be-OI 2011 **Finale** 30 maart 2011	**Gegevens invullen in HOOFDLETTERS en LEESBAAR, aub** VOORNAAM : ..... NAAM : ..... SCHOOL :	**Gereserveerd**

Belgische Olympiades in de Informatica (duur : 1u40 maximum)

Dit is de vragenlijst van de finale van de Belgische Olympiades 2011 voor de **categorie secundair onderwijs**. Ze bevat 7 vragen waarvoor je **maximum 1u40** de tijd krijgt. Naast elke vraag staat een indicatie van de tijd die het kan kosten om de vraag op te lossen. Dat is slechts een schatting.

Algemene opmerkingen (lees dit aandachtig voor je begint)

1. Vul je voornaam, naam en school in, **alleen op het eerste blad**. Op alle andere bladzijden mag je enkel schrijven in de daarvoor **voorziene kaders**. Als je door een fout toch buiten de antwoordkaders moet schrijven, moet je wel op hetzelfde blad papier verder schrijven. Anders kunnen we je antwoord niet verbeteren.

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

3. Schrijf je antwoorden met blauwe of zwarte **pen of balpen**. Laat geen antwoorden staan in potlood. Als je kladbladen wilt, vraag ze dan aan een toezichthouder.

4. Je **moet** op de vragen antwoorden in **pseudo-code** of in één van de **toegestane programmeertalen** (Java, C, C++, Pascal, Python, PHP). We trekken geen punten af voor syntaxfouten. Je mag geen voorgedefinieerde functies gebruiken, behalve diegenen die op de volgende bladzijde staan.

5. Je mag **op geen enkel moment met iemand communiceren**, behalve met de organisatoren of toezichthouders. Alle technische en inhoudelijke vragen mag je enkel stellen aan de organisatoren. Logistieke vragen mogen gesteld worden aan de toezichthouders.

6. Je mag je **plaats niet verlaten** tijdens de proef, bijvoorbeeld om naar het toilet te gaan of te roken. Afhankelijk van de plaats waar je de proef aflegt, kan het ook verboden zijn om te eten of te drinken in het lokaal.

7. Je krijgt **exact 1 uur en 40 minuten** de tijd om op de vragen te antwoorden. Een **overzicht** over pseudo-code en de afspraken en conventies die we hanteren staat op de volgende bladzijde.

Succes !

001

Vragenlijst finale secundair

Overzicht pseudo-code

In de vragenlijst kom je opgaves tegen waarbij je een stuk code moet aanvullen. We vragen je om dat te doen met ofwel *expressies*, ofwel *instructies*. Expressies berekenen een waarde, instructies voeren een actie uit.

Expressies bezitten een waarde en worden gebruikt in berekeningen, om de waarde van een variabele te veranderen, en in condities. Sommige expressies hebben een numerieke waarde (een geheel of reëel getal), andere een booleaanse waarde (**`true`** of **`false`**). Instructies stellen acties voor die uitgevoerd moeten worden. Een algoritme is een opeenvolging van instructies.

De meest eenvoudige expressies zijn de gehele getallen zelf (zoals 3, -12...) en variabelen (zoals x, sum...). We kunnen een expressie ook samenstellen met operatoren zoals de volgende tabel toont.

operatie	**notatie**	**voorbeelden**	
gelijkheid	$=$	$3 = 2$ is **`false`**	$1 = 1$ is **`true`**
verschil	$\neq$	$1 \neq 4$ is **`true`**	$3 \neq 3$ is **`false`**
vergelijking	$>, \geq, <, \leq$	$1 > 3$ is **`false`**	$5 \leq 5$ is **`true`**
optellen en aftrekken	$+, -$	$3 + 2$ is 5	$1 - 9$ is -8
vermenigvuldigen	$\times$	2×7 is 14	-2×3 is -6
gehele deling	$/$	$10/3$ is 3	$9/3$ is 3
rest van de gehele deling	$\%$	$10\%3$ is 1	$9\%3$ is 0
logische "niet"	**`not`**	**`not`** $3 = 2$ is **`true`**	**`not`** $1 = 1$ is **`false`**
logische "en"	**`and`**	$3 = 3$ **`and`** $5 \leq 9$ is **`true`**	$1 \geq 3$ **`and`** $2 = 2$ is **`false`**
logische "of"	**`or`**	$3 = 3$ **`or`** $12 \leq 9$ is **`true`**	$1 \geq 3$ **`or`** $2 \neq 2$ is **`false`**
toegang tot het element van een array	`[]`	gegeven arr is `[1,2,3]`, dan arr`[1]` is 2	
functie oproepen die een waarde teruggeeft		`size(`*list*`)`	`min(12, 5)`

