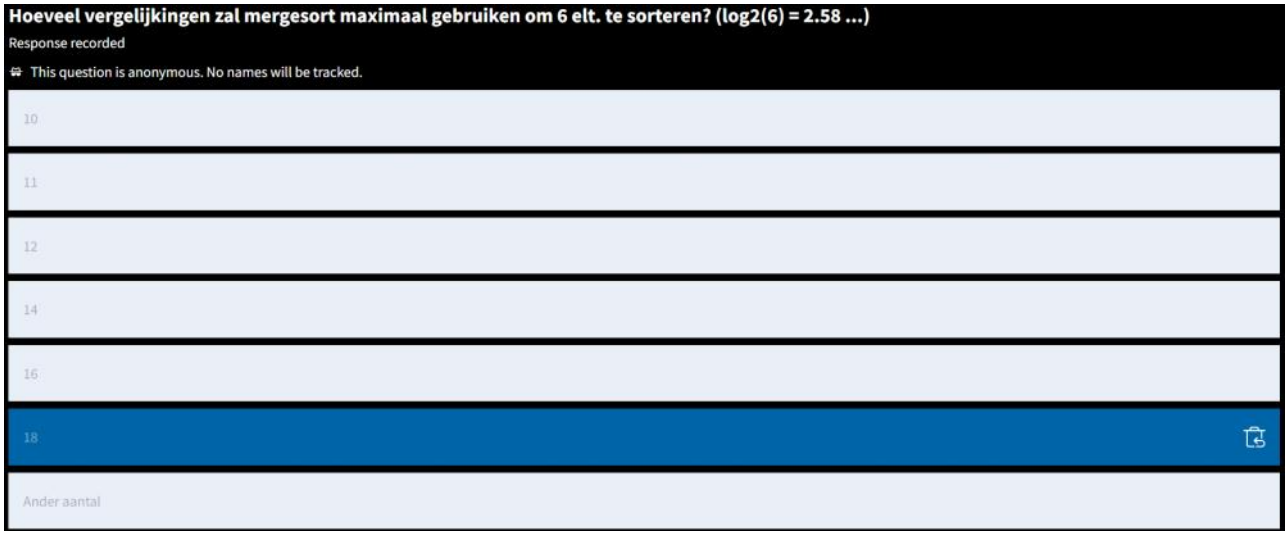
Analyse Merge Sort

- $n\log_2 n$ algoritme

- Worst:
  - Elke element in lijst is resultaat van vergelijking van de twee deellijsten --> n-1 vergelijkingen (de laatste wordt niet meer vergeleken) per rij in de recursie
  - Bij n lengte van originele lijst: 1e niveau (n-1), 2e (n-2), 3e (n-4), ….
  - Aantal niveaus die je hebt = Binair logaritme van 2 *(hoe vaak kan je n delen door 2 voor de lengte 1 is?)*
  - Totaal aantal vergelijkingen: $n\log_2 n$ - (1+2+4+8+16+…) --> $\sim n log_2(n)$

- Best: n/2 aantal vergelijkingen nodig per niveau
  - Totaal aantal vergelijkingen: $(n/2)\log_2 n$ --> $\sim(n/2)\log_2 n$

Elk sorteeralgoritme op basis van vergelijkingen heeft minstens ~$n\log_2 n$ vergelijkingen in gemiddeld geval



(fout, juiste antwoord 11)

Hier gaat het over het exact aantal vergelijkingen!
Elk element bij Merge Sort is het resultaat van een vergelijking van elementen in de deellijsten (behalve het laatste)

Waarom 11 en niet nlogn?

Merge Sort is een tilde nlogn algoritme, de tilde zegt iets over grote orde-termen, als je precies wil berekenen zou je de kleinere orde-termen ook in rekening moeten brengen en kom je iets anders uit.

Tilde nlog2n zegt niets over het exacte aantal, maar geeft de stijging weer

Hoe snel kunnen sorteeralgoritmes gaan?

Vergelijkende Sorteringen
- Elementen sorteren via vergelijkingen
- Merge Sort --> Asymptotisch gezien het beste algoritme wat we kunnen maken om te sorteren
    - Aantonen via bewijs dat nlog2n de beste mogelijke algoritme-grens kan zijn, waardoor Merge Sort het beste is.
        - Bewijs van $\log_2(n!)$ pagina 280-281

Afleiding van Stirling: $log_2(n!) = \sim nlog_2(n)$

$log_2(n!) = log_2(e) * \ln(n!)$

$$\ln(n!) = \ln(1) + \ln(2) + \ln(3) + \cdots + \ln(n) = \int_1^n \ln(x)\ dx = [x\ln(x) - x]_1^n = n\ln(n) - n + 1$$

$log_2(n!) = log_2(e) * (n\ln(n) - n + 1) = nlog_2(n) - nlog_2(e) + log_2(e)\ \ (benadering)$

--> $\sim nlog_2(n)$

Geen enkel sorteeralgoritme kan sneller dan $\sim nlog_2(n)$ op basis van vergelijking tussen elementen

Ander voorbeeld: Biljartballen *(zie slides)*

Nagekeken en vervolledigd op 31-05-2023

Les 3 - Sorting

vrijdag 3 maart 2023 10:30

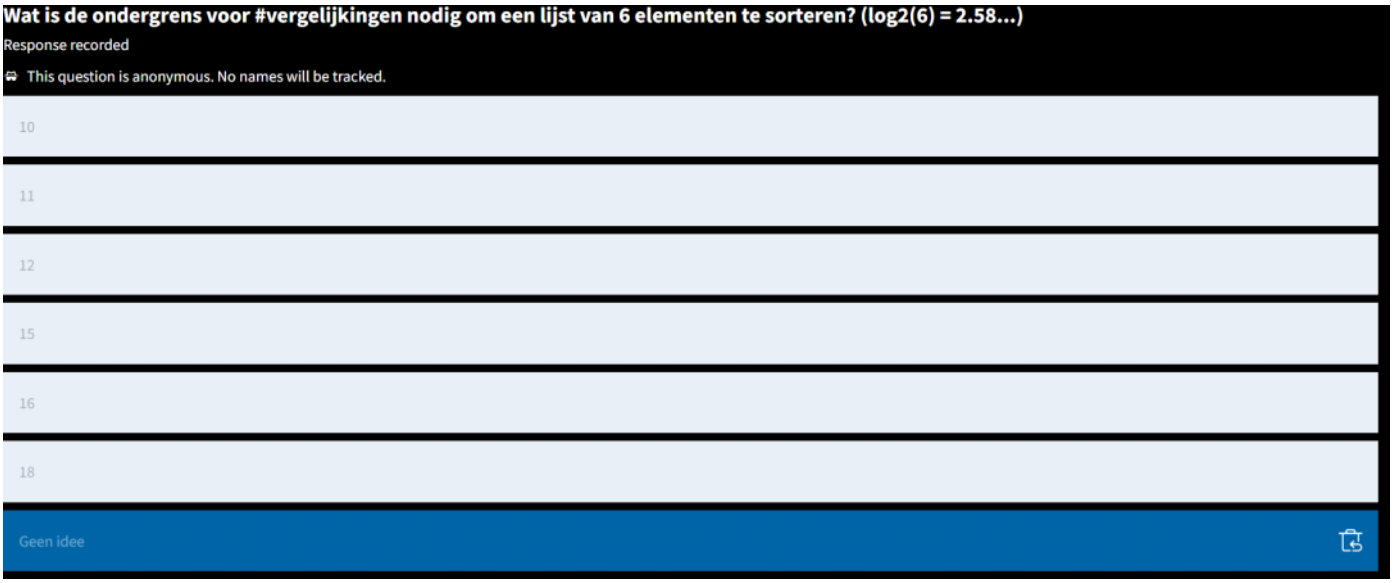
Liam Luys – r0934449 - 1e BA Informatica
Gegevensstructuren en Algoritmes

KU LEUVEN

Hoe snel kunnen sorteeralgoritmes gaan?

Vergelijkende Sorteringen
- Elementen sorteren via vergelijkingen
- Merge Sort --> Asymptotisch gezien het beste algoritme wat we kunnen maken om te sorteren $(\sim nlog_2(n))$

Niet-Vergelijkende Sorteringen
- Elementen sorteren zonder onderlinge vergelijkingen van elementen
- Counting Sort
- Radix Sort
- Bucket Sort



(goede antwoord 10)

6 Elementen --> 6! = 720
Diepte van de boom is dan log2(720) = 9,491853096329675 --> 10 vergelijkingen is de ondergrens!

Onderscheid belangrijk tussen concepten van exact, benadering, tilde, grote o, lagere orde-termen, …

QuickSort

Werking Quick Sort
- Kies de Pivot
- Vergelijk met alle elementen, kleiner komen in linker helft, groter komen in de rechter helft
- De pivot komt tussen de twee deellijsten en staat op zijn plaats
- De linker-en rechterhelft recursief oplossen

Sorteren
- Recursief

Partitionering: 3 Varianten

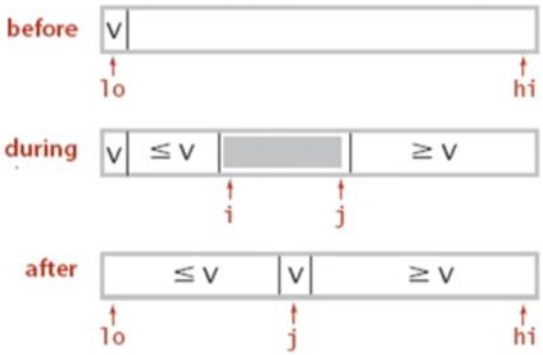
Partiëren (Pivot Kiezen)

- **Variant 1**
  - n-1 vergelijkingen (is goed!)
  - 2 Nieuwe arrays left[] en right[]
  - Daarna twee lijsten bij elkaar voegen

  Hier gooi je het idee van QuickSort weg, het memorymanagement is veel moeilijker (je hebt extra geheugen nodig voor de left en right) en de tijd gaat naar boven

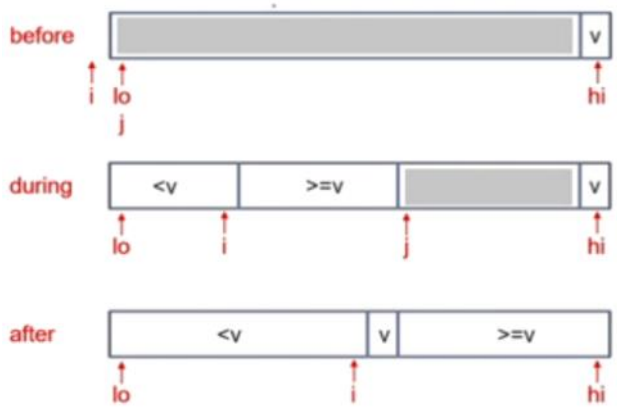
  - Nadelen
    - Hoe groot moeten de left en right zijn? Onbekend!
    - Twee lijsten bij elkaar voegen --> Kost zelf een lineaire tijdsschaal!
    - Het idee van in-place te werken wordt weggegooid
    - Zal nog in ~nlogn tijd werken, maar constante in absolute tijd gaat naar boven!

