Examen Algoritmen en Datastructuren 3

Naam : ....

- Lees de hele oefening zorgvuldig voordat je begint ze op te lossen!
  Als je niet goed verstaat wat de vraag of taak is, vraag het aan de lesgever! Voor oefeningen die fout verstaan zijn, kunnen geen punten gegeven worden!
- Schrijf leesbaar. Oplossingen die niet leesbaar zijn, kunnen ook niet beoordeeld worden.
- Als technieken toegepast moeten worden, toon altijd voldoende tussenstappen om te kunnen zien wat er gebeurt en dat de technieken goed verstaan zijn.
- Stellingen uit de les mogen natuurlijk altijd gebruikt worden zonder dat het bewijs opnieuw gegeven moet worden (behalve in gevallen waar het expliciet anders staat)!
- Geef alleen dan een antwoord als je denkt dat je de oplossing kent. Verspil geen tijd met de poging gewoon lange teksten te schrijven waarin zekere sleutelwoorden opduiken – zoals dat vaak geprobeerd wordt. Dergelijke oplossingen halen nooit punten en voor bijzonder slechte oplossingen worden punten afgetrokken!

1. **Tries (2.75 pt)**

   - Jouw alfabet is $\{a, c, g, t\}$. Geef de Patricia trie voor de woorden `acccct_`, `cacct_`, `accc_`, `ccagt_`, `ccatg_`,

- Geef een ternary trie met de woorden `ternary_`, `patricia_`, `test_`, `slagen_`, `paden_`, `sla_`.

- Je wilt lange Nederlandse teksten met ASCII tekens in een suffix boom opslaan. Je kan de pointers in de toppen in een array bijhouden of op de manier van een ternary trie. Waarvoor kies je – en waarom? Geef de voor- en nadelen van deze twee manieren voor deze toepassing.

- Stel dat je voor het alfabet met alle ASCII tekens een suffix boom op de manier van een ternary trie voorstelt. Is het algoritme van Ukkonen dan nog altijd $O(n)$?

2. **Compressie (3 pt)**

   - Stel dat het model voor arithmetisch coderen het alfabet (in deze volgorde) d, o,t gebruikt met $p(d) = 0.4$, $p(o) = 0.4$, en $p(t) = 0.2$. Welke string met 3 lettertekens wordt door de bitstring `0101` gecodeerd?

- Geef de statische Huffman codering van `toto_tour`.

- Decodeer de LZW code $98(=b)$, $97(=a)$, $110(=n)$, $97(=a)$, 259, 260, 257.

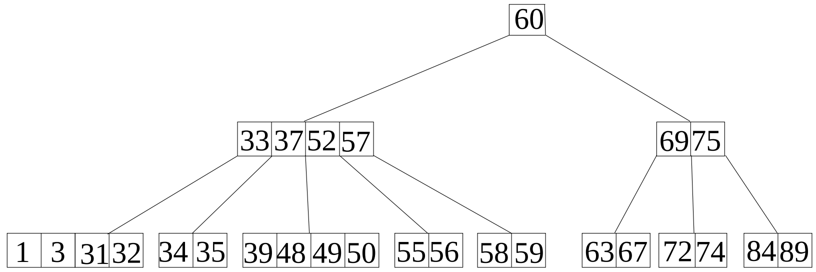
- Burrows en Wheeler stellen de move-to-front methode voor om van de getransformeerde tekst een tekst te maken die bijzonder goed Huffman codeerbaar is. Herhalingen hebben dan altijd de code 0 – en je hebt veel herhalingen. Maar je zou toch ook als volgt kunnen werken – laat ons dat de *relative position methode* noemen: je begint met de positie van het eerste teken en geeft dan altijd de relatieve positie van het volgende teken ten opzichte van het vorige teken, zonder tekens te verplaatsen.

  Voorbeeld: als de lijst `a,o,n,l` is en de string `loon` dan is de code $3, -2, 0, 1$.

  Ook hier geeft een herhaling dus een code 0. Denk je dat de relative position methode beter, even goed of slechter presteert dan move-to-front? Geef uitleg.

