Programmeeropdracht 1 — Tegelspel
Algoritmiek, voorjaar 2024

Inleiding

Femke en Lieke hebben een druk programma. Toch proberen ze af en toe tijd vrij te maken om te ontspannen. Dan spelen ze graag een zelfbedacht tegelspel.

Hierbij heeft elke speler een bord met een aantal rijen met vakjes, die ze moet proberen te vullen met gele en/of blauwe tegels. De tegels zijn afkomstig van schalen. Om de beurt kiest een speler een schaal en een kleur tegels. Alle tegels van die kleur op de schaal worden dan in een van haar rijen met vakjes gelegd, maximaal één tegel per vakje. De tegels op de schaal worden aangevuld met tegels uit de pot. De speler die aan het eind van het spel de meeste volle rijen met tegels heeft is de winnaar. Femke begint.

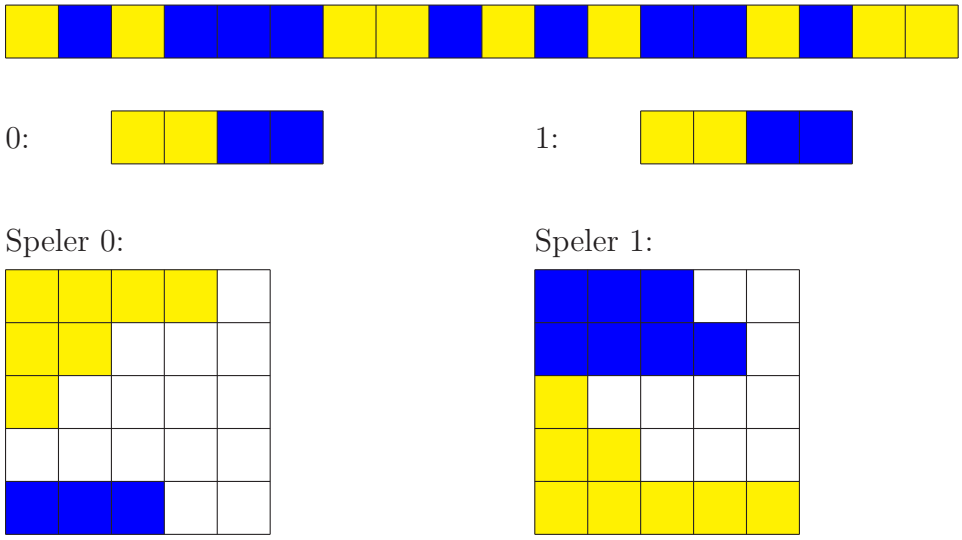

Figure 1: Voorbeeld van een stand bij het tegelspel, met achtereenvolgens de pot, twee schalen met elk vier tegels, en de twee borden van speler 0 (Femke) en speler 1 (Lieke). Femke is aan de beurt.

Uitwerking spelregels

Het aantal schalen in het spel noemen we M, het (maximum) aantal tegels op een schaal is N. Elke speler heeft een bord met K rijen met elk L vakjes. Er moet gelden: $1 \leq M \leq 5$, $1 \leq N \leq 5$, $1 \leq K \leq 10$ en $1 \leq L \leq 6$.

Femke en Lieke hebben bij het spel volledige informatie. Ze weten bijvoorbeeld welke tegels er in de pot zitten, en in welke volgorde ze worden overgeheveld naar de schalen. Bijvoorbeeld

in de stand in Figuur 1, komt er eerst een gele tegel uit de pot, dan een blauwe, dan weer een gele, dan drie blauwe, dan twee gele, enzovoort.

Als een speler een schaal en een kleur kiest, moet die kleur tegels daadwerkelijk voorkomen op de schaal. Alle tegels van die kleur van die schaal moeten dan bij elkaar in een zelfde rij van de speler gelegd worden. Ze mogen niet in een rij gelegd worden, waar al tegels van de andere kleur liggen. Ze mogen dus alleen in een lege rij gelegd worden, of in een rij die al gedeeltelijk is gevuld met tegels van diezelfde kleur. Er wordt een rij gekozen, waar de gekozen tegels van de schaal allemaal in passen, en die al zo ver mogelijk gevuld is. Als er meerdere rijen zijn die ditzelfde resultaat geven, mag daar willekeurig een rij van gekozen worden.