De functies die jullie mogen gebruiken zijn, naast diegenen die in de opgaves en op deze bladzijde worden gedefinieerd, ook de functies `max(`a`, `b`)`, `min(`a`, `b`)` en `pow(`a`, `b`)`. Zij berekenen respectievelijk het grootste van twee getallen, het kleinste van twee getallen, of de machtsverheffing (a^b).

De meest eenvoudige instructies staan in deze tabel samengevat :

operatie	**notatie**	**voorbeeld**
toekenning	$\leftarrow$	$x \leftarrow 20 + 11 \times 2$
teruggave	**`return`**	**`return`** 42
functie oproepen (die niets teruggeeft)	naam van de functie	`sort(`*list*`)`

Daarnaast hebben we ook *controle-instructies* waarvan we er 3 gebruiken : **`if`**–**`else`**, **`while`** en **`for`**. Die laten toe een groep andere instructies uit te voeren, afhankelijk van een conditie. De notatie **`for`** `(`$i \leftarrow a$ **`to`** b **`step`** k`) { [...] }` staat voor een lus die herhaald wordt zolang $i \leq b$, waarbij i begint vanaf a en telkens verhoogd wordt met k aan het eind van elke herhaling.

We kunnen variabelen gebruiken om een waarde in op te slaan, en *arrays* om meerdere waarden op te slaan. Een array arr van grootte n wordt geïndexeerd van 0 tot $n - 1$. De notatie arr`[`i`]` geeft toegang tot het $(i + 1)$-de element van de array. Zo is het eerste element van die array arr`[0]`. We kunnen een nieuwe array arr aanmaken van grootte n, waarvan alle elementen geïnitialiseerd zijn op 0, met de notatie $arr \leftarrow$ `newArray(`n`)`. We kunnen array arr van grootte n kopiëren naar een nieuwe array $newArr$ kopiëren met de notatie $newArr \leftarrow arr$.

Over de kost van operaties

Bij vragen 3 en 5 wordt gevraagd een zo efficiënt mogelijk algoritme te schrijven om de maximumscore te kunnen halen.

Als we de tijdscomplexiteit van een programma willen kennen (een schatting van de tijd die het zal kosten om uit te voeren), is één van de mogelijke manieren het tellen van het aantal elementaire operaties die het algoritme uitvoert. Alle basisinstructies laten we meetellen als één tijdseenheid. Het aanmaken van een array van grootte n, of het kopiëren van zo'n array, kost n tijdseenheden. Voor de **`if`**–**`else`**, **`while`** en **`for`** instructies, tellen we één tijdseenheid voor het uitrekenen van de conditie. Daarna tellen we alle instructies die worden uitgevoerd.

Vraag 1 – Haakjes sluiten (5 min)

We willen een programma schrijven dat controleert of een wiskundige expressie, geschreven als een reeks van karakters, op een juiste manier gebruik maakt van haakjes. We willen dat er voor elk open haakje "(", verderop ook een gesloten haakje ")" aanwezig is. Dat wil ook zeggen dat er voor elk gesloten haakje eerder een open haakje werd tegengekomen.

