OI 2010 **Halve finale** 24 maart 2010	**Vul deze gegevens in, in HOOFDLETTERS en LEESBAAR aub** VOORNAAM : ..... NAAM : ..... SCHOOL :	**Gereserveerd**

Belgische Olympiades in de Informatica (duur : maximum 3u)

Dit is de vragenlijst van de halve finale van de Belgische Olympiades in de Informatica voor de categorie hoger onderwijs. Ze bevat 6 vragen. De eerste twee vragen zijn meerkeuzevragen, de vier overige vragen zijn open vragen. Naast elke vraag vind je een indicatie van de tijd die het mogelijk kost om de vraag op te lossen. Dit is slechts een schatting.

Algemene Opmerkingen (lees dit aandachtig voordat je begint met het beantwoorden van de vragen)

1. Vul je naam, voornaam en school **enkel in op het eerste blad**. Op alle andere bladzijden mag je alleen schrijven in de **kaders voorzien voor je antwoord**.

2. Je mag alleen iets bij je hebben om te schrijven. Rekenmachines, GSM, ... zijn **verboden**.

3. Schrijf je antwoorden met een **(bal)pen**. Laat geen antwoorden staan in potlood. Als je kladbladen wenst, vraag ze dan aan een toezichthouder.

4. Bij de meerkeuzevragen mag je slechts **één enkel antwoord** kiezen. Een correct antwoord levert je 1 punt op, geen antwoord 0 punten, en een fout antwoord wordt bestraft met $-0{,}5$ punt.

5. Op de open vragen mag je antwoorden in pseudo-code, met behulp van een diagram of in een van de toegelaten programmeertalen: Java, C, C++, C#, Pascal, Python, Ruby, PHP en Visual Basic. In het laatste geval, mag je **geen enkele externe bibliotheek gebruiken**, enkel standaardconstructies uit de taal zijn toegestaan. Syntaxfouten worden niet bestraft. In geval van twijfel, doe beroep op een toezichthouder.

6. Je mag hulpfuncties definiëren en recursie gebruiken, behalve als in de vraag expliciet wordt vermeld dat het verboden is.

7. Let goed op de indexering van arrays. De conventies kunnen elke vraag veranderen.

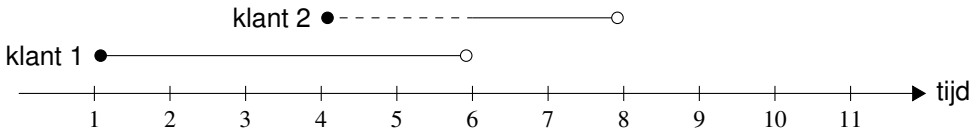
8. Je hebt **exact 3 uur** de tijd om alle vragen te beantwoorden.

Succes !

Vraag 1 – De supermarkt (10 min)

Je bestudeert het gedrag van klanten van een supermarkt, op het moment dat ze langs de kassa passeren. Een van je taken is het ogenblik te berekenen waarop de klanten zullen vertrekken (na hun boodschappen betaald te hebben), als je weet wanneer ze zijn beginnen aanschuiven en hoeveel tijd een kassier voor de klant nodig heeft gehad (d.w.z. de tijd dat het voor de kassier duurt om de rekening op te maken en af te rekenen).

Gegeven n klanten, genummerd van 1 tot n, aan eenzelfde kassa. De lijst in bevat de aankomsttijden in de rij aan de kassa van deze klanten, de lijst $service$ bevat de benodigde tijd aan de kassa en de lijst out bevat de resultaten die je berekent. Beschouw een voorbeeld met twee klanten. De eerste komt aan op tijdstip 1 en de kassier heeft 5 tijdseenheden nodig om hem te bedienen, de tweede komt aan op tijdstip 4 en heeft 2 tijdseenheden nodig aan de kassa. De eerste klant vertrekt dus op tijdstip 6 $(1 + 5)$. De tweede klant moet nog 2 tijdseenheden wachten totdat de eerste klant is bediend, en heeft zelf 2 tijdseenheden nodig om bediend te worden, en zal dus vertrekken op tijdstip 8.



