# Tentamen
# ORIENTATIE INFORMATICA
## 9 november 2004
## 09.00 – 12.00

---

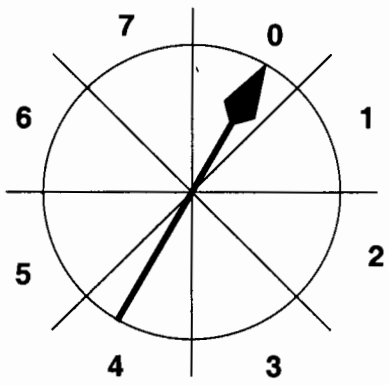
Opmerkingen vooraf:

- geef bij elke Haskell-functie ook de typering.
- in elk onderdeel mag je gebruik maken van vorige onderdelen, ook als je niet in staat bent geweest deze te maken.
- Niet elke opgave telt even zwaar. Bij elk onderdeel staat aangegeven hoeveel punten er maximaal gescoord kunnen worden. Bij 100 of meer punten is het tentamenresultaat een 10. In andere gevallen is het tentamenresultaat het aantal punten gedeeld door 10.
- Studenten die de 3ec-variant van dit vak willen doen, dienen dit expliciet bovenaan het eerste in te leveren blad te vermelden. Een score van 50 punten levert voor hen de beoordeling 10.
  De items 1 tot en met 14 horen bij deze versie van het vak, maar het is toegestaan ook andere items uit te werken om op die manier de score te verhogen.

---

## Opgave 1 (Gray-code)

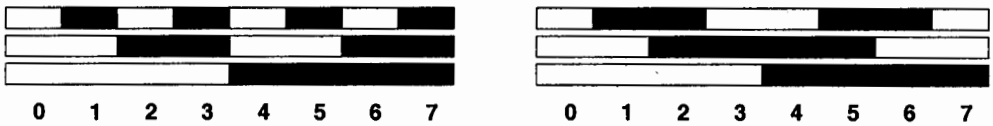
Bekijk de tekening hiernaast. Een cirkel is verdeeld in acht sectoren, genummerd van 0 tot en met 7. De pijl is draaibaar opgesteld. Je zou kunnen denken aan een apparaatje om de windrichting te bepalen of aan een spelletje. In het eerste geval geeft het nummer van de sector een aanduiding van de windrichting; in het tweede geval wordt het winnende nummer aangewezen.




Het doel is nu om de aangewezen sector electronisch te bepalen. De cirkel is daarvoor uitgevoerd als een opstaande rand bestaande uit drie stroken en onder de pijlpunt zitten drie contacten. De stroken zijn verdeeld in isolerende en geleidende gebieden. Komt een contact in aanraking met een geleidend gebied, dan wordt een spanning doorgegeven (1), anders niet (0).

lees verder

Als we de cirkelrand uitvouwen krijgen we een patroon van geleidende en isolerende gebieden. Als we dit patroon kiezen overeenkomstig de binaire codering van de getallen 0 tot en met 7 (zie figuur 1-(a)), dan is uit de doorgegeven spanningen eenvoudig de aangewezen sector te bepalen.




(a)-Binaire codering (b)-Gray codering

*Figuur 1: Patronen op de cirkelrand*

Als de pijl blijft staan op de grens tussen twee sectoren, dan kan dit problemen opleveren. Bijvoorbeeld als de pijl blijft staan op de grens tussen de sectoren 3 (011) en 4 (100) dan kan dat leiden tot het doorgeven van de code 111, behorend bij sector 7. En dat ligt een heel eind uit de buurt.

Een bit-codering die dit probleem enigszins ondervangt is een Gray-code (zie figuur 1-(b)). Een Gray-code heeft de eigenschap dat de coderingen van twee naast elkaar liggende sectoren in precies één positie verschillen. De volgende tabel geeft een overzicht van een aantal getallen in decimale, binaire en Gray-codering.

| | | gray-code | |
|---|---|---|---|
| decimaal | binair | 2 bits | 3 bits |
| 0 | 0 | 00 | 000 |
| 1 | 1 | 01 | 001 |
| 2 | 10 | 11 | 011 |
| 3 | 11 | 10 | 010 |
| 4 | 100 | | 110 |
| 5 | 101 | | 111 |
| 6 | 110 | | 101 |
| 7 | 111 | | 100 |

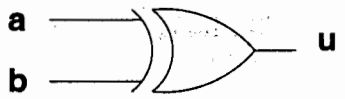
De hier gepresenteerde Gray-codering wordt wel *Gespiegelde Gray-code* genoemd. Bekijk de lijst van 3 bits. In de eerste vier codes herkennen we rechtstreeks de lijst van 2 bits: ze zijn ontstaan door de lijst van 2 bits te nemen en voor elke bitstring een 0 te plaatsen.
In de tweede groep van vier herkennen we ook de lijst van 2 bits, maar nu van *beneden naar boven* en voorafgegaan door een 1.