Als Femke bijvoorbeeld in Figuur 1 kiest voor schaal 0 en de gele tegels, dan worden de twee tegels in haar tweede rij van boven gelegd. In de eerste rij passen niet nog eens twee tegels, in de derde en de vierde rij passen ze wel, maar die zijn minder vol dan de tweede rij. En in haar laatste rij, mogen geen gele tegels gelegd worden, want daar liggen al drie blauwe tegels. We krijgen dan de stand in Figuur 2.

Femke had overigens feitelijk maar één andere keuze: ze had ook kunnen kiezen voor de *blauwe* tegels van schaal 0. De keuze van schaal 1 had geen ander spelverloop opgeleverd, omdat schalen 0 en 1 precies dezelfde tegels bevatten in Figuur 1. [1]

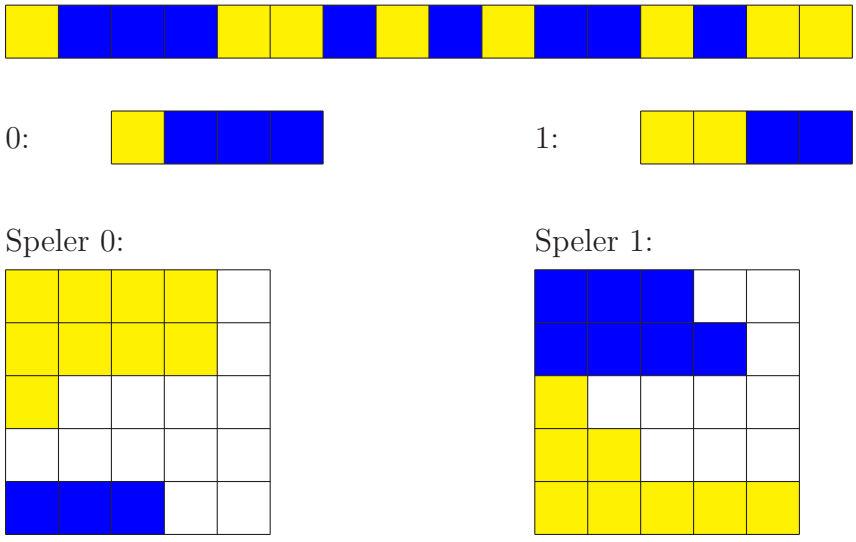

Figure 2: De stand bij het tegelspel volgend op de stand in Figuur 1, als Femke heeft gekozen voor de gele tegels van schaal 0.

In deze stand kan Lieke, die aan de beurt is, niet kiezen voor de (drie) blauwe tegels van schaal 0. Die passen namelijk in geen van haar blauwe rijen.

[1] Als bijvoorbeeld schaal 0 twee gele tegels en twee blauwe tegels bevat, en schaal 1 twee gele tegels en één blauwe tegel (omdat de pot leeg is, en schaal 1 dus niet vol is), zijn er voor het spelverloop feitelijk drie verschillende zetten: (schaal 0, geel), (schaal 0, blauw) en (schaal 1, blauw). De zet (schaal 1, geel) is feitelijk gelijk aan (schaal 0, geel).

Als de pot niet voldoende tegels bevat om de van de schaal weggehaalde tegels te vervangen, worden er zo veel mogelijk tegels uit de pot naar de schaal verplaatst, waarna de pot leeg is. Femke en Lieke kunnen dan in principe nog gewoon doorspelen, zolang geen van de spelers al haar rijen heeft gevuld, en zolang de speler die aan de beurt is, nog een geldige zet heeft. Anders spreken we van een eindstand. Als de pot en de schalen leeg zijn, dan is er geen geldige zet meer voor de speler aan de beurt, en is er dus ook een eindstand.