- **Variant 2** *(Hoare)*
  - n+1 vergelijkingen
  - Meest ingewikkelde variant maar beste efficiëntste methode
  - Eerste element als pivot kiezen, en in de lijst elk element vergelijken en bijhouden welk stuk nog niet vergeleken is. Pivot blijft vooraan staan en wordt uiteindelijk verwisselt met laatste element van het linkse (kleinere) deel



  - n+1 vergelijkingen
    - 2 vergelijkingen meer
    - Op het laatste komen i en j bij elkaar, kruisen elkaar en doen zo 2 vergelijkingen te veel
  - Relatief weinig verplaatsingen
    - Elke verplaatsing doet iets nuttigs en is nodig

- **Variant 3** *(Lomuto)*
  - n-1 vergelijkingen, wel meer verplaatsingen
  - Minder efficiënt, makkelijker en compacter te implementeren
  - Pivot wordt op het einde gekozen, groep gesorteerd op kleiner en groter dan pivot, afgebakende groep die nog niet gesorteerd/vergeleken is.
    Pivot wordt verwisseld met eerste element van de rechtse groep
  - Overlopen ongeziene elementen
    Is element groter dan pivot --> Staat al juist in rechtse groep, j mag 1 opschuiven
    Is element kleiner dan pivot --> Verwisselen met eerste element van groep groter/gelijk aan pivot



- Meer verplaatsingen dan bij Hoare, minder vergelijkingen

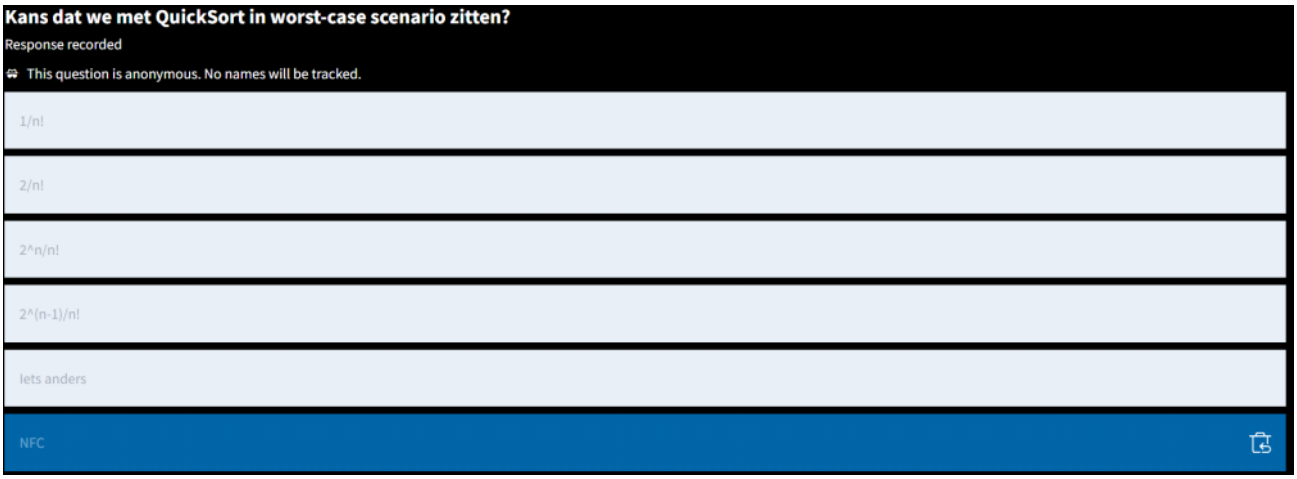
Leg partitionering van Hoare en Lomuto uit

Analyse van QuickSort

Analyse QuickSort

- Best Case
  - Veronderstellen dat pivot altijd in het midden staat, rij wordt precies in 2 delen gesplitst
  - Partitionering neemt altijd n vergelijkingen
  - $\sim nlog_2(n)$ (even goed als Merge Sort)

- Worst Case
  - Eén van de deelrijen na kiezen pivot is altijd leeg
  - Deelrijen grootte (n-1), (n-2), (n-3), ….
  - $\sim \frac{n^2}{2}$ (even "goed" als Selection Sort)



(fout, juiste is 4e)

Altijd 2 kansen dat je in worst case komt --> Of je pakt het grootste, of het kleinste element in de deellijst

Voor n elementen altijd de kleinste of grootste element kiest --> $\frac{2}{n}$ kans

Voor (n-1) --> $\frac{2}{n-1}$

Voor (n-2) --> $\frac{2}{n-2}$

….

$$\frac{2}{n} * \frac{2}{n-1} * \frac{2}{n-2} * \cdots * \frac{2}{2} = \frac{2^{(n-1)}}{n!}$$

De kans dat je ooit in de worst case van QuickSort komt is zeer klein!

Implementatie QuickSort

Belangrijke aspecten bij implementatie QuickSort

- Partitionering in Place
  - Een nieuwe array maken zou het algoritme zeer hard vertragen, daarom moet alles in-place gebeuren

- Tussen grenzen blijven
  - Zorgen dat de i en j niet buiten de array gaan

- Zorgen voor willekeurigheid
  - Enerzijds zorgen dat de rij geen structuur bevat in het begin
  - Andere mogelijkheid om de pivot random te kiezen

- Beëindigen van de loop
  - Zorgen dat de lus in algoritme altijd eindigt
  - Rekening houden dat er meerdere elementen gelijk (in waarde) kunnen zijn aan de pivot

- Behandelen van elementen met gelijke waarde als pivot
  - Zorgen dat je groter/kleiner dan of gelijk aan gebruikt in algoritme
  - Vermijden van kwadratisch gedrag

- Beëindigen van de recursie
  - Zorgen dat je niet in een oneindige lus terecht komt

Wat maakt QuickSort snel?

- MergeSort doet in de binnenste lus van het algoritme ook een verplaatsing, QuickSort enkel een vergelijking (-> Sneller)
- Gebruikt minder vergelijkingen

Nagekeken en vervolledigd op 01-06-2023

Les 4 - Sorting

vrijdag 10 maart 2023 10:27

Liam Luys – r0934449 - 1e BA Informatica
Gegevensstructuren en Algoritmes

KU LEUVEN

Quicksort

Zelf manueel de code schrijven, begrijpen hoe het werkt

Aantal Vergelijkingen

Algoritme	Best Case	Gemiddeld	Worst Case
Selection Sort	$\sim\frac{n^2}{2}$	$\sim\frac{n^2}{2}$	$\sim\frac{n^2}{2}$
Insertion Sort	$\sim n$ Reeds gesorteerd	$\sim\frac{n^2}{4}$ Gemiddeld gezien	$\sim\frac{n^2}{2}$ Omgekeerd gesorteerd
Merge Sort	$\sim\frac{1}{2}nlog_2(n)$ 1 helft gemerged, andere gekopieerd	$\sim 0{,}74\ nlog_2(n)$	$\sim nlog_2(n)$ Alle elementen resultaat van vergelijking tussen twee deelrijen
Quick Sort	$\sim nlog_2(n)$ Deelrijen steeds even groot	$\sim 1{,}39\ nlog_2(n)$	$\sim\frac{n^2}{2}$ Telkens deelrij van 1 en (n-1)

Max	***Selection Sort***	***Selection Sort***	***Selection Sort / Insertion Sort***
Min	***Merge Sort***	***Merge Sort***	***Merge Sort***

Best mogelijk	***?***	$\sim nlog_2(n)$	$\sim nlog_2(n)$

★ Examen: Vat alles samen zoals deze tabel

Analyse QuickSort

Best Case QuickSort

- Rij wordt telkens in 2 even grote delen gesplitst
- Aantal keer dat je door 2 kan delen met n elementen --> $log_2(n)$
- Bij elke rij in de boom maak je n vergelijkingen

Totaal aantal vergelijkingen = $nlog_2(n)$

Wat als je niet perfect in twee splitst?

- Stel telkens in 1/10 en 9/10
- Elke rij in de boom blijft n vergelijkingen
- Aantal rijen is het aantal keer dat je n kan delen door 9/10 --> $log_{\frac{10}{9}}(n)$

$log_{\frac{10}{9}}(n)$ = $log_{\frac{10}{9}}(2)$ * $log_2(n)$ = constante * $log_2(n) = \sim c\ log_2(n)$

Constante hangt af van verdeling van lijsten, van de maximale diepte

Zolang je fracties ($\frac{10}{9}, \frac{1}{2}, \frac{99}{100}, \dots$) afsplitst, zal het een logaritmische functie ($\sim c\ log_2(n)$) blijven met een andere c
Als je geen fracties maar vast aantal elementen gaat afsplitsen (1e getal, laatste getal, 0 elementen, …)--> Worst Case en niet meer logaritmisch, wel kwadratisch

$$\sim \boldsymbol{c\ log_2(n) \rightarrow c\ liefst\ zo\ klein\ mogelijk}$$

Slechte stap zetten in QuickSort

- Eén of meerdere slechte stappen --> QuickSort zal hiervan herstellen en blijft logaritmisch voor grote n
- Als je veel slechte stappen zet (bv elke tweede stap een slechte) --> Algoritme niet meer logaritmisch maar kwadratisch!

Examen --> Kunnen afleiden van ~1,39 nlog2n van QuickSort

Afleiding 1 (Average Case QuickSort): Top-Down

Mogelijke scenario's als je willekeurige pivot kiest en rij zonder structuur

- Linkse array 0 elementen, pivot, rechtse array (n-1) elementen
- Links 1 element, pivot, rechts (n-2) elementen
- Links 2 elementen, pivot, rechts (n-3) elementen
- …
- Links (n-1) elementen, pivot, rechts 0 elementen

- Totaal n mogelijke scenario's, elke mogelijkheid heeft 1/n kans om voor te komen

$C_n: aantal\ vergelijkingen$

$$C_n = (n+1) + \frac{1}{n}(C_0 + C_{n-1}) + \frac{1}{n}(C_1 + C_{n-2}) + \frac{1}{n}(C_2 + C_{n-3}) + \cdots + \frac{1}{n}(C_{n-1} + C_0)$$

$$C_n = (n+1) + \frac{2}{n}(C_{n-1} + C_{n-2} + \cdots + C_1 + C_0) \quad \rightarrow Vervelend, we\ willen\ recursieve\ functie$$

$$C_n = (n+1) + \frac{2}{n}(C_{n-1} + C_{n-2} + \cdots + C_1 + C_0) \quad \rightarrow Formule\ voor\ C_n$$
$$nC_n = n(n+1) + 2(C_{n-1} + C_{n-2} + \cdots + C_1 + C_0) \quad \rightarrow Vereenvoudigd\ breuk$$

$$C_{n-1} = n + \frac{2}{n-1}(C_{n-2} + C_{n-3} + \cdots + C_1 + C_0) \quad \rightarrow Herschrijving\ formule\ naar\ C_{n-1}$$
$$(n-1)C_{n-1} = (n-1)n + 2(C_{n-2} + C_{n-3} + \cdots + C_1 + C_0)$$

$$nC_n - (n-1)C_{n-1} = n(n+1) - (n-1)n + 2C_{n-1}$$

$$nC_n = n^2 + n - n^2 + n + 2\ C_{n-1} - (n-1)\ C_{n-1}$$
$$nC_n = 2n + (n+1)\ C_{n-1}$$

$$\frac{C_n}{n+1} = \frac{2}{n+1} + \frac{C_{n-1}}{n} \quad \rightarrow Recursieve\ vergelijking\ van\ C_n\ in\ termen\ van\ C_{n-1}$$