Dit is het algoritme dat je voorstelt (een array van grootte n wordt geïndexeerd van 0 tot $n-1$) :

```
Input  : in, lijst van lengte n+1 : de aankomsttijden van klanten, in[0] = 0
         service, lijst van lengte n+1 : de benodigde tijd aan de kassa, service[0] = 0
         out, lijst van lengte n+1 en out[j]=0 voor 0 ≤ j ≤ n+1
         Ook: n > 0, in[j]> 0 en service[j]> 0 voor 1 ≤ j ≤ n , en in is oplopend gesorteerd
Output : out bevat de vertrektijden van de klanten

for (i ← 1 to n step 1)
{
    if ([...])
    {
        out[i] ← out[i − 1] + service[i]
    }
    else
    {
        out[i] ← in[i] + service[i]
    }
}
return out
```

Wat is de ontbrekende conditie in het algoritme opdat het de correcte vertrektijden zou berekenen?

☐	$in[i] + service[i-1] > out[i-1]$
☐	$in[i-1] + service[i-1] > in[i+1]$
☐	$out[i-1] > in[i]$
☐	$out[i] > service[i-1] + in[i-1]$

Vraag 2 – Mysterieuze Functie (10 min)

Het onderstaande algoritme doet een test op een reeks van karakters s. Twee functies worden gebruikt in het algoritme, die beschreven worden als volgt:

`length (s)`	geeft de lengte van de reeks s terug (het aantal karakters)
`substr (s, i, n)`	geeft een deelreeks van s terug, beginnend op index i en met lengte n.

We veronderstellen dat reeksen van karakters geïndexeerd worden vanaf 1 (het eerste karakter heeft dus index 1). Hier is het mysterieuze algoritme:

```
Input  : s, een niet-lege reeks karakters
Output : ?

function mystery (s)
{
    if (length (s) < 2)
    {
        result ← true
    }
    else
    {
        if (s[1] = s[length (s)])
        {
            result ← mystery (substr (s, 2, length (s) − 2))
        }
        else
        {
            result ← false
        }
    }
}
```

Wat geeft deze functie terug bij de volgende input:

1. `"nora bedroog o zo goor de baron"`
2. `"de sla nemen als ed"`
3. `"er is daar nog onraad sire"`

☐	**true**, **false**, **true**
☐	**true**, **true**, **true**
☐	**false**, **true**, **false**
☐	**false**, **true**, **true**

Vraag 3 – Sudoku (20 min)

Voor een nieuwe website, die bezoekers online sudoku's wilt laten oplossen, moet je een algoritme schrijven dat nagaat of een sudoku correct is ingevuld. Voor een sudoku met zijden van lengte n, moet je nagaan dat elke rij en elke kolom de gehele getallen van 1 tot n bevat, waarbij elk getal exact één keer voorkomt. Hieronder staan 2 algoritmes die een array van gehele getallen arr aanvaarden, geïndexeerd van 1 tot n, met $n > 0$, en die een booleaanse waarde teruggeeft: **`true`** als arr een permutatie van de gehele getallen van 1 tot n bevat, en anders **`false`**. Bijvoorbeeld, $[2, 4, 1, 3, 5]$ is een permutatie van de gehele getallen van 1 tot 5 en beide algoritmes geven dus **`true`** terug. Voor $[1, 4, 2]$ daarentegen geven ze **`false`** terug.

Algoritme 1

```
temp ← array van gehele getallen met lengte n, geindexeerd van 1 tot n,
       zodat temp[j] = 0, ∀j : 1 ≤ j ≤ n
b ← true
i ← 1

while (i ≤ n and b)
{
    b ← (1 ≤ arr[i] and arr[i] ≤ n)
    if (b)
    {
        temp[arr[i]] ← temp[arr[i]] + 1
        b ← (temp[arr[i]] = 1)
    }
    i ← i + 1
}
return b
```