Bijvoorbeeld, de reeks karakters "$(a + 2 + (c/4) - (1 + e))$" maakt correct gebruik van haakjes, terwijl deze twee dat niet doen : "$)a + (2 - x)/(1$" en "$a + (2 - 3)/((3 + a) + 3$". Vervolledig het volgende algoritme.

```
Input  : • s, een array van karakters.
         • n, positief geheel getal, de lengte van de array van karakters.
Output : • geeft true (waar) terug als de wiskundige expressie voorgesteld door s
           juist gebruik maakt van haakjes, en anders false (onwaar).
num ← 0

for (i ← 0 to n step 1)
{
    if (s[i] = '(')
    {
        [...]                                                    // Q1(a)
    }
    if (s[i] = ')')
    {
        [...]                                                    // Q1(b)
    }
}
return (num = 0)                    // geeft true als num gelijk is aan 0, anders false.
```

Q1(a) **(1 tot 5 instructies)**

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Q1(b) **(1 tot 5 instructies)**

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Vraag 2 – O dennenboom... (10 min)

Het onderstaande algoritme tekent een dennenboom op het scherm. Een dennenboom wordt getekend als een opeenstapeling van gelijkbenige driehoeken, die op hun beurt getekend worden met het karakter `X`, zoals hier :

```
   X
  XXX
   X
  XXX
 XXXXX
   X
  XXX
 XXXXX
XXXXXXX
   X
```

De lengte van de dennenboom hangt af van een parameter n. Vertrekkend van onder naar boven, bevat de dennenboom eerst een driehoek waarvan de basis lengte n heeft. De driehoek daarboven heeft een basis van lengte $n-2$. En zo gaan we verder tot de laatste driehoek, waarvan de basis 3 is. In het voorbeeld hierboven is n dus gelijk aan 7. De dennenboom bestaat dan uit 3 driehoeken, van onder naar boven : één met basis 7, één met basis 5 en één met basis 3. De dennenboom wordt afgewerkt met een stam zoals op de afbeelding.

Vervolledig het onderstaande algoritme, zodat het correct werkt volgens de aanwijzingen hierboven. Om dat te doen, gebruikt ons algoritme een functie `draw` die twee parameters x en y krijgt. Deze functie tekent op het scherm een lijn die bestaat uit x spaties, gevolgd door y keer het karakter `X` (en een regel-einde). Het tekenen gebeurt natuurlijk wel van boven naar beneden.

```
Input  : • n, strikt positief oneven getal, lengte van de basis
           van de onderste driehoek van de dennenboom.
Output : • een dennenboom van breedte n (zie uitleg) wordt op het scherm getekend.

function dennenboom (n)
{
    basis ← 3
    afstand ← n/2 − 1
    while (basis ≤ n)
    {
        numkar ← 1
        while ([...])                                          // Q2 (a)
        {
            spaties ← afstand + [...]                          // Q2 (b)
            draw (spaties, numkar)
            numkar ← numkar + 2
        }
        basis ← basis + 2
        [...]                                                  // Q2 (c)
    }
    draw ([...], 1)                                            // Q2 (d)
}
```

Q2(a) **(een expressie)**

. . . . . .

Q2(b) **(een expressie)**

. . . . . .

Q2(c) **(een instructie)**

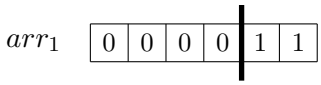
. . . . . .

Q2(d) **(een expressie)**

. . . . . .

Vraag 3 – Maak de som... (15 min)

Gegeven een array die bestaat uit nullen en énen, en die in 2 stukken verdeeld kan worden : het linkerdeel bevat enkel 0 en het rechterdeel enkel 1. Dit is een voorbeeld van zo'n array, met de plaats van de verdeling aangeduid door een dikke lijn :



We vragen je een algoritme te schrijven dat toelaat de som van de elementen in de array te berekenen. Bovendien moet je algoritme **zo efficiënt mogelijk zijn**.

Input : • arr, een array bestaand uit een reeks 0-en gevolgd door een reeks 1-en.
• n, een positief geheel getal, lengte van arr.
Output : • De som van de elementen in de array wordt teruggegeven.
• de array arr mag niet gewijzigd worden.