Merk op dat het in een eindstand mogelijk is dat de speler die niet aan de beurt is, nog mogelijke zetten zou hebben, namelijk als de speler die *wel* aan de beurt is geen zetten meer heeft. Of omgekeerd, als de speler die niet aan de beurt is, al haar rijen vol heeft, is het een eindstand, ook al zou de speler aan beurt nog zetten hebben.

In een eindstand wordt gekeken hoeveel volle rijen elke speler heeft. Als bijvoorbeeld Femke drie volle rijen heeft, en Lieke vier, dan heeft Femke verloren, met een score van $3 - 4 = -1$. Lieke heeft dan een score $4 - 3 = 1$. Een gelijkspel (score 0) is ook mogelijk bij dit spel. De spelers proberen een eindstand te bereiken waarin zij zelf een zo hoog mogelijke score hebben.

De nummering van de schalen is van belang. Ze hebben achtereenvolgens nummers $0, 1, \ldots, M - 1$.

Invoer

Een stand van het spel (waarbij mogelijk al stenen in rijen op de borden liggen) kan worden ingelezen uit een tekstbestand. Zo'n tekstbestand heeft het volgende formaat:

- Een regel met een string van letters, die de inhoud van de pot voorstelt. Als het goed is (maar dat moet je controleren), bevat de string alleen letters 'g' en 'b' (voor geel en blauw). Je mag ervanuit gaan dat deze string niet leeg is, en ook geen whitespace (b.v. spaties) bevat. Als een schaal moet worden (aan-)gevuld met tegels uit de pot, dan worden die tegels van links naar rechts uit de pot gehaald.

- Een regel met twee gehele getallen: M en N, voor het aantal schalen en het maximum aantal tegels op een schaal. De schalen worden (voor zover mogelijk) gevuld met de eerste letters uit de ingelezen pot: schaal 0 met de eerste N tegels uit de pot, schaal 1 met de volgende N tegels uit de pot, enzovoort.

- Een regel met twee gehele getallen: K en L, voor het aantal rijen op het bord van elke speler, en het aantal vakjes per rij.

- K regels overeenkomend met de K rijen van Femke (speler 0). Elke regel bevat twee gehele getallen: het huidige aantal gele tegels en het huidige aantal blauwe tegels in de corresponderende rij.

- K regels overeenkomend met de K rijen van Lieke (speler 1). Elke regel bevat twee gehele getallen: het huidige aantal gele tegels en het huidige aantal blauwe tegels in de corresponderende rij.

- Een regel met 1 geheel getal, voor de speler die aan de beurt is: 0 (voor Femke) of 1 (voor Lieke).

Getallen op een zelfde regel van de invoer worden gescheiden door een spatie. In Figuur 3 is de inhoud van het invoerbestand te zien dat overeenkomt met Figuur 1. Merk op dat in de

```
gbbggbbggbgbbbggbgbgbbgbgg
2 4
5 5
4 0
2 0
1 0
0 0
0 3
0 3
0 4
1 0
2 0
5 0
0
```

Figure 3: Inhoud van invoerbestand, overeenkomend met Figuur 1.

resulterende stand in Figuur 1 de tegels op de schalen gesorteerd zijn: ze liggen in een andere volgorde dan ze uit de pot gekomen zijn.

Bij de stand in het invoerbestand is het overigens toegestaan dat een speler meerdere korte rijen tegels van dezelfde kleur op haar bord heeft, zoals de twee korte rijen met gele tegels bij zowel Femke als Lieke in Figuur 1. Bij geldig spel zou dat nooit kunnen ontstaan, omdat die tegels dan bij elkaar in een rij of bij een andere rij zouden worden gelegd.

Voor u beschikbaar

Op de website van het vak

`https://liacs.leidenuniv.nl/~vlietrvan1/algoritmiek/`

is een skeletprogramma beschikbaar, waarin een klasse `TegelSpel` wordt gedefinieerd, die door het hoofdprogramma gebruikt wordt om het spel te spelen en experimenten te doen. Het programma wordt onder Linux gecompileerd met het commando `make`. Vervolgens kun je het met het commando `./TegelSpel` runnen. Je krijgt dan een menu aangeboden, met de keuze om een spel in te lezen uit een tekstbestand (met mogelijk al wat tegels op het bord). Bedoeling is om de `TODO`'s in de gegeven bestanden, in het bijzonder in `tegelspel.cc` en `tegelspel.h` uit te voeren.