Algoritme 2

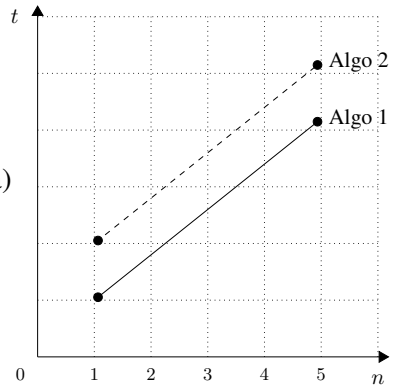
```
b ← true
i ← 1

while (i ≤ n and b)
{
    b ← (1 ≤ arr[i] and arr[i] ≤ n)
    j ← i + 1
    while (j ≤ n and b)
    {
        b ← not (arr[i] = arr[j])
        j ← j + 1
    }
    i ← i + 1
}
return b
```

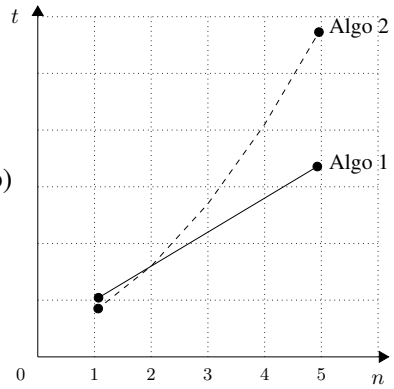
Analyseer de tijdscomplexiteit van deze twee algoritmes, om te kunnen vinden welk van de twee het snelste uitvoert. Je website moet immers zo snel mogelijk zijn en zo min mogelijk tijd spenderen aan het uitvoeren van het algoritme, zeker voor sudoku's groter dan 9×9.

Om de uitvoeringstijd te bepalen, stellen we de volgende hypotheses op om de zaken te vereenvoudigen: de toekenning van een variabele telt als 1, het evalueren van de conditie in een **`if`** of **`while`** telt ook als 1. De twee algoritmes werden uitgevoerd met arrays van grootte 1, 2, 3, 4 en 5 en de uitvoeringstijd werd telkens gemeten. Welke van deze vier grafieken is diegene die we hebben bekomen?

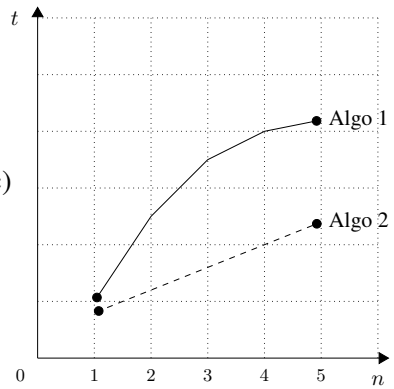
☐ (a)



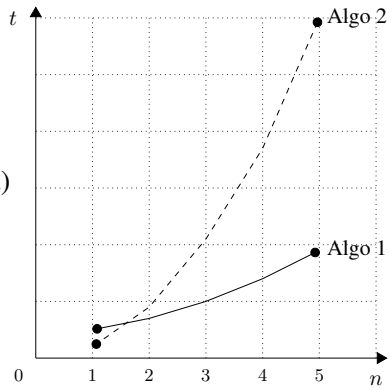
☐ (b)



☐ (c)



☐ (d)



Als je deze twee curves bekijkt, wat is dat het algoritme dat het efficiëntst is in de tijd, als de waarde van n toeneemt? Wat moest er opgeofferd worden opdat het sneller zou zijn dan het andere?

Q3

Vraag 4 – Te ingewikkeld! (15 min)

Je broer is op computerkamp geweest vorig jaar. Hij is heel fier dat hij veel heeft bijgeleerd, en vooral dat hij er in geslaagd is zijn eerste algoritme helemaal zelf te schrijven. Dat algoritme neemt twee natuurlijke getallen `a` en `b` en geeft als resultaat een natuurlijk getal. Hieronder staat het algoritme, waarin de operatoren `/` en `%` respectievelijk het quotient en de rest van de gehele deling berekenen.

```
Input  : a en b, twee natuurlijke getallen
Output : ?

c ← 0
p ← 1
while (a ≠ 0 and b ≠ 0)
{
    c ← c + p * ((a % 10) + (b % 10))
    p ← p * 10
    a ← a / 10
    b ← b / 10
}
return c
```

Leg eerst uit wat dit algoritme doet. Stel daarna een eenvoudigere versie voor.