Je algoritme moet correct werken, maar we houden in de beoordeling ook rekening met :

- het aantal leesoperaties die worden uitgevoerd **in het slechtst mogelijk geval** ;
- de eenvoud van je algoritme. Idealiter bevat het slechts één **`while`**-lus, één **`if`**–**`else`**-structuur, enkele toekenningen aan variabelen en als laatste een instructie **`return`**.

Q3

Vraag 4 – Boekenkast sorteren... (15 min)

Ik heb twee onvolledige verzamelingen van magazines, opgestapeld en gesorteerd. Het meest recente nummer bevindt zich bovenaan de stapel, die we in het Engels "stack" noemen. Op zo'n stack kunnen we verschillende operaties uitvoeren. Gegeven een stack s is het volgende mogelijk :

- `top(`p`)` geeft je het bovenste element van de stack p.
- `pop(`p`)` geeft je het bovenste element van de stack p en verwijdert het ook uit de stack.
- `push(`p`,`e`)` plaatst een element e bovenop de stack p.
- `isEmpty(`p`)` laat je weten of de stack p leeg is of niet.

Nemen we als voorbeeld twee stapels of stacks, $p1$ en $p2$:

$p1$
1
3
5
8
8
12

$p2$
2
3
5
10

Schrijf een algoritme dat, gegeven twee stacks $p1$ en $p2$, twee nieuwe stacks aanmaakt die we $collectie$ en $dubbels$ zullen noemen. De twee nieuwe stacks bevatten dezelfde magazines, maar anders geklasseerd. De eerste nieuwe stapel zal een zo compleet mogelijke collectie magazines bevatten. Ze mag slechts 1 exemplaar van elk magazine bevatten en moet gesorteerd zijn op zo'n manier dat het nieuwste magazine onderaan ligt, en het oudste bovenaan. De tweede nieuwe stapel bevat dan alle dubbels en is op dezelfde manier gesorteerd als de eerste nieuwe stapel. Voor het voorbeeld hierboven, verwachten we dus het volgende resultaat :

$collectie$
12
10
8
5
3
2
1

$dubbels$
8
5
3

We vragen je om het volgende algoritme te vervolledigen zodat het doet wat er gevraagd wordt.

```
Input  : • p1, p2 zijn twee stacks waarvan de elementen gesorteerd zijn,
           het kleinste bevindt zich bovenaan.
Output : • twee stacks (collectie, dubbels) die de collectie en dubbels bevatten
    zoals gedefinieerd in de opgave.

collectie ← een nieuwe lege stack
dubbels ← een nieuwe lege stack

while ([...])                                                    // Q4(a)
{
    if ([...])                                                   // Q4(b)
    {
        min ← pop (p1)
    }
    else
    {
        min ← pop (p2)
    }

    if ([...])                                                   // Q4(c)
    {
        push (collectie, min)
    }
    else
    {
        push (dubbels, min)
    }
}

return (collectie, dubbels)
```

Q4(a) **(een expressie)**

. . . . . .

Q4(b) **(een expressie)**

. . . . . .

Q4(c) **(een expressie)**

. . . . . .

Vraag 5 – Vermenigvuldigen (15 min)

Het volgende algoritme vermenigvuldigt twee positieve getallen die zijn opgeslagen in de vorm van een array van cijfers. Het resultaat wordt teruggegeven als een nieuwe array van cijfers. Vervolledig dit algoritme.

Bijvoorbeeld, gegeven a en b zijn respectievelijk :

a | 3 | 1 | 0 | b | 1 | 8 |

Dan is de oplossing :

r | 0 | 5 | 5 | 8 | 0 |

```
Input  : • a en b, twee niet-lege arrays van cijfers.
         • n en m, positieve gehele getallen, lengtes van respectievelijk a en b.
Output : • Een array die het resultaat voorstelt van de vermenigvuldiging van de
           getallen die door a en b worden voorgesteld. De arrays a en b worden niet
           aangepast.

function mult (a, b, n, m)
{
    r ← een nieuwe array van cijfers met lengte (n + m), geïnitialiseerd met 0-en

    for (j ← m − 1 to 0 step −1)
    {
        for (i ← n − 1 to 0 step −1)
        {
            x ← a[i] × b[j]
            r[...] ←[...]                                          // Q5(a)
            r[...] ←[...]                                          // Q5(b)
        }
    }
    return r
}
```

Vervolledig dit algoritme.