[4 pt] ☐ 1. Geef in een tabel van de getallen 8 tot en met 15 de binaire- en de Gray-code.

[5 pt] ☐ 2. Geef de Gray-code van het getal 140. Laat zien hoe je aan je antwoord komt.

lees verder

Een schakeling die bij een Gray-code de bijbehorende binaire code geeft, maakt gebruik van de XOF-poort:

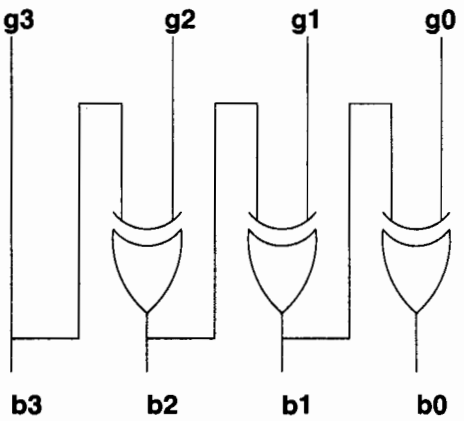



| a | b | u |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

*Figuur 2: De XOF-poort: symbool en functietabel*

[4 pt] □ 3. Ontwerp een schakeling bestaande uit de standaard poorten (EN, OF, NIET) die de XOF-poort implementeert.

De tekening hiernaast geeft een conversie-schakeling die bij de vier-bits Gray-code $g_3g_2g_1g_0$ de bijbehorende binaire code $b_3b_2b_1b_0$ genereert.




[5 pt] □ 4. Bereken met een vergelijkbare schakeling welk geheel getal de bitstring `11010101` als Gray-code heeft. Laat duidelijk zien hoe je aan je antwoord komt.

Het converteren kunnen we natuurlijk ook door een Haskell-functie laten uitvoeren. Dat gaat echter het eenvoudigst als het meest significante bit achteraan staat. Dat wil zeggen dat we de bitstring $a_n \ldots a_2a_1a_0$ representeren door de lijst `[a0, a1, a2, ... , an]`. We kiezen er voor om de bits te representeren met de karakters `0` en `1`.

[3 pt] □ 5. Geef een Haskell-functie `xof :: Char -> Char -> Char` die de XOF-schakeling implementeert.

[5 pt] □ 6. Geef bij de beschreven representatie van de bitstrings een Haskell-functie `gray2bin` die bij een gray-code de bijbehorende binaire code oplevert.

lees verder

# Opgave 2 (Verzamelingen)

Deze opgave gaat over verzamelingen van gehele getallen. Als representatie van zo'n verzameling kiezen we een lijst van integers. Aangezien een verzameling geen duplicaten heeft, zullen de lijsten ook geen duplicaten (mogen) bevatten.

Een veelvoorkomende operatie op verzamelingen is het bepalen van de *doorsnede* van twee verzamelingen. De doorsnede van twee verzamelingen $A$ en $B$ is de verzameling van alle elementen die zowel in $A$ als in $B$ zitten.

[3 pt] ☐ 7. Geef een Haskell-functie `isElt` die bij een element `x` en een lijst `lst` oplevert of `x` voorkomt in `lst`.

[4 pt] ☐ 8. Geef een Haskell-functie `doorsnede` die bij twee verzamelingen (lijsten) de doorsnede oplevert.

[4 pt] ☐ 9. Bepaal de worst case tijdcomplexiteit van `doorsnede`.

Als de lijsten die verzamelingen representeren gesorteerd zijn, dan is het mogelijk om een efficiëntere implementatie van `doorsnede` te maken.

[4 pt] ☐ 10. Geef, onder de veronderstelling dat de lijsten gesorteerd zijn, een efficiëntere implementatie van `doorsnede`.

[4 pt] ☐ 11. Bepaal de worst case tijdcomplexiteit van de functie uit de vorige vraag.

lees verder

# Opgave 3 Knapzakprobleem (1)

Deze opgave gaat over het knapzakprobleem. De context is als volgt. Je gaat op rugzakvakantie. Als voorbereiding verzamel je alle zaken die je in principe mee zou willen nemen. Als je dat hebt gedaan, dan blijkt (waarschijnlijk) dat je dat nooit allemaal kunt dragen.

Daarom bepaal je als eerste bij ieder artikel het gewicht. Ten tweede geef je elk artikel een waardering (een positief geheel getal) waarmee je uitdrukt hoe nodig dit artikel is of hoe graag je het mee wilt nemen.