Q4

Vraag 5 – Hoeveel woorden zijn er ? (50 min)

We beschouwen een reeks karakters, samengesteld uit woorden die gescheiden worden door een of meerdere scheidingstekens. De woorden bevatten de karakters `a-z` en `A-Z`, en de scheidingstekens kunnen een spatie zijn ( ), een komma (`,`) of een punt (`.`).

Veronderstel dat deze reeks karakters voorgesteld wordt als een array van karakters `s`. Er worden 2 dingen gevraagd:

1. Stel x een positief geheel getal. Schrijf een functie `countWords (s, x)` die het aantal woorden van lengte x telt die zich bevinden in de reeks karakters `s`;
2. Schrijf een functie `countWordsWithChar (s, c)` die in de reeks karakters `s`, het aantal woorden telt die het gegeven karakter `c` bevatten.

Hier zijn enkele voorbeelden:

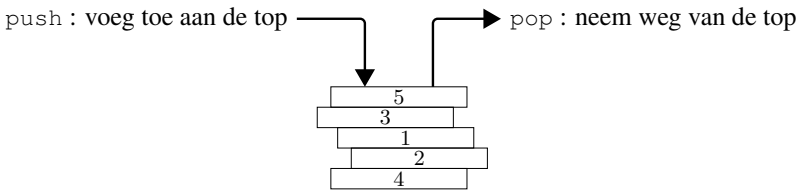
- De zin `"Toto zag Lulu ... Maar Lulu zag Toto niet."` bevat 6 woorden van lengte 4. Het karakter `'o'` komt voor in 2 woorden; het karakter `'a'` in 3 woorden.
- De zin `"Leve de olympiades in de informatica"` bevat 1 woord van lengte 4. Het karakter `'i'` komt voor in 3 woorden.

Q5

Q5 (vervolg)

Question 6 – Sorteer je stripcollectie ! (50 min)

Je broer is een grote fan van de stripreeks *"Computer Man"* en heeft alle nummers van deze reeks. Het enige nadeel is dat hij ze nadat hij ze leest, teruglegt in een willekeurige volgorde op een willekeurige plaats. Op een dag beslist hij op te ruimen en hij heeft alle nummers op elkaar gestapeld.



Een gegevensstructuur die we *stack* noemen laat toe om deze situatie voor te stellen. Zodra we een stack hebben, kunnen we de grootte ervan opvragen (het aantal elementen in de stack), we kunnen er een element aan toevoegen (steeds bovenaan) en we kunnen er een element afhalen (steeds het bovenste). De 4 mogelijke operaties zijn dus:

`new`	maakt een nieuwe, lege stack
`push (p, i)`	voegt het element i toe aan de top van stack p
`pop (p)`	neemt het bovenste element van de stack p weg en geeft het terug
`size (p)`	geeft de grootte van de stack p terug.

Hieronder staat de code waarmee je de illustratie van de stack hierboven aanmaakt, en de stack daarna volledig leegmaakt.

```
Strips ← new
push (Strips, 4)
push (Strips, 2)
push (Strips, 1)
push (Strips, 3)
push (Strips, 5)
while (size (Strips) ≠ 0)
{
    bd ← pop (Strips)
}
```

Schrijf een algoritme dat enkel stacks en gehele getallen gebruikt, zonder enige andere datastructuur (lijsten, arrays, maps, etc). Het algoritme krijgt als input een stack van gehele getallen, allemaal door elkaar. Na de uitvoering van het algoritme moet de stack die als input werd gegeven zijn aangepast zodat die dezelfde getallen bevat als in het begin, maar nu op volgorde. Als de gegeven stack niet leeg was, moet na afloop van je algoritme de `pop`-functie het kleinste element uit de stack teruggeven. Als extraatje: kan je er geraken door de operatie `pop` maximaal $n * (n + 1)$ keer op te roepen voor een stack van grootte n ? (bonus)

Q6

. . . . . .

. . . . . .

. . . . . .

Q6 (vervolg)