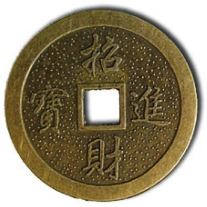
Q5(a) **(een toekenning)**

. . . . . .

Q5(b) **(een toekenning)**

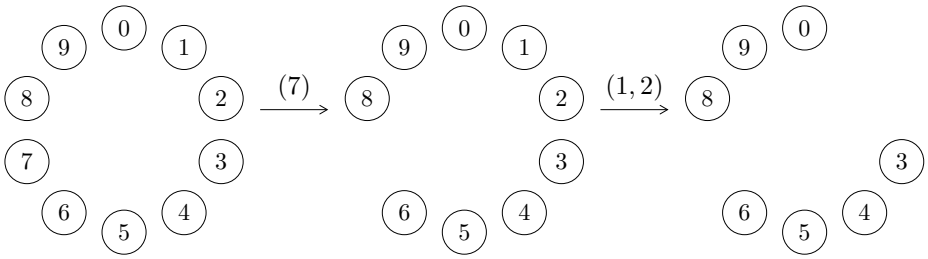
. . . . . .

Vraag 6 – Wie heeft de laatste ? (15 min)



In het volgende spel, wordt een reeks van n (n is een even getal) Chinese muntstukken in een cirkel op tafel gelegd. We nummeren ze van 0 tot $n-1$ in wijzerzin. We spelen het spel met 2 spelers. Om beurt moet de speler 1 of 2 stukken wegnemen, en als er 2 worden weggenomen moeten die naast elkaar liggen. De winnaar is diegene die de laatste aan de beurt is (d.w.z. diegene die het laatste stuk wegneemt).

Hier is een voorbeeld van de beginsituatie :



In dit spel kan de tweede speler altijd winnen, ongeacht hoeveel stukken de eerste speler steeds wegneemt. Het volgende algoritme stelt de strategie voor die speler 2 daarvoor moet gebruiken. Je moet het nog vervolledigen, op zo'n manier dat speler 2 er altijd zeker van is te winnen als hij deze strategie volgt vanaf het begin (n muntstukken op tafel).

```
Input  : • n, een even positief geheel getal > 3, het aantal stukken.
         • (last_i, last_b), met last_i het nummer van het muntstuk weggenomen door speler 1
           tijdens de vorige beurt, en last_b het aantal weggenomen stukken (1 of 2).
           In het geval dat er 2 stukken zijn weggenomen, stelt last_i het eerste van de
           twee voor (in wijzerzin). Het spel is in een configuratie waarbij speler 2
           aan de beurt is en die heeft vanaf het begin deze zelfde strategie gebruikt.

Output : • Een koppel getallen (i, b) waarbij i het nummer is van het stuk dat speler 2
           moet wegnemen, en b het aantal stukken dat weggenomen moet worden (1 of 2).
           Als er 2 stukken worden weggenomen, is i het eerste van de twee
           (in wijzerzin).

return ([...], [...])                                          // Q6(a,b)
```

Q6(a) **(een expressie)**

. . . . . .

Q6(b) **(een expressie)**

. . . . . .

Vraag 7 – Orde in de chaos (25 min)

De elementen van een array sorteren in oplopende volgorde is een klassiek algoritmisch probleem. De efficiëntie waarmee dat kan gebeuren is bepalend voor de efficiëntie van talloze andere veel voorkomende algoritmes. Een van de meest gebruikte, want meest efficiënte, sorteeralgoritmes is *"Quick Sort"*. Het werkt als volgt.