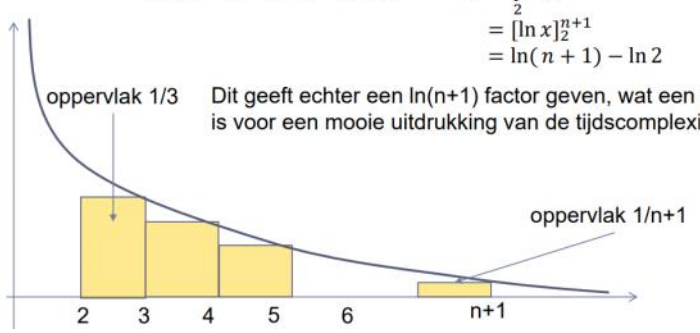
Recursie uitwerken

$$\frac{C_n}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{C_{n-2}}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \frac{2}{n-2} + \cdots + \frac{2}{3} \quad \rightarrow Stoppen\ bij\ \frac{2}{3}\ omdat\ voor\ 2\ of\ 1\ element\ er\ geen\ kost\ meer\ is$$

$$\frac{C_n}{n+1} = 2\left(\frac{1}{n+1} + \frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \cdots + \frac{1}{3}\right) \quad \rightarrow Harmonische\ som, kunnen\ we\ benaderen\ door\ een\ integraal!$$

We willen volgende som benaderen:

$$(\frac{1}{n+1}+\frac{1}{n}+\frac{1}{n-1}+\frac{1}{n-2}+\ldots+\frac{1}{3}) \approx \int_2^{n+1} \frac{1}{x}dx$$
$$= [\ln x]_2^{n+1}$$
$$= \ln(n+1) - \ln 2$$



$\frac{C_n}{n+1} = 2\int_2^n \frac{1}{x}dx + \frac{1}{x+1}$ $\rightarrow \frac{1}{x+1}$ *is van het laatste balkje* $(n+1)$

$\frac{C_n}{n+1} = 2\left([\ln(x)]\frac{n}{2} + \frac{1}{x+1}\right) = 2\left(\ln(n) - \ln(2) + \frac{1}{x+1}\right)$

$\Leftrightarrow C_n = 2(n+1)\left(\ln(n) - \ln(2) + \frac{1}{x+1}\right)$ $\rightarrow$ *Negeer lage ordetermen voor* $\sim$ *notatie*

$= \sim 2n * \ln(n) = \sim 2n * \ln(2) * log_2(n) = \sim 1{,}39\, n\, log_2(n)$

Dit is het asymptotisch gedrag, waarbij de lagere ordetermen worden weggelaten!
Als je geïnteresseerd bent in exacte waardes mag je die niet weglaten.

Afleiding 2 (Average Case QuickSort): Bottom Up

Wat is de kans dat twee elementen ooit met elkaar worden vergeleken?

Als je de kansen van alle mogelijke vergelijkingen optelt, heb je een verwachting van aantal vergelijkingen

(juist)

Ze worden enkel vergeleken als één van beide de pivot is.
De kans dat 1 of 10 als pivot wordt gekozen is 1/10 + 1/10 = 0,2 (20%)
Ze gaan altijd in een andere deellijst komen (omdat het de uiterste elementen zijn) bij andere pivots, dus de totale kans is 20%

(juist)

Er kan geen pivot tussen 5 en 6, dus uiteindelijk worden ze sowieso met elkaar vergeleken

2 elementen die vlak naast elkaar komen in de gesorteerde rij, worden altijd vergeleken met elkaar

(fout)

Zolang de pivots kleiner dan 1 of groter dan 10 zijn, worden ze niet met elkaar vergeleken.
Tot het moment dat je pivot kiest tussen 1 en 10, dan worden ze gescheiden in aparte groepen.
Ze worden wel vergeleken als de pivot 1 of 10 is. Er zijn 2 kansen dat ze vergeleken worden.
Tussen 10 en 1 zitten 10 mogelijke pivots, 2 daarvan zijn 1 en 10 zelf dus 20% (Antwoord C)

De kans hangt dus niet af van de lengte van de lijst, maar van het aantal tussenliggende elementen

- ***Hoe dichter elementen bij elkaar in gesorteerde lijst, hoe meer kans dat ze vergeleken zijn***

  Kans: $P(Z_i <-> Z_j) = 2/(j-i+1)$

Bottom-Up manier om QuickSort te beschrijven

$$C_n = \sum_{i=1}^{n}\sum_{j=i+1}^{n-1} p_{ij} = \sum_{i=1}^{n}\sum_{j=i+1}^{n-1} \frac{2}{j-i+1} \quad \xrightarrow[k=j-i+1]{} \quad C_n = 2 * \sum_{i=1}^{n}\sum_{k=2}^{n-i+1} \frac{1}{k}$$

$$C_n = 2 * \int_1^n \int_2^{n-i+1} \frac{1}{k}\, dkdk = 2 * \int_1^n \left([\ln(|k|)] \right)_2^{n-i+1} dk = 2 * \int_1^n \ln(n-i+1) - \ln(2)\, dk$$

$$C_n = 2 * \int_1^n \ln(n-i+1)\ dn - 2 * \int_1^n \ln(2)\, dn = 2\left(-\frac{1}{2}(n-1)(2i-n-3)\right) - 2n * \ln(2)$$

$$C_n = -2\ln(n) - 2n\ln(2)$$

$$C_n = \sim 2n * \ln(n) = \sim 2n * \ln(2) * log_2(n) = \sim 1{,}39\, n\, log_2(n)$$

Verdere uitwerking -> Nog niet in orde!!

Optimalisaties / Varianten QuickSort

Optimalisaties van QuickSort

- Cutt-Off naar Insertion Sort
  - Als je lijst dmv QuickSort al bijna gesorteerd is (bv de deellijsten maar 4 elementen lang zijn), verder werken met Insertion Sort om de deellijsten te sorteren
  - Insertion Sort vaak beter voor kleine waarde van n

- Gemiddelde van 3 random waardes als pivot
  - Proberen om de 1,39 te verkleinen
  - De kans vergroten dat pivot in het midden van de lijst komt te liggen --> Meer kans op Best Case

  - Gemiddelde van 3 keer het gemiddelde van 3 random waardes als pivot

- 3 Groeps-partitionering
  - Een partitionering toevoegen van elementen die gelijk zijn aan de pivot
  - Die elementen kunnen direct blijven staan
  - Is handig als je weet dat in je data meerdere dezelfde elementen zitten

Varianten op QuickSort

- Multi-pivot QuickSort
  - Gebruik maken van 2 of meerdere pivots
  - Recursieboom is minder diep, maar je hebt meer vergelijkingen nodig per niveau (vergelijking per pivot)
  - Weegt het op tegen elkaar?

Wat hebben we geleerd?

- Gedrag van sorteeralgoritmes
- Goede algoritmes zijn beter dan goede computers!
- Perfecte algoritmes zijn beter dan goede algoritmes

Nagekeken en vervolledigd op 01-06-2023

Les 5 - Lineaire Tijd

vrijdag 17 maart 2023 10:22

Liam Luys – r0934449 - 1e BA Informatica
Gegevensstructuren en Algoritmes

KU LEUVEN

Lineaire Sorteeralgoritmes (5.1 String Sorts)

Bij sorteren door het vergelijken van elementen kunnen we niet onder de $\sim n\ log_2(n)$ tijd gaan

Als we gaan sorteren maar niet door vergelijken kunnen we dit wel in lineaire tijd gaan doen!

Lineaire Algoritmes

- Worden vaak gebruikt op andere datatypes: Strings, …
- Gebruiken eigenschap van bepaalde structuur die al in de data zit
- Verschillende algoritmes
  - Bucket Sort
  - Counting Sort

Bucket Sort

Werking Bucket Sort

- Verdeel de invoerreeks in verschillende buckets
  - Op basis van eigenschappen
- Plaats elk element in de juiste bucket
- Sorteer elke bucket door vergelijking-algoritmes (Merge/Quick/Selection Sort)
  - Zodra element in bucket komt direct juist zetten en sorteren binnen de bucket
- Voeg de buckets samen in een lijst

Worst Case: 1 bucket is vol, de rest is leeg
Best Case: Alle buckets zijn evenredig verdeeld

Hoe zorg je voor best case?

- Buckets strategisch kiezen zodat je een zo evenredig mogelijke verdeling krijgt
- Informatie over structuur van data gebruiken om buckets te verdelen
  - Welke kans hebben elementen? Praktisch domein?

Analyse BucketSort

Hoeveel buckets?

- Neem aan dat sorteeralgoritme per bucket werkt voor $\sim c\ * n\ log_2(n)$
- Average Case: $\sim k * c * \frac{n}{k} * log_2\left(\frac{n}{k}\right) = \sim cn\ log_2\left(\frac{n}{k}\right)$
  - Als k een constante --> $\sim cn\ log_2(n)$
  - Als k afhangt van n --> $log_2\left(\frac{n}{k}\right)\ is\ constant \rightarrow \sim c' * n \quad \left(met\ c' = c * log_2\left(\frac{n}{k}\right)\right)$

Werkt enkel goed als je op voorhand weet hoeveel elementen je hebt, en een goede keuze van buckets om een zo evenredig mogelijke verdeling te maken van elementen in de buckets

Counting Sort / Key-indexed Sort

Aantal voorkomens van een element tellen in een lijst

Afhankelijk van hoeveel verschillende waardes in lijst zitten!

★ **Belangrijk voor het examen**

Algoritme begrijpen en zelf kunnen schrijven.

Complexiteit?
- We vergelijken geen elementen
- Complexiteit uitdrukken door het aantal array-accesses (opvragen/plaatsen/bewerken)

- Lineair in het aantal originele elementen + Lineair in het aantal waardes die de elementen kunnen bevatten
- Zowel $\sim n$ als een $\sim r$ (aantal mogelijke waarden)

- $8n + 3r + 1$ array accesses (afgeleid uit de code)

Werking Counting Sort
- Aantal keer voorkomens tellen in lijst, resultaat in andere lijst
    - Werkt enkel met beperkt aantal mogelijke waarden in de lijst

- Accumuleren in count lijst
- Elementen in juiste volgorde plaatsen, count lijst aanpassen om open plaats goed bij te houden

Soorten Sorteeralgoritmes

Soorten Algoritmes
- In Place Algorithm --> Algoritme past elementen in lijst zelf aan
- Stable Algorithm --> Elementen met dezelfde sorteerwaarde veranderen onderling nooit van volgorde

Is het belangrijk om stabiel te zijn?
- Als je eerst al een gesorteerde lijst had volgens een record-waarde en dan op een andere waarde sorteert
    - Bij stabiel algoritme blijft ook je eerste sortering behouden binnen de tweede

<u>Counting Sort is een stabiel algoritme!</u>

Toepassingen Counting Sort
- LSD Sort - Least Significant Digit Sort
- MSD Sort - Most Significant Digit Sort

★ ***Beide uit elkaar houden en afzonderlijk kunnen uitleggen!***

Least Significant Digit Sort

LSD-Sort / Radix Sort
- Karakters bekijken van rechts naar links
- Stabiel sorteren van de karakters van rechts naar links (door Counting Sort)
- Uiteindelijk is alles gesorteerd

- $\sim 7Wn + 3Wr$ array accesses + extra ruimte nodig van $n + r$ om n karakters te sorteren
    - W lengte strings

    - r aantal verschillende mogelijke karakters
    - Lineair algoritme t.o.v. n

- **Werkt enkel op strings van gelijke lengte**

Waarom nuttig?
- In sommige gevallen is het zeer moeilijk om data rechtstreeks met elkaar te vergelijken
- vb lange strings met elkaar vergelijken, dan gaat het sneller om het karakter per karakter te doen (LSD Sort)

Toepassing: Punch Cards

Most Significant Digit Sort

MSD-Sort
- Karakters bekijken van links naar rechts
- Eerst op 100-tallen, daarna op 10-tallen, …
- Werkt niet om aparte karakters te sorteren van links naar rechts!