De bedoeling is nu om een keus uit de verzameling artikelen te maken zo, dat de som van de gewichten niet groter is dan wat je kunt dragen en waarbij de som van de waarderingen maximaal is:

**Knapzakprobleem (Knapzak)**
**Parameter:** een verzameling $V$ van paren van positieve gehele getallen en een natuurlijk getal $G$.
**Gevraagd:** een deelverzameling $U$ van $V$ waarvan de som van de eerste elementen hoogstens $G$ is en de som van de tweede elementen maximaal.

In Haskell modelleren we de verzameling artikelen door een lijst van paren van positieve gehele getallen. Het eerste element in zo'n paar is het gewicht; het tweede element is de waardering.

Bekijk nu de volgende Haskell-functie:

```
maxWrd :: [(Integer, Integer)] -> Integer -> Integer

maxWrd [] rest = 0
maxWrd ((g,w) : lst) rest
  | g > rest          = zonder
  | otherwise         = max (w + met) zonder
           where met    = maxWrd lst (rest - g)
                 zonder = maxWrd lst  rest
```

[5 pt] □ 12. Leg uit dat bij een lijst `artLst` van artikelen (paren van positieve gehele getallen) en een te dragen gewicht `gew` ($\geq 0$) de aanroep `(maxWrd artLst gew)` de maximaal te realiseren totale waardering oplevert.

[5 pt] □ 13. Bepaal de worst case tijdcomplexiteit van de functie `maxWrd`.

[5 pt] □ 14. Geef een functie

```
besteKeus :: [(Integer, Integer)] -> Integer -> (Integer, [(Integer,Integer)])
```

die bij een lijst `artLst` van artikelen en een te dragen gewicht `gew` een paar `(wrd, keus)` oplevert. Hierin is `wrd` de maximaal te realiseren waardering en is `keus` een deellijst van `artLst` die deze waardering realiseert.

lees verder

## Opgave 4 Knapzakprobleem (2)

Het knapzakprobleem uit de vorige opgave is een optimaliseringsprobleem. We hebben voor een aantal gevallen gezien hoe bij een optimaliseringsprobleem een vergelijkbaar beslissingsprobleem kan worden geformuleerd.

[3 pt] ☐ 15. Leg uit wat een beslissingsprobleem is.
[4 pt] ☐ 16. Formuleer het met het knapzakprobleem uit de vorige opgave corresponderende beslissingsprobleem (**Knapsack**).
[5 pt] ☐ 17. Laat zien dat (**Knapsack**) $\in$ **NP**.

We kunnen bewijzen dat (**Knapsack**) NP-volledig is, gebruikmakend van het feit dat (**TSP**) NP-volledig is.

[3 pt] ☐ 18. Leg uit wat het betekent dat een beslissingsprobleem NP-volledig is.
[5 pt] ☐ 19. Leg uit in welke richting de reductie moet gaan en wat de bewijsverplichtingen zijn.

## Opgave 5 Grammatica's en automaten

Met alfabet {a, b} en startsymbool <S> is gegeven de grammatica $G_0$ met productieregels:

```
<S>  ::=  a<A> | b<B>
<A>  ::=  a | aa<A> | ab<B> | ba<B> | bb<A>
<B>  ::=  b | aa<B> | ab<A> | ba<A> | bb<B>
```

[5 pt] ☐ 20. Geef een afleidingsboom bij de string aabaaabbab
[5 pt] ☐ 21. Is de grammatica $G_0$ ambigu? Licht je antwoord duidelijk toe.
[4 pt] ☐ 22. Beschrijf duidelijk welke strings met behulp van de grammatica $G_0$ kunnen worden afgeleid.
[5 pt] ☐ 23. Teken een eindige automaat bij de grammatica $G_0$

De grammatica $G_1$ is een uitbreiding van de grammatica $G_0$. Het alfabet is {a, b, c} en het startsymbool is nu <T>:

```
<T>  ::=  c <T> c | <S><S>
<S>  ::=  a<A> | b<B>
<A>  ::=  a | aa<A> | ab<B> | ba<B> | bb<A>
<B>  ::=  b | aa<B> | ab<A> | ba<A> | bb<B>
```

[5 pt] ☐ 24. Is de grammatica $G_1$ ambigu? Licht je antwoord duidelijk toe.
[6 pt] ☐ 25. Is er een eindige automaat bij de grammatica $G_1$? Bewijs je bewering.
[4 pt] ☐ 26. Is er een Turingmachine te maken die van de input nagaat of het een string is die door de grammatica $G_1$ kan worden geproduceerd? Licht je antwoord toe.

einde