Allereerst kiezen we een element in de ongesorteerde array (meestal het laatste) dat we *pivot* noemen. Vervolgens delen we de elementen van de array op in twee groepen : de eerste groep bevat alle waarden die strikt kleiner zijn dan het pivot element, en de tweede groep alle waarden die gelijk zijn aan of groter zijn dan het pivot element. Door deze opdeling willen we komen tot een array die van links naar rechts de volgende elementen bevat : alle elementen uit de eerste groep, het pivot element, en daarna alle elementen uit de tweede groep. We passen hetzelfde systeem toe op de twee subgroepen die we zonet gemaakt hebben, en zo verder, totdat we groepen van elementen bekomen die slechts 1 element bevatten. Het principe hierachter staat ook bekend als "verdeel en heers".

Het algoritme ziet eruit als volgt :

Input : • arr, een array van gehele getallen
• $begin$ en $eind$, indices van de array, zodat $begin \leq eind$.
Output : • De elementen in de array tussen de posities $begin$ en $eind$ ($begin$ en $eind$ inbegrepen) zijn gesorteerd in oplopende volgorde.
• Het algoritme geeft geen waarde terug.

```
function quicksort (arr, begin, eind)
{
    if (eind − begin > 0)
    {
        pivot ← partition (arr, begin, eind)
        quicksort (arr, begin, pivot − 1)
        quicksort (arr, pivot + 1, eind)
    }
}
```

Het algoritme doet beroep op een functie *partition* die hieronder is beschreven. Om redenen van efficiëntie, worden er geen tijdelijke arrays gebruikt in *partition*. De opdeling in groepen gebeurt door het wisselen van elementen (`swap`). De functie $swap(arr, i, j)$ laat toe om de waarden op indices i en j van de array arr om te wisselen.

Input : • arr, een array van getallen.
• $begin$ en $eind$, indices van de array zodat $begin \leq eind$.
Output : • De waarden van arr tussen $begin$ en $eind$ (beide inbegrepen) zijn opnieuw georganiseerd zodat er een positie j bestaat, waarvoor $begin \leq j \leq eind$, zodat $arr[j]$ het pivot element bevat (dit is de waarde die zich eerst op $arr[eind]$ bevond). Op alle andere indices i, bevatten de posities $begin \leq i < j$ alle waarden die kleiner zijn dan het pivot element, en de posities $j < i \leq eind$ alle waarden die gelijk zijn aan of groter zijn dan het pivot element.
• arr blijft ongewijzigd vóór de index $begin$ en na de index $eind$.
• De uiteindelijke index van het pivot element wordt teruggegeven.

```
function partition (arr, begin, eind)
{
    pivot ← eind
    x ← begin
    for (i ← begin to eind step 1)
    {
        if (s[i] < s[pivot])
        {
            swap (arr, i, [...])                    // Q7(a)
            [...]                                   // Q7(b)
        }
    }
    [...]                                           // Q7(c)

    return x
}
```

Vervolledig dit algoritme (je mag de functie `swap` gebruiken als je dat nodig vindt).

Q7(a) **(een expressie)**

. . . . . .

Q7(b) **(een instructie)**

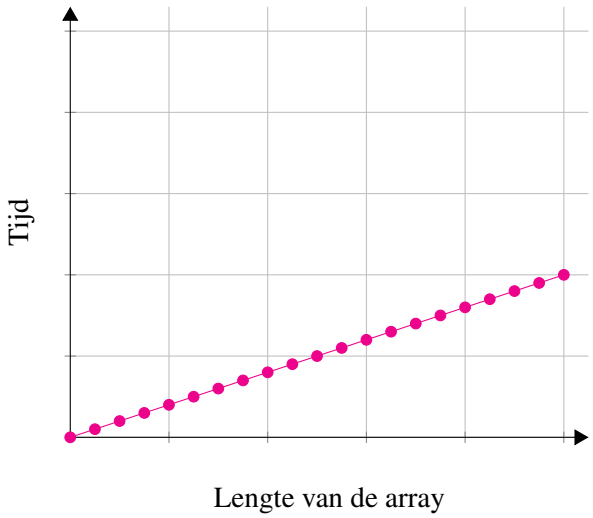
. . . . . .