Hoe werkt het algoritme wel?
- Sorteren op most significant digit (meest links)
- Die sortering behouden, en binnen de groepen van verschillende waardes sorteren op het tweede karakter
- Recursieve oproep binnen verschillende gesorteerde karakters
- Zo blijven de gesorteerde karakters staan, en worden vervolgens de volgende karakters bekeken

- MSD Sort vergelijkt enkel de karakters die echt nodig zijn om onderscheid te maken tussen de strings

- Strings moeten niet noodzakelijk gelijke lengte hebben

- Gemiddeld gezien $n\ log_R(n)$ karakters vergeleken (niet kunnen afleiden)
    - Hoe vaak n delen door R totdat je 1 uitkomt --> $log_R(n)$ (diepte recursie) met n hoeveelheid werk per niveau

- Tussen $8n + 3r$ en $\sim 7Wn + 3Wr$ array accesses
    - W gemiddelde lengte strings
    - r aantal verschillende mogelijke karakters
    - Uit code afleiden, in best case maar 1, in worst case gelijk aan LSD

- Plaats nodig: $r\ *\ lengte\ langste\ string\big(diepte\ recursie\big) + n\ (voor\ aux\ array)$

3-way string QuickSort

3-way QuickSort
- Partities <, >, = met gekozen pivot
- Recursief uitwerken op < en > met nieuwe pivots
- In de = partitie naar het volgende karakter kijken

Telkens vergelijken op 1 karakter, afhankelijk van positie een karakter opschuiven.

Enkel karakter opschuiven in partitie waarbij elementen gelijk zijn aan pivot
- In de andere moet je nog het eerste element sorteren!

Complexiteit
- $\sim 1{,}39n\ log_2(n)$

Zie slides voor verduidelijking algoritme

Tijdscomplexiteit vanbuiten kennen van elk gezien algoritme

Samenvatting

Wat hebben we geleerd?

- Veel mogelijke sorteeralgoritmes
- Keuze algoritme hangt hard af van de structuur

- Insertion Sort
    - Vaak slecht algoritme, maar als lijst al bijna gesorteerd is werkt het goed!

- Bij een recursief algoritme --> Vaak in logaritmische tijdscomplexiteit

- Algoritmes reduceren tot een ander algoritme --> Dan zijn de complexiteit gelijk

Nagekeken en vervolledigd op 01-06-2023

Les 6 - Tree / Heap

vrijdag 24 maart 2023 10:31

Liam Luys – r0934449 - 1e BA Informatica
Gegevensstructuren en Algoritmes

KU LEUVEN

Priority Queues & Search Trees

Prioriteitsrijen

Priority Queue
- We willen objecten bijhouden in een structuur, kunnen toevoegen en volgens ordeningsrelatie een element kunnen terugvinden
- Grootste element telkens verwijderen

Toepassingen Priority Queue
- Statistieken
- Besturingssystemen
- AI, stromen van data

Stack
- Datastructuur; Verzameling van objecten
- Geen ordeningsrelatie, enkel de volgorde waarin elementen op de stack staan.
- Nieuwste element verwijderen

Queue / Wachtrij
- Datastructuur
- Fifo structuur --> First in First Out
- Oudste element verwijderen

Priority Queues
- Bijhouden adhv een ongesorteerde lijst
  - Elementen toevoegen is makkelijk --> Gewoon achteraan toevoegen
  - Maximum vinden is moeilijk --> Je moet alle elementen overlopen

- Bijhouden adhv een gesorteerde lijst
  - Elementen toevoegen moeilijk --> Moet in gesorteerde lijst komen
  - Maximum vinden is makkelijk --> Laatste element van de gesorteerde lijst

Wat is beter? Ene manier moeilijk voor toevoegen, ander voor vinden van maximum…
  - Kan afhangen van applicatie
    - Moet je meer toevoegen dan opvragen? Dan is ongesorteerd beter!

Wat willen we bereiken?
- Logaritmische tijd voor zoeken van maximum en toevoegen van elementen.

Hoe beide situaties verbeteren?
- Deels geordende array
- Werken met heap

Binaire Boom / Binary Tree

Binary Tree / Binaire Boom
- Kan leeg zijn
- Knopen met links en rechts een tak

Compleet Binaire boom
- Alle niveaus mooi opgevuld, gebalanceerd (uitzondering voor onderste niveau, van links naar rechts)
- Hoogte van boom met N knopen is binair logaritme: $\log_2 N \ \ (naar\ beneden\ afgerond)$

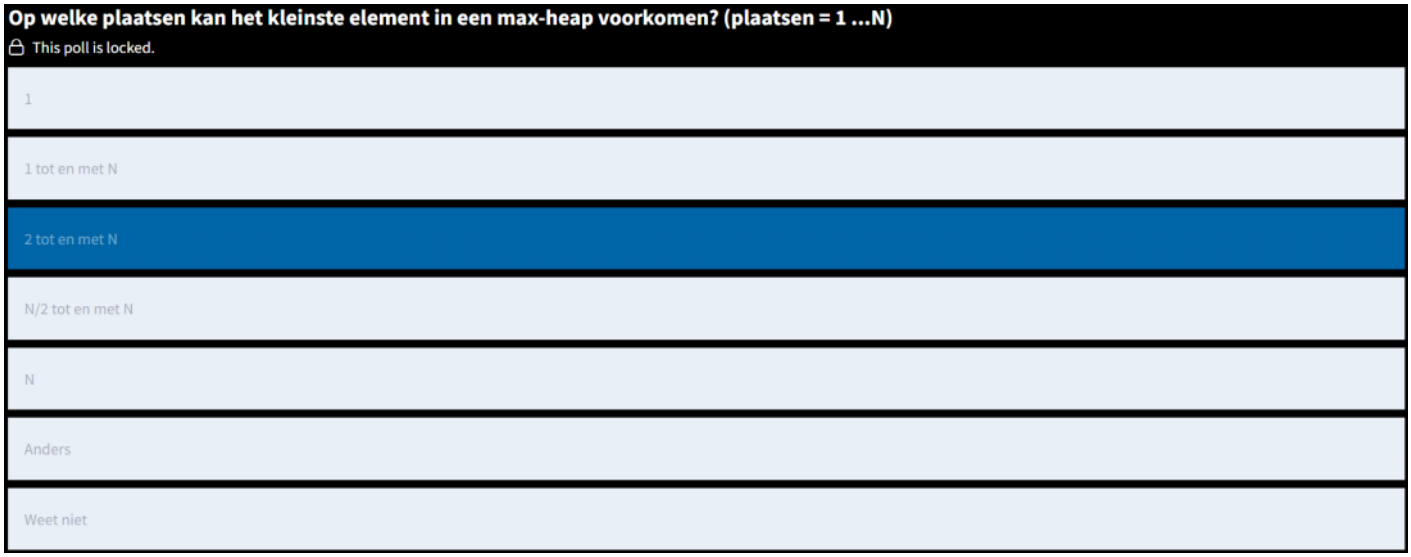
Binary Heap

Binary Heap

- Datastructuur die de Priority Queue voorstelt
- Maakt gebruik van compleet binaire boom
    - Hoogste element staat bovenaan in de heap
    - We kunnen knopen nummeren
    - Operaties afhankelijk van hoogte van boom ($\log_2 N$)

- Legt voorwaarde op knoop-waardes
    - Elke knoop is groter dan zijn (twee) kinderen
    - Maximale waarde staat bovenaan

- Binaire boom voorgesteld in een array
    - Knopen van boven naar onder, van links naar rechts in array
    - Geen expliciete linken tussen elementen noodzakelijk
    - Parent van knoop k --> k/2 (afgerond naar beneden)
    - Kinderen van knoop k --> Op 2k en 2k+1



(fout)

Kleinste element kan geen kinderen hebben --> Enige posities die dat zijn: N/2 + 1 tot N

Operaties op Heap

Promotion / Promotie in Heap (swim)

- Knoop vervangen door grotere waarde
- Kijken of knoop groter is dan ouder, wisselen indien dat het geval is
- Herhalen tot heap terug geordend is

- Maximaal hele boom afgaan, hoogte van de boom --> Maximaal 1 + $\log_2$N vergelijkingen

Insertion / Toevoegen in Heap

- Knoop toevoegen op de laatste positie, dan terug 'zwemmen' om knoop op de juiste plaats te zetten

Demotion / Demotie in Heap (sink)

- Knoop vervangen door kleinere waarde

- Vergelijken met je kinderen, verwisselen met je grootste kind als je kleiner bent
- Per niveau kost het 2 vergelijkingen (1 per kind)

- Maximaal boom afgaan, hoogte van de boom en telkens 2 vergelijkingen per niveau --> Max $2\log_2 N$

Delete Maximum
- We weten dat het maximum bovenaan staat
- Na verwijderen moet structuur van heap geldig blijven
- Laatste element helemaal bovenaan zetten, via demotie (sink) terug op juiste plaats zetten

Resultaat tijdscomplexiteit op Binary Heap
- Insert: $\log_2 N$
- Delete Max: $2\log_2 N$

Heap Opbouwen

Optie 1: via heap-operaties
- Elk element toevoegen onderaan, naar boven brengen tot het op zijn plaats staat
  - Heap wordt groter en groter

- Aantal vergelijkingen
  - $\log_2 1 + \log_2 2 \ \log_2 3 + \cdots + \log_2 N + N \quad (Stirling's\ Methode)$
  - $\sim N log_2 N \ \ totaal$

Optie 2: vanuit bestaande array
- Beginnen van de knoop op N/2 (laatste knoop met kinderen)
- Alle elementen afgaan en vergelijken met de kinderen, laten zakken (demotie) tot juist staat

- Aantal vergelijkingen
  - $\frac{N}{2}\ (aantal\ ouders)\ * 2\log_2 N\ (\text{max mogelijke compares per sink per ouder}) = N log_2 N\ (schatting)$
  - Exacte cijfers (niet zelf afleiden)
    - Hoe dieper in de boom, hoe minder diep elementen kunnen zakken
    - $2N$ vergelijkingen
    - N verplaatsingen

Heap Sort

Werking Heap Sort
- Eén voor een maximum uit de binary heap halen (bovenste element) en achteraan in finale array zetten
- Laatste element bovenaan zetten en laten sinken (zoals bij demotie)
- Als je telkens maximum elementen aanvult van achter naar voor is je rij gesorteerd

Complexiteit
- Bouwen van de heap $(N log_2 N)$ + Alle elementen verwijderen uit de heap
- $\sim 2n \log_2 n \ \ vergelijkingen\ maximaal$

- Eigenschappen
  - In-Place algoritme
  - Geen extra geheugen nodig
  - Niet stabiel

Binary Search Tree BST

Structuur

- Binaire Boom (geen heap!)
- Linker kind is kleiner dan de knoop, rechterkind is groter dan de knoop

Zoekoperatie

- Geef waarde terug of null indien de waarde niet in de boom staat
- Bovenaan beginnen, naar onder zoeken