3. **Zoekbomen (2 pt)**

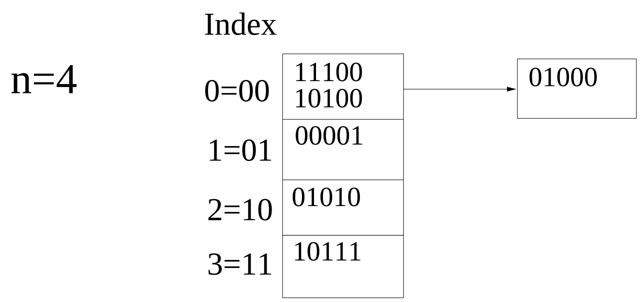
- Voeg sleutel 2 toe aan de volgende *B*-tree met grootte 4. In de tussenstappen is het voldoende de betrokken delen te tekenen – maar toon ook het eindresultaat.



- Wij hebben de grootte van een $B+$-boom altijd berekend door een zo groot mogelijk getal te kiezen dat de sleutels en de pointers nog in één blok passen. Daarbij hebben wij altijd verondersteld dat de pointers in de interne toppen (die naar andere toppen gaan) en in de bladeren (die naar records gaan) even veel bytes nodig hebben. Stel nu dat de pointers die naar records gaan bv. twee keer groter moeten zijn. Natuurlijk kan je dan gewoon met deze grootte rekening houden, maar dan krijg je een B+-boom T die een kleinere vertakking in de interne toppen heeft dan in principe mogelijk zou zijn. Definieer een gewijzigde $B+$-boom die het geheugen efficiënter gebruikt dan deze boom T, maar waar je nog altijd efficiënt sleutels kan toevoegen.

4. **Hashing (1.75 pt)**

- Voeg de record met sleutel `01001` toe aan de volgende linear hashing tabel die 2 sleutels per emmer mag bevatten en waar een maximale laadfactor van 0.8 is toegelaten. Natuurlijk mag je ook hier – net zoals in de les – alleen de hashwaarde toevoegen om te tonen hoe de tabel werkt.



- Extendible hashing werkt met de eerste a bits van de binaire voorstelling van de hashwaarde. Als een record r hashwaarde $h(r)$ heeft, wordt het dus in de emmer geplaatst waar de pointer op positie $h(r)|_a$ naar wijst. Dat is natuurlijk een detail en je zou even goed de laatste a bits kunnen gebruiken zoals bij linear hashing (daarvoor hebben wij $h(r)|^a$ geschreven). Beschrijf de dubbeloperatie van de pointerarray voor dit geval expliciet en werk ook een klein voorbeeld uit – bv. een voorbeeld waar door een toevoeging een pointerarray van lengte 2 naar lengte 4 uitgebreid moet worden.

5. **Bloom filters (1.25 pt)**

   Een bedrijf heeft een belangrijk dataset dat door veel mensen wordt gebruikt en waar buitenstaanders zeker geen toegang mogen hebben. Om toegang tot de dataset te krijgen, krijgt elke geautoriseerde medewerker een eigen password (van lengte ten hoogste 12). Omdat de toegekende passwoorden soms ook verstuurd moeten worden om op andere servers getoetst te kunnen worden, stelt iemand voor, geheugen te besparen en een bloomfilter te gebruiken. Geef voor elke van de drie volgende mogelijkheden expliciet antwoord of dat een goede keuze is en waarom het dat wel dan niet is.

   a.) Je slaat de passwoorden in een Bloom filter op en toetst of een gegeven password in de filter zit. Als ja, krijgt de gebruiker toegang.

   b.) Je slaat de strings met lengte ten hoogste 12 die geen passwoorden zijn in een Bloom filter op en toetst of een gegeven password in de filter zit. Als ja, wordt de toegang eerst geweigerd en wordt het (gecodeerde) password aan een centrale server doorgestuurd om daar door middel van een lijst getoetst te worden.

   c.) Je gebruikt beter geen Bloom filter.

6. **String matching (3.25 pt)**

- Zoek de tekst `punt` in de tekst `geen_gouden_punten` met het algoritme van Boyer-Moore-Horspool.