In de bestanden `standaard.h` en `standaard.cc` zijn enkele standaardfuncties uitgewerkt: `integerInBereik` en `randomGetal`. Je kunt die gebruiken om te testen of een integer tussen bepaalde grenzen ligt (al dan niet met een foutmelding), en om een random getal tussen bepaalde grenzen te genereren.

Bij het skeletprogramma is ook een voorbeeld van een geldig invoerbestand gegeven: `spel1.txt`. Dit komt overeen met Figuur 1 en Figuur 3.

De meeste functies die je moet implementeren worden toegelicht in het skeletprogramma, speciaal in `tegelspel.h`. Goed om te weten: als er in het commentaar boven een functie `Pre:` staat, dan volgt de **preconditie** voor die functie: een aantal aannames waar je vanuit mag gaan als je in die functie binnenkomt. Je hoeft dat dus in de functie niet meer te controleren.

De gebruiker van de functie moet daar van tevoren voor zorgen. Net zo staat `Post:` voor de **postconditie** van de functie: zaken die na afloop van de functie moeten gelden. Daar moet je in de functie dus voor zorgen.
Van een aantal functies geven we nu een nadere toelichting:

- `bool TegelSpel::leesInSpel (...)`

  Deze functie leest een tegelspel in vanuit een tekstbestand. Dit bestand heeft het formaat zoals hierboven beschreven onder het kopje Invoer. Je moet nog wel controleren of het tekstbestand daadwerkelijk te openen is voor lezen, of de string die je inleest van de eerste regel de juiste letters bevat, en of de getallen die je vervolgens inleest aan bepaalde eisen voldoen (zie `tegelspel.h`).

  Uiteraard betekent *in*lezen dat het uit het bestand gelezen spel op een of andere wijze wordt opgeslagen in het betreffende object van de klasse `TegelSpel`.

  Als je in C++ een `ifstream fin` gebruikt voor je invoer-tekstbestand, dan kun je heel eenvoudig een (positief of negatief) getal inlezen, bijvoorbeeld met

  ```
  fin >> getal;
  ```

  waarbij `getal` bijvoorbeeld gedeclareerd is als:

  ```
  int getal;
  ```

  Op deze wijze wordt automatisch over spaties en newlines in het tekstbestand heengesprongen. Je hoeft het getal dus niet karakter-voor-karakter op te bouwen.

- `vector ... TegelSpel::bepaalVerschillendeZetten ()`

  Deze functie moet een vector met de verschillende zetten opleveren die de speler-aan-beurt in de huidige stand zou kunnen doen. Elke zet is een pair, bestaande uit een integer (voor de gekozen schaal) en een karakter `'g'` of `'b'`.

  Twee zetten zijn gelijk (dus **niet** verschillend), als de kleur hetzelfde is en de twee gekozen schalen hetzelfde aantal tegels van die kleur bevatten. In dat geval moet alleen de zet met **de laagst genummerde schaal** in de vector worden opgenomen. Het maakt niet uit in welke volgorde de zetten in de vector komen te staan.

- `int TegelSpel::besteScore (...)`

  Dit is de functie die verantwoordelijk is voor het brute force element van de opdracht. Zonder deze functie zal je oplossing voor de opdracht **nooit** voldoende worden, hoe goed de rest ook is.

  De functie moet de toestand-actie-boom van het spel aflopen om te bepalen wat de eindscore wordt voor de speler die aan de beurt is in de huidige stand (= de stand van de huidige recursieve aanroep), wanneer beide spelers vanaf dat moment (voor zichzelf) optimaal verder spelen.