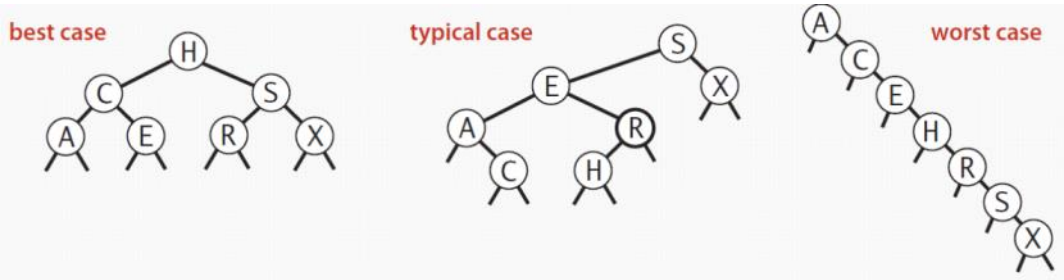
- Aantal vergelijkingen
  - Voor een Search hit: 1 + diepte van de knoop (1 omdat we diepte tellen vanaf 0)
  - Voor een Search miss: 1 + diepte van waar de knoop zou zijn

Waarde toevoegen

- Telkens onderaan de boom
- Zoek de waarde die je wil toevoegen in de boom
  - Zit die al in de boom?
  - Zit die nog niet in de boom? --> Toevoegen

- Aantal vergelijkingen
  - 1 + diepte van de knoop waar de insert terecht komt

- Volgorde van toevoegen bepaald de diepte!
  - Kan heel willekeurig zijn



Complexiteit

- Wat is de gemiddelde complexiteit, als we **<u>in willekeurig volgorde</u>** elementen toevoegen?
- $\sim 1{,}39 \log_2 N \;\; vergelijkingen \; gemiddeld$

Waarvan komt 1,39?

- We willen gemiddelde diepte van een toegevoegd element in de boom

Als je 1 knoop bovenaan toevoegt:

$P(T) = P(T_{links}) + \; P(T_{rechts}) + (n-1) \quad \rightarrow Totale \; diepte \; van \; alle \; n \; mogelijke \; knopen \; in \; de \; boom$

P(T) = som van diepte van alle knopen in de boom
(n-1) komt van elke knoop die 1 niveau dieper zakt door toevoeging van bovenste knoop

Gemiddelde diepte van elke knoop?

- $Gemiddelde \; diepte \; knoop = \frac{totale \; diepte \; alle \; knopen}{aantal \; knopen} = \frac{P(T)}{n}$

- Mogelijke situaties van een boom
  - Links 0 elementen, rechts (n-1)
  - Links 1 element, rechts (n-2)
  - …
  - Links (n-1) elementen, rechts 0

Elke situatie heeft een gelijke kans om voor te komen ($\frac{1}{n}$)
Gemiddeld totale diepte $P(T_n)$? Alle mogelijkheden optellen

ZELFDE REDENERING ALS AFLEIDING VAN QUICKSORT! --> $\sim 1{,}39\ n\ log_2(n)$

$P(T_n) = 1{,}39 n log_2 n \quad \rightarrow Gemiddelde\ totale\ diepte\ boom$

$\frac{P(T_n)}{n} = 1{,}39 \log_2 n \quad \rightarrow Gemiddelde\ diepte\ knoop\ in\ boom\ voor\ search\ hit$

- Is iets van verzameling die je op willekeurige wijze in 2 delen splitst (en blijft doen) mbv één element (pivot/top)
  - Komt in vele algoritmes voor (QuickSort, BST, …)
  - Is een design-template

implementation	guarantee		average case	
	search	insert	search hit	insert
sequential search (unordered list)	N	N	N/2	N
binary search (ordered array)	lg N	N	lg N	N/2
BST	N	N	1.39 lg N	1.39 lg N

BST Ordered Operations

Minimum vinden in BST
- Zo ver mogelijk naar links gaan in de boom
- Linker kind is steeds kleiner dan rechter

Maximum vinden in BST
- Zo ver mogelijk naar rechts gaan in de boom
- Rechter kind is steeds kleiner dan rechter

Floor vinden in BST
- Grootste element, kleiner dan of gelijk aan gegeven element

- 3 scenario's
  - Als het element == Root
    - Floor = root
  - Als element < root
    - Floor zit in linkse kind-boom
  - Als element > root
    - Floor kan in rechtse kind-boom
      (Als er een waarde bestaat <= dan het element in de rechter kind)
    - Floor is de root

Ceiling vinden in BST
- Kleinste element, groter dan of gelijk aan gegeven element

- Zelfde scenario's als Floor, omgekeerd

Rang vinden in BST
- Hoeveel elementen zijn kleiner dan een gegeven element?

1. If key == root:
   return size of left subtree
2. If key < root:
   return rank of key in left subtree
3. If key > root:
   return rank of key in right
   subtree + 1 + size of left subtree

Verwijderen uit BST

- Niet makkelijk, nog steeds onderzoek naar doen
- Verschillende aanpakken

Aanpak 1: Lazy Approach (Tombstone)

In plaats van element te verwijderen, een element markeren als "verwijderd"

Nadelen

- Niet overzichtelijk
- Er wordt geen geheugen gebruikt, komt telkens meer geheugen --> Vertraagt operaties

Aanpak 2: Verwijder minimum

Minimum in BST verwijderen

- Ga naar de knoop die zo links mogelijk staat
- Verwissel die knoop met zijn rechter kind
- Werk de counts van de boom bij

Aanpak 3: Verwijder bepaalde waarde

Bepaalde waarde verwijderen

- Verschillende gevallen mogelijk

- Geval 0: De knoop heeft geen kinderen
  - Knoop gewoon weglaten

- Geval 1: De knoop heeft 1 kind
  - Vervang de knoop met het kind

- Geval 2: De knoop heeft 2 kinderen
  - Vervang de knoop met de kleinste knoop uit de rechter kind-boom

Probleem: bij veel delete-operaties worden rechterbomen steeds kleiner en gaat boom zwaar naar links leunen

- Wordt nog onderzocht

implementation	guarantee			average case			ordered ops?
	search	insert	delete	search hit	insert	delete	
sequential search (unordered list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$	
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✔
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔

Binaire Zoekbomen --> Niet zo goed vanwege de onverwachte structuur, beter zijn de gebalanceerde bomen

Nagekeken en vervolledigd op 07-06-2023

Les 7 - Bomen

vrijdag 31 maart 2023 10:35

Liam Luys – r0934449 - 1e BA Informatica
Gegevensstructuren en Algoritmes

We willen de worst-case vermijden bij BST
- We willen de bomen evenwichtiger maken
- Een bovengrens opzetten voor de diepte van bomen
- Gebalanceerde Bomen

2-3 Bomen

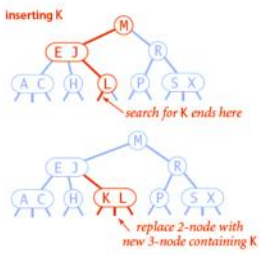
2-3 Trees
- Vorm van zoekbomen
- Is gebalanceerd
  - Elk pad van root naar onderste knoop is even lang
- Andere structuur
  - Knopen van 2 waardes toelaten (= *3-node/knoop)*
    - Met 3 vertakkingen: kleiner, tussen, groter
  - Knopen van 1 waarde toelaten (= *2-node/knoop)*
    - Met 2 vertakkingen: kleiner, groter

Zoeken in een 2-3 Boom
- Juiste vertakkingen volgen
- Kijken of op de juiste plaats het element aanwezig is

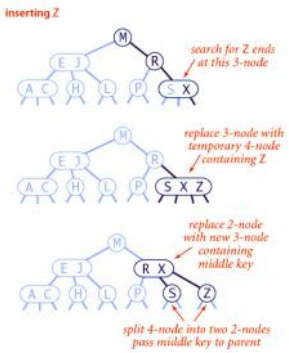
Hoe garantie van gebalanceerde eigenschap?
- Goed toevoegen van elementen

Elementen toevoegen aan 2-3 Boom
- Case 1: Waarde toevoegen aan een 2-knoop
  - 3-knoop van maken
- Case 2: Waarde toevoegen aan een 3-knoop met ouder een 2-knoop
  - Middelste element van de 3-knoop naar boven trekken (bij de 2-knoop), vertakkingen maken bij de originele 3-knoop
- Case 3: Waarde toevoegen aan een 3-knoop met ouder een 3-knoop
  - Waardes doorschuiven en vertakkingen maken van de knopen (zie figuur)

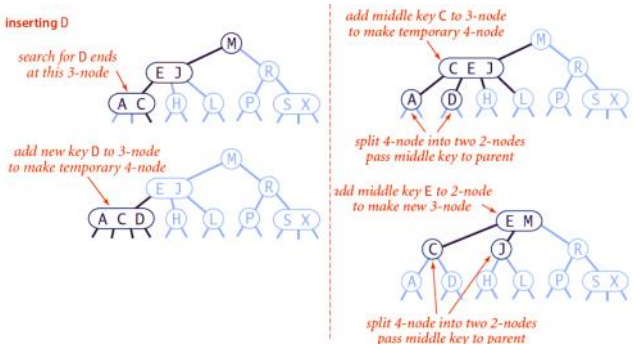
CASE 1



CASE 2



CASE 3



De boom groeit als de root een 3-knoop wordt

Balans moet behouden blijven
- Diepte overal gelijk

- Symmetrische orde (links kleiner, midden, rechts groter)

Wat is de diepte van de boom?
- Varieert naar 2-knopen/3-knopen

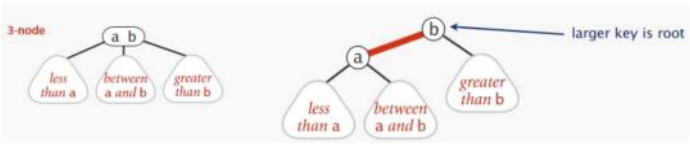
- (Worst) Indien enkel 2-knopen --> $\log_2 N \rightarrow bovengrens\ diepte\ 2\ 3\ bomen$
- (Best) Indien enkel 3-knopen --> $\log_3 N = \log_3 2 * \log_2 N = 0.631 \log_2 N \rightarrow minimale\ diepte\ 2\ 3\ boom$

- Gemiddeld --> Tussen 0.631 log2N en log2N

implementation	guarantee			average case			ordered ops?
	search	insert	delete	search hit	insert	delete	
sequential search (unordered list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$	
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✔
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔
2-3 tree	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	✔

Rood-Zwart Bomen / RB Trees

Hoe kunnen we de 2-3 bomen voorstellen als binaire boom (met twee vertakkingen)?
- Implementatie van rood-zwart bomen
- Veel varianten
    - Links-leunende Rood Zwart boom



Regels van Rood-Zwart Bomen
- Geen enkele knoop mag 2 rode connecties hebben
- Elk pad van de root tot de bodem heeft dezelfde aantal zwarte knopen
- Rode knopen leunen links (in deze variant)

- Maak de vergelijking met de 2-3 bomen!

Operaties op RB-Trees
- Zoeken Operatie
- Linkse rotatie
- Rechtse Rotatie
- Color Flip

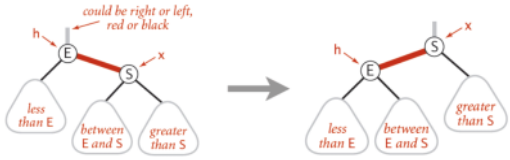
★ Operaties van buiten kennen!