- Gegeven q en m zodat elke mogelijke hashwaarde (met de hashfunctie van Rabin-Karp) met meerdere bitstrings van lengte m correspondeert.

  Toon aan: Voor elke $n > m$ bestaat een zoekstring $z[]$ van lengte m en een tekst $t[]$ van lengte n die $z[]$ niet bevat, maar waarvoor `strings_gelijk()` meer dan $\frac{n}{m}$ keer opgeroepen wordt.

- Gegeven een tekst $t[]$ met lengte n en twee (kortere) strings $s_1[], s_2[]$ met lengtes n_1, n_2. Wij zoeken de twee posities in $t[]$, waar een kopie van $s_1[]$ zo dicht mogelijk bij een kopie van $s_2[]$ staat. Of precies

  Positie i is een startpositie van $s_1[]$ als voor $0 \leq k < n_1$ geldt $s_1[k] = t[i+k]$. Positie j is een startpositie van $s_2[]$ als voor $0 \leq k < n_2$ geldt $s_2[k] = t[i+k]$.

  Gezocht zijn i, j zodat i een startpositie van $s_1[]$ is, j een startpositie van $s_2[]$ is en $|i-j|$ minimaal is.

  Geef een $O(n)$ tijdsbegrensd algoritme dat dergelijke posities berekent als die bestaan. Een $O(n \log n)$ algoritme wordt ook aanvaard, maar haalt niet de volle punten.

7. **Benaderend string matching (1 pt)**

- Pas het shift-AND algoritme voor matches met ten hoogste één fout toe om alle matches met ten hoogste één fout van het woord `dodo` in de tekst `frodo_is_doof` te vinden.

8. **Parallelle algoritmen (1 pt)**

   Wij hebben een universum $U = \{0,\ldots,n\}$ en verzamelingen $V_0, V_1, \ldots, V_k$ die allemaal deelverzamelingen van U zijn, waarbig $V_0 \neq U$. De taak is nu uit te vissen of V_0 als snede van verzamelingen $V_1, \ldots, V_k$ voorgesteld kan worden en als ja wat het kleinste aantal verzamelingen is, waarmee dat mogelijk is. Je gebruikt een heel eenvoudig recursief algoritme dat je geparalleliseerd hebt en dat in de volgende pseudocode staat. De variabele *min* is globaal en dezelfde voor alle delen die je opstart. De variabele houdt bij hoe goed de tot nu toe beste oplossing is. Ga ervanuit dat *min* door elk deel gelezen en door middel van de functie *updatemin(x)* op $\min\{min, x\}$ gezet kan worden zonder dat er problemen met gelijktijdig lezen of schrijven zijn. De variabele *counter* is een globale integer variabele die voor alle delen verschilt en in het begin -1 is. Ga er ook vanuit dat de globale variabele *splitlevel* zo vastgelegd is dat een goede verdeling gegarandeerd is.

```
procedure MINVERZAMELING( i, V, aantal, mijndeel, mod)
    if (i > k)||(aantal ≥ (min − 1)) then
        return;
    end if
    if (i = splitlevel) then
        counter = counter + 1
        if (counter = mijndeel) then
            counter = counter − mod;
        else
            return;
        end if
    end if
    minverzameling( i+1, V, aantal, mijn_deel, mod);
    if V_0 ⊈ V_i then
        return;
    end if
    V = V ∩ V_i
    if V_0 = V then
        updatemin(aantal+1); return;
    end if
    minverzameling( i+1, V, aantal+1, mijn_deel, mod);
end procedure
```

   Het programma moet dan ogestart worden met $i = 1$, $V = U$, $min = \infty$ en $aantal = 0$ en als je het in *mod* delen wilt opstarten met de waarden $0, \ldots, mod - 1$ voor $mijndeel$. Als op het einde nog altijd $min = \infty$, bestaat er geen oplossing, anders is *min* het minimale aantal.

   Jammer genoeg is dit programma fout. . . Wat is er mis?

NOG NIET OMDRAAIEN !