Q7(c) **(een instructie)**

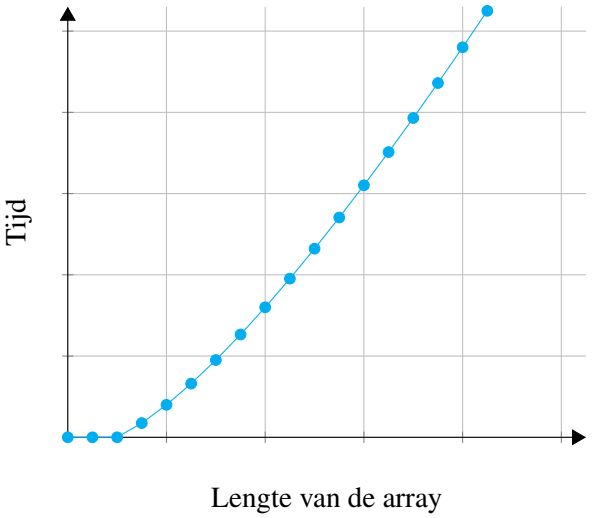
. . . . . .

Vraag 7 (vervolg)

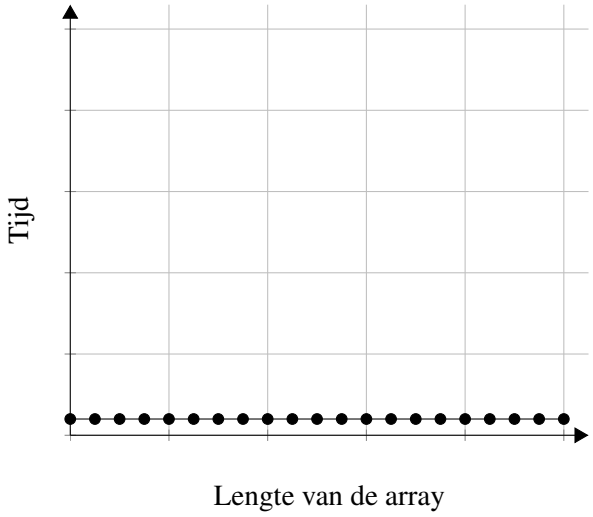
We willen nu weten wat de uitvoeringstijd is van *"Quick Sort"* naarmate de arrays groter en groter worden. Daarvoor stellen we dat alle gebruikte instructies (het testen van de conditie bij een `if` of een `while`, een toekenning, ...) dezelfde hoeveelheid tijd kosten om uit te voeren. Kies uit de onderstaande grafieken diegene die overeenkomt met de evolutie van de gemiddelde tijdskost van *"Quick Sort"*, in functie van de grootte van de array die we willen sorteren. Bij deze vraag worden geen punten afgetrokken bij een fout antwoord.



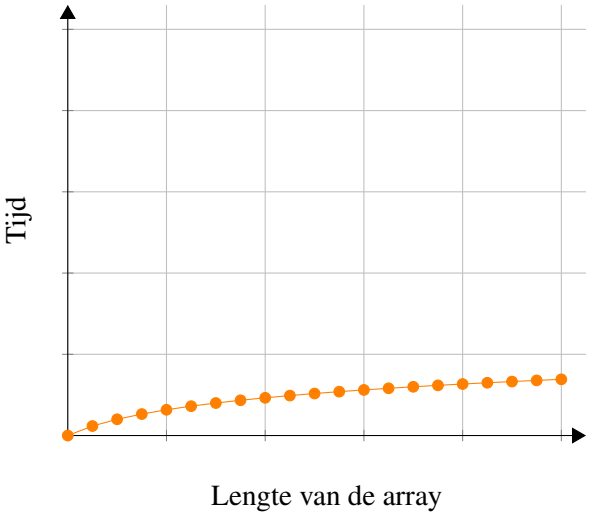
☐ **A**



☐ **B**



☐ **C**



☐ **D**