Zoeken in Rood-Zwart Boom

- Zelfde als in BST
- Vertakkingen afgaan tot element gevonden/tot de bodem

Linkse Rotatie

- Stel de rode tak is aan rechterkant van knoop --> Linkse rotatie nodig

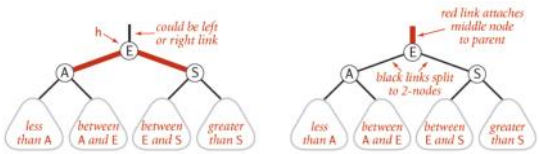


Rechtse Rotatie

- Stel de rode link wijst naar links maar je wil naar rechts --> Rechtse rotatie nodig

Color Flip

- Stel beide takken van een knoop zijn rood
- Middelste waarde naar boven trekken --> Kleuren van takken rond knoop omwisselen



Insert in Rood-Zwart Boom

- Telkens nieuwe waarde met rode link toevoegen
- Indien je ze rechts toevoegt ook een rotatie invoeren
- Indien de parent na toevoeging 2 gekleurde links heeft ook een color flip
- Indien 2 opeenvolgende rode links eerst rotatie, dan color flip

Zie slides en cursus voor visualisatie

Complexiteit

- Hoogte met n knopen begrensd door $2 \log_2 N$
  - Telkens even veel zwarten, tussen zwarte potentieel een rode --> mogelijk dat het rood-zwart-rood-zwart-… ($2 \log_2 N$)

implementation	guarantee			average case			ordered ops?
	search	insert	delete	search hit	insert	delete	
sequential search (unordered list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$	
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✔
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔
2-3 tree	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	✔
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N^*$	$1.0 \lg N^*$	$1.0 \lg N^*$	✔

Tries

Hoe strings organiseren in zoekbomen? Strings per karakter bekijken in plaats van geheel?

Werkwijze Trie
- Strings per karakter in de boom
- Elke knoop stelt een karakter voor
- Vanaf de knoop een pad naar een eindknoop stelt een string voor

Zoeken in een Trie
- Karakter afgaan en kijken of er een eindknoop is op de string die je zoekt

Toevoegen in een Trie
- Standaard, zie slides/demo
- Vertakkingen toevoegen zodat je string in de boom komt te staan

Complexiteit
- **<u>Structuur van de Trie hangt niet af van de volgorde van de invoer</u>**

- Maximaal (1+lengte van string die je zoekt) nakijken om een string te zoeken

- Search hit: alle karakters aflopen van de string
- Search miss
  - Hangt af hoeveel gelijkaardige strings in de boom zitten
  - Gemiddeld voor n random strings: $\log_R N$ (R = aantal mogelijke karakters, N = aantal strings)

Toepassingen
- Autocomplete Google
- …

Ternary Search Tries (TST)

Verwant aan 3-way string QuickSort

Opbouw TST
- 1 Root karakter met 3 vertakkingen
  - Links --> Iets dat kleiner is dan de parent
  - Middelste link --> Iets dat met de parent zelf begint

    - Rechtste link --> Iets dat groter is dan de parent

- Boom is qua breedte smaller, maar heeft grotere diepte
    - Is een balans in alle boomstructuren!

Complexiteit

- Totale links tussen 3N en 3Nw (w de gemiddelde lengte van een string)
- Search hit
    - $\sim L + \ln(N)$ $vergelijkingen\ van\ karakters$

- Search Miss
    - $\sim \ln(N)$ $vergelijkingen\ van\ karakters$

Hybrid Trie - TST

Combinatie van gewone trie met een Ternary Search Trie

Kan beter zijn voor bepaalde toepassingen

Keuze van datastructuur hangt af van de data zelf!

Nagekeken en vervolledigd op 07-06-2023

Les 8 - Stacks / Hashtabel

Liam Luys – r0934449 - 1e BA Informatica
Gegevensstructuren en Algoritmes

KU LEUVEN

vrijdag 21 april 2023 10:29

Stacks & Queues

Stack
- Last in First out (LIFO)
- Stapel van munten: bovenste munt eerst afhalen

Queue
- First in First out (FIFO)
- Wachtrij

Stack Implementeren
- Array - Implementatie
  - Objecten in een array
- Linked List - Implementatie
  - Objecten gelinkt aan elkaar

Gelinkte Lijst implementatie

Operaties op stack
- Pop()
  - Laatste Item van de stapel halen
  - Constante tijd
- Push()
  - Item toevoegen op de stapel
  - Constante tijd

Operaties gebeuren in constante tijd, hangt niet af van grootte van de stapel

Array implementatie

Operaties op stack *(n = lengte lijst)*
- Pop()
  - Verwijder item op plaats S[n-1]
- Push()
  - Voeg item toe op plaats S[n]

Nadelen array
- Niet dynamisch, je moet op voorhand grootte kennen
- Wat als de array vol zit?

Voordelen
- Geheugen wordt efficiënter gebruikt

Wat als array vol zit?
- Nieuwe grotere array maken
  - Niet te groot
- Elementen kopiëren uit oude array naar nieuwe array

Wat als array bijna leeg is?
- Terug kleiner maken
  - Niet te klein

- Elementen kopiëren uit oude array naar nieuwe array

Optie 1: Array vergroten/verkleinen met 1 element

Telkens array vergroten met 1 element als vol zit, verkleinen met 1 element wanneer een element weggaat

Niet meer in constante tijd! Hoe groter de array, hoe meer werk.

N² complexiteit voor array accesses

- $N + 2\big(1 + 2 + 3 + 4 + 5 + \cdots + (N-1)\big) = 2N\ \frac{N(N-1)}{2} = N^2 - N + N = \sim N^2$
  - $N \rightarrow Kost\ voor\ elke\ nieuwe\ push$
  - $2 \rightarrow Elk\ element\ eerst\ ophalen\ uit\ oude, dan\ terug\ kopieren\ in\ nieuwe\ lijst$
  - $(1 + 2 + \cdots) \rightarrow$ Kost van uitbreiding van k naar $k + 1$ elementen
    $\big(1 + 2 + 3 + \cdots + (N-1)\big) = \frac{N(N-1)}{2}$

- Geamortiseerde / Gemiddelde kost per toevoeging element?
  - $\frac{N^2}{N} = N\ array\ accesses$
  - Dit is geamortiseerde kost in lineaire tijd!

Optie 2: Array verdubbelen / Halveren in lengte

Verdubbelen als array vol zit, halveren als array voor 1/4 vol zit

Alles wat verdubbeld en halveert leidt vaak tot beter gedrag (denk aan recursie-algoritmes)

3N complexiteit voor array accesses

- $N + 2\left(1 + 2 + 4 + 8 + 16 + \cdots + \frac{N}{2}\right) = N + 2(N-1) = \sim 3N$
  - $N \rightarrow Kost\ voor\ elke\ nieuwe\ push$
  - $2 \rightarrow Elk\ element\ eerst\ ophalen\ uit\ oude, dan\ terug\ kopieren\ in\ nieuwe\ lijst$
  - $(1 + 2 + 4 + \cdots) \rightarrow$ Kost van uitbreiding van k naar 2k elementen
    $\left(1 + 2 + 4 + \cdots + \left(\frac{N}{2}\right)\right) = (N-1)$

- Geamortiseerde / Gemiddeld kost per toevoeging element?
  - $\frac{3N}{N} = 3\ array\ accesses$
  - Sommige accesses heel duur (wanneer rij verdubbeld wordt), andere veel goedkoper
  - Dit is een geamortiseerde constante tijd!

Halveren van array

- Als bezetting van array $\frac{1}{4}\ is$
- Als je het bij een halfvolle stack zou doen, heb je na halveren een volle array
  - Kans op hoge kosten als je daarop elementen toevoegt/weghaalt

Geamortiseerde Analyse

Bekijken van gemiddeld gedrag per operatie

Complexiteit

- Over een aantal mogelijke operaties/instructies
- Gaat niet over de invoer-grootte!

- Je zoekt een gemiddelde kost per operatie
    - Er zijn heel goedkope operaties
    - Er zijn enkele zeer dure operaties

Queue

Implementaties

- Linked List
    - Zowel een first-pointer en last-pointer bijhouden

- Array
    - Bijhouden van tail en head die begin en einde van queue bijhouden

Hash Tables

Werking Hash Tabel

- Alle data bijhouden in een array
- Door een hash functie bepalen welk element op welke plaats komt te staan

Hash Functie

Hash Functie

- Op basis van sleutel een plaats berekenen
- Map tussen de sleutels en het (beperkt) aantal plaatsen in array
- Er zijn vaak meer elementen dan plaatsen in de tabel
- Hash-functie geeft voor verschillende elementen dezelfde hash-waarde --> collisions.

Verwachtingen Hash Functie

- Gemakkelijk te berekenen
- Uniforme verdeling in bereik van array
    - Vermijden van collisions

- Zelfde sleutel moet zelfde hash waarde teruggeven
    - Zo kunnen we zoeken naar waardes
    - Nooit een random element in hash functie

Hash Functie vs Hash Code

- Hash Code --> Genereert van bepaald object een unieke code
- Hash Functie --> Codering van de Hash Code in een bepaald bereik

Opzet Hash Functie

- Modulaire Hashing
    - Gebruik van de modulo-operatie op de sleutel van het element
    - Als deler vaak een priemgetal
        - Zo wordt de sleutel vaak volledig gebruikt
        - Wordt verdeling over mogelijke plaatsen evenwichtiger

- Wordt vaak in meerdere delen gedaan
    - Meerdere keren modulo hashing op de sleutel van het element

Collisions

Collision
- Twee sleutels gaan naar dezelfde plaats in tabel
- Hoe oplossen?
  - Separate Chaining
  - Linear Probing

Separate Chaining

Separate Chaining
- Per array element een gelinkte lijst bijhouden om elementen op dezelfde plaats te kunnen bijhouden
- Gemakkelijke oplossing
- Insert is gemakkelijke en korte operatie
- Search operatie kan lang duren (max de lengte van langst gelinkte lijst)

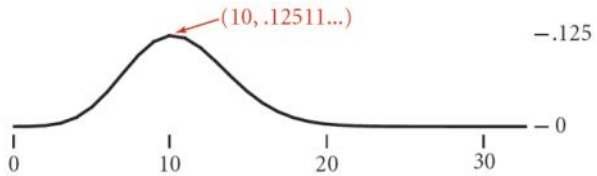
Evenwicht nodig tussen lengte van array en max lengte van gelinkte lijst
- Grote hashtabel --> Kleinere gelinkte lijsten (maar ook veel ongebruikte ruimte)
- Kleine hashtabel --> Grotere gelinkte lijsten (zoeken wordt moeilijker)

- Hashtabel vergroten als de gelinkte lijsten te lang worden.

Analyse Separate Chaining
- Hangt af van de lengte van gelinkte lijsten
- Gemiddelde lengte van de lijst $\alpha = \frac{N}{M}$
  - $N \rightarrow aantal\ elementen$
  - $M \rightarrow aantal\ array\ plaatsen$

- Wat is de kans dat k sleutels naar dezelfde hash-waarde gaan mappen <u>in de veronderstelling dat er een uniforme verdeling</u> is van elementen in de array?
  - Kans dat één k op een bepaalde waarde wordt gehashed → $\frac{1}{M}$
  - Voor alle k's, waarbij de andere sleutels op andere plaatsen terechtkomen:
    - $C_N^k \left(\frac{1}{M}\right)^k \left(\frac{M-1}{M}\right)^{N-k} \quad \rightarrow \quad Binomiale\ verdeling$

      Combinatie C_N^k omdat het voor elke mogelijke plaats in array kan gelden



Binomial distribution $(N = 10^4, M = 10^3, \alpha = 10)$

- Gevolgen
  - De uitzonderlijke gevallen dat de gelinkte lijsten lang ($> 2\alpha$) worden komen niet vaak voor! De kans op zo'n verdeling is zeer klein

  - Aantal probes voor een search (miss) is $\sim\alpha = \sim\frac{N}{M}$
  - Aantal probes voor een insert is $\sim\alpha = \sim\frac{N}{M}$

Hoe groot Hash Tabel kiezen?

- Te groot --> te veel lege array-plaatsen
- Te klein --> te lange gelinkte lijsten
- Gemiddeld --> $\alpha = \frac{N}{M} \quad \Rightarrow \quad Constante\ tijd\ operaties$

Resizing Hash Tabel

- Wanneer de lijsten te lang worden (Load-factor α te groot)
- Elk element opnieuw hashen in een nieuwe grotere array

- Doel: $\alpha\ constant\ houden$

Linear Probing

Linear Probing

- Geen gebruik van gelinkte lijsten
- Zit er al iets op de plaats waar je wil plaatsen? Schuif plaats op tot waar er wel plaats is.
- Insert gemakkelijk
- Zoeken is moeilijker, omdat sleutel niet altijd op de hash-waarde zit

Zoeken in Linear Probing

- Bereken hash-waarde
- Kijk op die plaats in array
    - Lege plaats?
        - Element zit niet in array
    - Element zit op hash-waarde?
        - Element gevonden
    - Ander element op hash-waarde?
        - Kijk op de plaats erlangs in de array tot je het element of een lege plaats tegenkomt

Deleten in Hash Tabel

- Wordt zeer moeilijk
- Als je een lege plaats creëert zal zoekfunctie niet meer correct werken

- 2 Mogelijke oplossingen
    - Veld markeren "marked as deleted" zodat zoekfunctie weet dat die verder mag gaan
    - Alles rehashen in een andere Hash Tabel

Analyse Linear Probing

- Hangt af van de lengte van het aantal opeenvolgend gevulde plaatsen in array
    - Hoe korter, hoe beter
    - Langere opeenvolgingen worden snel nog langer

- Analyse zelf niet kunnen afleiden, zie boek propositie M

Resizing Hash Tabel

- Wanneer de lijsten te lang worden (Load-factor α te groot)
- Elk element opnieuw hashen in een nieuwe grotere array

- Doel: $\alpha \leq \frac{1}{2}$

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$		equals()
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✔	compareTo()
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔	compareTo()
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$	$1.0 \lg N$	$1.0 \lg N$	✔	compareTo()
separate chaining	N	N	N	3-5 *	3-5 *	3-5 *		equals() hashCode()
linear probing	N	N	N	3-5 *	3-5 *	3-5 *		equals() hashCode()

* under uniform hashing assumption

Andere Hash Methodes

- Two-Probe Hashing (Chaining)
  - 2 hash-waarde berekenen (met verschillende hash-functies), in de waarde met kortste lijst het element zetten
  - Terugvinden van element? Kijken naar 2 hash-waardes.

- Double Hashing (Lineair Probing)
  - Hash berekenen op basis van sleutel en het aantal pogingen om het element in de lijst te voegen
  - Vermijden van collisions

- Cockoo Hashing
  - Zit er al iets op de hash-waarde? Ga er zelf in zitten en laat het oude element zelf een nieuwe plaats zoeken.

Toepassingen

- DOS Attacks op systemen (gebruik maken van slechte implementatie hash tabel)
- Verkeer, parking

Nagekeken en vervolledigd op 13-06-2023

Les 9 - Dynamic Programming

Liam Luys – r0934449 - 1e BA Informatica
Gegevensstructuren en Algoritmes
KU LEUVEN

vrijdag 28 april 2023 10:34

Dynamic Programming

Alles in slides, niet in het boek.

Voorbeeld: Fibonacci
- Bij de 'eenvoudige' implementatie
- Exponentieel verloop --> Zeer slecht!
- Dit komt omdat je telkens terug de oproepen doet voor vorige Fibonacci-getallen

Dynamic Programming
- Probleem oplossen door deelproblemen op te lossen
- Tussenresultaten deelproblemen bijhouden om later terug te kunnen gebruiken in echt probleem

- Verschillende Aanpakken
  - Top-down (memorisatie)
    - Rekentijd ruilen voor geheugen
    - Fibonacci getallen bijhouden eens je ze berekend hebt
  - Bottom-Up (Dynamic Programming)
    - Fibonacci getallen berekenen vanaf 0 tot hetgeen dat je zoekt
- Vaak uit recursieve relaties die elkaar overlappen

Voorbeelden Dynamic Programming

Werkwijze
- Probleem opdelen in subproblemen (Recursieve definitie)
- Uitkomsten van subproblemen opslaan om te herbruiken
- Door subproblemen op te lossen het volledige probleem oplossen

Voorbeelden zeker bekijken en snappen. Lesopnames/slides

Coins in a line

Zie slides

Longest Common Subsequence LCS

Probleem: Wat is de langste gemeenschappelijke subsequentie van twee strings?

Subsequentie String
- Niet zelfde als een substring!
- Gemeenschappelijke tekens maar niet perse opeenvolgend

Naïeve Oplossing
- Alle mogelijke subsequenties bekijken in String A (lengte n) $(2^n\ mogelijkheden)$
- Kijken of die subsequenties ook in String B (lengte m) zitten $(\sim m\ tijd)$

- $\sim 2^n * m \quad \rightarrow Exponentieel\ algoritme \quad = \quad SLECHT$

Oplossing met Dynamic Programming

- Optimale structuur zoeken
  - Probleem uitdrukken als oplossing van subproblemen

- Subprobleem
  - We zoeken langst gemeenschappelijke subsequentie in interval van i tot j
  - Hoe drukken we dat uit als combinatie van subproblemen?
    Zie slides

Optimale Binaire Zoekbomen

Zie slides

Gebruik Dynamic Programming

Wanneer gebruik je Dynamic Programming?

1. Optimale Substructuur

- Optimale Substructuur
  - Optimale oplossing bestaat uit de optimale oplossing van subproblemen

- Variaties
  - Hoeveel subproblemen voor de optimale oplossing?
  - Hoeveel keuzes maken om te beslissen welk subprobleem?

	Coin Problem	**LCS**	**Optimal BST**
#Subproblemen	2	1 of 2	2
#keuzes	2	?	n

- Als de oplossingen van deelproblemen onafhankelijk zijn van elkaar

2. Overlappende Subproblemen

- Recursieve oplossing die meerdere keren dezelfde deelproblemen gebruikt (Fibonacci)
  - Eén keer uitrekenen en hergebruiken

- Grafenprobleem

Rod Cutting Problem

Zie slides

Dynamic Programming

- Heeft optimale substructuur nodig
  - Probleem opdelen in deelproblemen die je gemakkelijker kan oplossen en combineren tot globaal probleem

- Overlappende Deelproblemen
  - Oplossingen voor deelproblemen die terugkomen bijhouden *(Fibonacci)*

- Voorgesteld door subprobleem-grafe
  - Boom die weergeven op welke manier de opgeslagen data wordt gebruikt
  - Bij Dynamic Programming -> Bottom Up

Nagekeken en vervolledigd op 13-06-2023

Les 10 - Greedy

vrijdag 5 mei 2023 10:34

Liam Luys – r0934449 - 1e BA Informatica
Gegevensstructuren en Algoritmes

KU LEUVEN

Greedy Algorithms

Werking
- Om oplossing te vinden de keuze maakt die op dat moment het beste lijkt
- Van het probleem telkens een stukje afnemen zodat het kleiner wordt

Verschil Dynamic Programming
- Bij DP alle mogelijkheden afgaan en kijken welke beste is
- Bij Greedy één mogelijkheid zoeken

- Greedy maakt probleem steeds kleiner, Dynamic Programming begint van klein probleem en maakt het groter

Problemen
- Is er een oplossing mogelijk door een greedy algoritme?
- Is de bekomen oplossing de best mogelijke oplossing?

Wat heb je nodig?
- Manier om te beslissen wat de best mogelijke stap is
- Waarnemen wanneer je een oplossing hebt bekomen
- Bewijzen dat het een optimale oplossing is

Coin Exchange Problem

Hoe munten kiezen om bedrag te betalen?
- Is een probleem dat met Greedy Algoritme kan opgelost worden
- Telkens grootst mogelijke briefje/munt nemen om zo tot volledig bedrag te komen

- Werkt bij e Euro, maar niet perse voor elk coin exchange probleem!

Kunnen bewijzen dat je een optimale oplossing bekomt

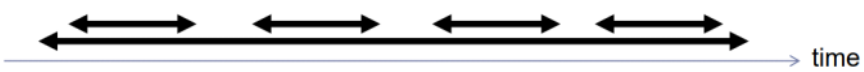
Malfatti Problem

Malfatti Problem
- Hoe 3 cirkels in een driehoek met samen de grootste oppervlakte?
- Lukt met Greedy Algoritme
    - Telkens de grootst mogelijke cirkel in de driehoek plaatsen

Activity Sheduling

Hoe zo veel mogelijk niet-overlappende activiteiten plannen?
- Recursief oplossen is niet slim
- Oplossen op een greedy manier

Mogelijkheid 1
- Activiteit nemen die zo vroeg mogelijk start
- Dit is geen slimme oplossing

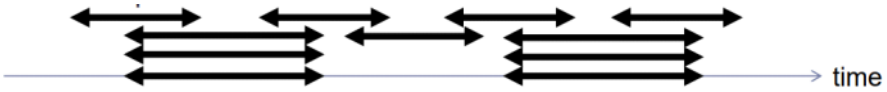


Mogelijkheid 2

- Kortste activiteit nemen die nog niet conflicteert met andere activiteiten
- Dit is geen slimme oplossing

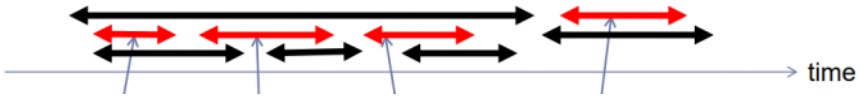
Mogelijkheid 3

- Activiteit nemen met minst aantal conflicten
- Dit is geen slimme oplossing



Mogelijkheid 4

- Activiteit die het vroegste eindigt en niet conflicteert met eerdere gekozen activiteiten
- Dit is de optimale oplossing!



Bewijs door substitutie: Deze aanpak geeft een optimale oplossing

$A = \{a_1, a_2, a_3, \dots, a_n\}$ is een verzameling activiteiten gesorteerd op de eindtijd.
(Activiteiten overlappen niet, dus zijn ook gesorteerd op starttijd)

$B = \{b_1, b_2, b_3, \dots, b_m\}$ is een optimale verzameling van activiteiten

- $m \geq n \rightarrow$ want B is optimaal
- We willen aantonen dat $m = n$ en dus A ook optimaal is

Neem aan dat $\boldsymbol{a_1} ! = \boldsymbol{b_1}$

- We weten dat a_1 als eerste eindigt, dus zal a_1 eindigen voor b_1
- Hierdoor zal a_1 niet conflicteren met b_2, omdat b_2 start na b_1, en a_1 voor b_1 klaar is
- Hierdoor kunnen we b_1 vervangen door a_1 en behouden een optimale set

Substitutie

- We kunnen met die redenering alle activiteiten b_i van B verplaatsen met a_i van A
- Dus is $\{a_1, a_2, a_3, \dots, a_n, b_{n+1}, \dots, b_m\}$ een optimale oplossing

Door definitie van A en B weten we dat m gelijk moet zijn aan n

- Hierdoor wordt A een optimale oplossing voor het probleem

- Verschil Dynamic Programming en greedy? Voorbeelden gebruiken!
- Zou je bepaald algoritme greedy noemen? Selection Sort? Dijkstra? Is dat Greedy of Dynamic, of nog iets anders (domme recursie)?

Compressie Data

Compressie van Data

- Een gecomprimeerde versie genereren van bepaalde data
- Zonder dataverlies

- Kan gedecomprimeerd worden
- Compressie ratio: $\frac{Aantal\ bits\ in\ gecomprimeerde\ data}{Aantal\ bits\ in\ originele\ data}$

Compressie Algoritmes

Compressie Algoritmes
- Run-Length Encoding
- Variable-Length Coding

Run-Length Encoding
- 4-bit lijst bijhouden waarin je het aantal nullen en enen voorstelt door 4 bits
- Eerste 4 bits zijn aantal nullen, tweede 4 bits het aantal enen, derde aantal nullen, enz…

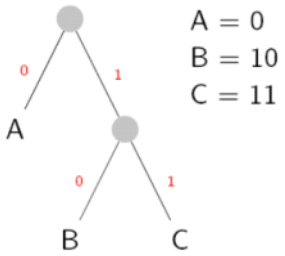
Variable Length Coding
- Tekens die vaker voorkomen hebben een kortere code
- Voorbeelden
  - UTF-8
  - Morsecode

- Hoe onderscheid maken tussen waar karakter eindigt en het volgende begint?
  - Bij morse -> Spatie laten tussen twee karakters
  - In computertaal -> prefix-vrije code

Fixed Coding
- Elk karakter een specifieke code geven
- Is vaak minder optimaal dan variable lenght Coding

Huffman Coding

Huffman Coding
- Data comprimeren met zo weinig mogelijk bits
- Karakters die vaker voorkomen een kortere code geven
- Hangt af van de tekst/woord dat je wil coderen
- Werkt met prefix-vrije code
  - Geen enkele code is prefix van een andere code
  - Kan je voorstellen door een boom



A = 0
B = 10
C = 11

A B C C A B B A
0 10 11 11 0 10 10 0

Opstellen Huffman codeboom
- Tellen hoe vaak een letter voorkomt
- De minst-frequente elementen bij elkaar nemen en een gezamenlijke ouder-knoop aan koppelen
- Telkens de minst frequente groepen/elementen samennemen tot je een volledige boom hebt
- In de boom de takken benoemen met '0' en '1' of andere symbolen

Bewijs van Huffman Codering

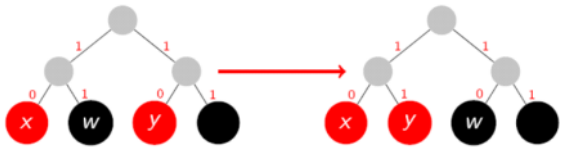
Bewijzen dat de codeboom de optimale boom is

Lemma 1: Elke optimale codeboom is een volledige boom (ouder heeft altijd 2 kinderen)

Lemma 2: Er bestaat een optimale boom waarbij de twee minst-voorkomende karakters broer en zus zijn op de maximale diepte

Bewijs

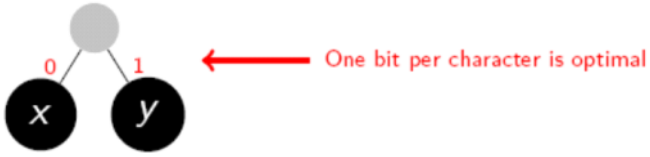
- Elk blad heeft een broer (Lemma 1)
- Twee minst-voorkomende symbolen moeten op maximale diepte in de boom zitten
  Zie slides
- Als de minst-voorkomende symbolen geen broer/zus zijn, verwissel ze



Bewijs door inductie

- Basis: Bewijs dat $W(T_H)$ optimaal is voor 2 symbolen
- Inductie: Als T_H optimaal is voor $R-1$ symbolen, is T_H optimaal voor R symbolen

Basisstap



Inductiestap

- Neem aan dat T_H optimaal is voor $R-1$

$T_{H,R}$ → Huffman boom voor R symbolen $(s_1, f_1), \dots, (s_R, f_R)$

s_i en s_j zijn eerste twee symbolen met laagste frequentie

- We vervangen ze door symbolen $(s^*, f_i + f_j)$ en hebben nog $R-1$ symbolen over

Huffman Algoritme berekend dan $T_{H,R-1}$:

$W(T_{H,R}) = W(T_{H,R-1}) + (f_i + f_j)$ (**)

Dit is verband tussen Huffman boom en Huffman boom van 1 karakter minder

Beschouw de optimale boom $T_{opt,R}$ voor $(s_1, f_1) \dots (s_R, F_R)$

- s_i en s_j moeten op het diepste level zitten en zijn broer/zus (Lemma 2)
- Maak een nieuwe boom T_{R-1} door het samenvoegen van s_i en s_j in een ouder $(s^*, f_i + f_j)$
- $W(T_{opt,R}) = W(T_{R-1}) + (f_i + f_j)$ (α)

Inductie hypothese

- $T_{H,R-1}$ is optimaal voor $R-1$ symbolen
- $W(T_{H,R-1}) \leq W(T_{R-1})$ (β)

$W(T_{H,R}) = W(T_{H,R-1}) + (f_i + f_j)$ (**)

$\beta \Rightarrow \; W(T_{H,R}) \leq W(T_{H,R-1}) + (f_i + f_j)$

$\alpha \Rightarrow \; W(T_{H,R}) \leq W(T_{opt,R})$

Besluit: $W(T_{H,R}) = W(T_{opt})$ dus is $T_{H,R}$ de meest optimale boom voor R symbolen

***Q*.*E*.*D*.**

Nagekeken en vervolledigd op 14-06-2023

Les 11 - Substring

vrijdag 12 mei 2023 10:33

Liam Luys – r0934449 - 1e BA Informatica
Gegevensstructuren en Algoritmes

KU LEUVEN

Substring Search

Substring Search
- Willekeurige patroon van karakters vinden in een andere (grotere) sequentie van karakters
- Lente van patroon = M, lengte van tekst = N, logisch geldt dat M << N

Voorbeelden gebruik
- Zoekmachines
- Spam filters
- Zoeken in tekst, …
- Page Scraping
- Beeldherkenning

Uitdaging
- We willen patronen herkennen in stromen data die we niet kunnen opslaan

Wild card
- Iets variabel wat begint met bepaalde karakter(s) en eindigt met bepaalde karakter(s)
- Niet belangrijk in deze cursus

Algoritmes Substring Search

Algoritmes voor Substring Search
- Brute Force
- Knuth-Morris-Pratt KMP
- Boyer-Moore
- Rabin-Karp

Brute Force

Op alle mogelijke posities in karakters kijken of substring voorkomt

Worst worst Case
- $\sim M * N$
- Als je op elke startpositie telkens een mismatch hebt, en toch heel de string doorloopt om te testen

Best Case
- Substring gevonden op eerste startpositie

Teruglopend algoritme
- Door bekijken van elke mogelijke beginpositie loop je terug in de sequentie om te testen
- Als we deze back-up vermijden kunnen we betere complexiteit hebben!

Niet ideaal, we willen een lineair algoritme en nooit teruggaan in de tekst

Knuth-Morris-Pratt

Probleem voorstellen als deterministische eindige automaat

Nummer toestand = Aantal gematchte karakters

Telkens inlezen van karakter en veranderen van toestand

Zie slides

Tijdscomplexiteit voor maken automaat?
- Tabel invullen die toestanden koppelt aan hoeveelheid verschillende karakters
- $\sim R * M$
  - R aantal verschillende tekens in substring
  - M lengte van de substring

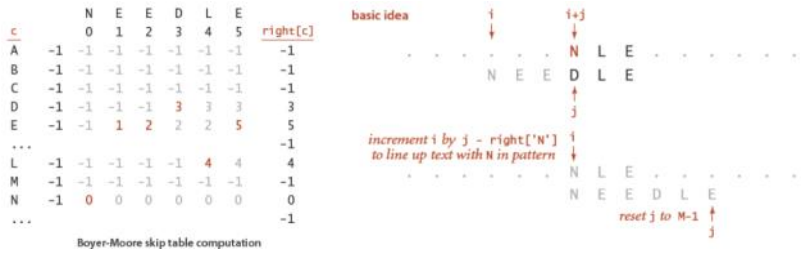
Examen
- Tabel verder aanvullen, koppelen aan string
- Construeer toestandsmachine voor een gegeven string

Boyer-Moore

Patronen zoeken van achter naar voren in de tekst

Werking Algoritme
- Bekijk karakters van rechts naar links
- Je kan M karakters overslaan als je een karakter vindt dat niet in het patroon staat
- Je kan < M karakters overslaan als je een karakter vind dat wel in het patroon staat, maar op een andere locatie
  - Zodra je een karakter vindt, doorschuiven zodat de posities overeenkomen, volgende vergelijken

- Skiptabel bijhouden waarin staat hoeveel je mag verschuiven als je bepaald karakter tegenkomt van het patroon
  - Telkens de meest rechtse positie van het karakter in het patroon



Boyer-Moore skip table computation

Tijdscomplexiteit
- Verschilt sterk, we raken niet elk karakter aan in de tekst

- Best Case
  - $\sim N/M$
  - Als je telkens volledig patroon doorschuift en mismatch hebt op de laatste elementen die eerst vergeleken worden

- Worst Case
  - $\sim N * M$ (Brute force)
  - Als je telkens met 1 positie kan verschuiven in de tekst

Rabin-Karp

Substring zoeken op basis van berekende (modulaire) hash-code van de substring
In de tekst zoek je de hash die overeenkomt met die van de substring, indien overeenkomt nog elementen zelf nakijken

Werking
- Belangrijkste hier is over de manier waarop de modulaire hash wordt berekend
- Gemakkelijkste om ze te berekenen adhv de vorige hash, omdat je steeds 1 plaats opschuift
  - Kan d.m.v. rekenregels van modulo, zie slides

Waarschijnlijk geen examenvraag over implementatie

Zeer vaak zijn er alternatieve implementaties of combinaties van de geziene algoritmes

Varianten

- Las Vegas Variant
    - Als je een hash gevonden hebt, controleer je nog eens de string
    - Steeds 100% correct
    - Bijna in Lineaire Tijd

- Monte Carlo Variant
    - Als je een hash gevonden hebt ga je er vanuit dat de strings overeenkomen
    - Niet altijd 100% correct
    - Steeds in Lineaire Tijd

Zeer vaak zijn er alternatieve implementaties of combinaties van de geziene algoritmes

Examen

Volledige leerstof bedekt, zowel oefeningen als theorie

Geef en bewijs tijdscomplexiteit van sorteeralgoritme …

Nagekeken en vervolledigd op 14-06-2023