  Anders dan bij bijvoorbeeld Nim in college 3 zijn er nu meer dan twee mogelijke uitkomsten van het spel. De score kan namelijk binnen bepaalde grenzen elk positief of negatief getal of 0 zijn. Voor de rest is het idee wel hetzelfde, en kan een recursieve functie de volgende globale opzet hebben:

```
int besteScore (...)
{
  if (eindstand())
    ...
  else
  { for alle mogelijke zetten z do
    { doeZet (z);
      score = - besteScore (...);
      unDoeZet (z);

      onthoud beste score en bijbehorende zet
    }
  }
}
```

Merk op dat de score na de recursieve aanroep met -1 wordt vermenigvuldigd. De recursieve aanroep levert namelijk de beste score voor de tegenstander op, die in de vervolgstand aan de beurt is. Winst voor de tegenstander betekent verlies voor jou, en omgekeerd.

Zoals in de Inleiding beschreven, is de score voor een bepaalde speler in een gegeven stand gelijk aan haar aantal volle rijen min het aantal volle rijen van de andere speler.

De functie `besteScore` moet ook het aantal standen bepalen dat bij het doorlopen van de toestand-actie-boom wordt bezocht. `Je mag er niet vanuit gaan` dat de parameter `aantalStanden` bij de eerste aanroep al gelijk is aan 0. Een elegante manier om dit voor elkaar te krijgen is door `besteScore` als *wrapper-functie* te gebruiken, met een recursieve hulpfunctie waar het echte rekenwerk gebeurt.

- `bool TegelSpel::doeZet (...)` en `bool TegelSpel::unDoeZet ()`

  Deze functies maken gebruik van een lijst met gedane zetten. Bedenk daarbij wat je op moet slaan over een zet, om hem later, desgewenst, weer ongedaan te kunnen maken, en terug te keren in de oude toestand. Als je een spel hebt ingelezen met `leesInBord`, hoef je alleen zetten bij te houden die na het inlezen zijn gedaan.

- `pair ... TegelSpel::bepaalGoedeZet (...)`

  Je zult merken dat het brute force doorrekenen van de complete toestand-actie-boom bij grotere of legere borden al snel te veel tijd kost. Je kunt dan ook met een *Monte Carlo methode* een goede zet bepalen. Daar is deze functie voor.

  Voor elke mogelijke zet in de huidige stand speel je het spel (na het doen van die zet) `nrSimulaties` keer uit met random toegestane zetten. Van al deze simulaties bereken je de gemiddelde eindscore voor de speler die oorspronkelijk aan de beurt was. De zet (of een zet, als er meerdere zijn) die, met deze simulaties, de hoogste gemiddelde eindscore oplevert, is de zet die geretourneerd moet worden.

  N.B.: bij het experiment (zie verderop) gaan we kijken hoe goed een 'goede zet' werkelijk is, vergeleken met een beste zet.

- `void TegelSpel::doeExperiment ()`

Code voor een experiment dat gedaan moeten worden voor het verslag. Vanuit de huidige stand van het spel moet de functie eerst met behulp van 'goede zetten' (verkregen met `bepaalGoedeZet`) het spel uitspelen tot een eindstand. Vervolgens wordt steeds een zet extra ongedaan gemaakt. Na elke zet die ongedaan wordt gemaakt, wordt `besteScore` aangeroepen, en wordt op het scherm gezet hoeveel standen daarbij worden bekeken en hoeveel tijd dit kost. Net zolang tot de oorspronkelijke stand van het spel weer verkregen is.

Met dit alles kun je zien hoe snel de hoeveelheid werk groeit als je bij `besteScore` een groter aantal zetten vooruit moet kijken.

Enkele tips bij het programmeren

- Kijk in het skeletprogramma waar allemaal `TODO` staat, voor met name de functies die geïmplementeerd moeten worden.
- In het skeletprogramma wordt gebruik gemaakt van `pair`, om een tweetal getallen in op te slaan. Als je niet weet hoe je die gebruikt, zoek het op in de C++ reference.

  Handig om te weten: je kunt een nieuw pair aanmaken met de functie `make_pair`. En als de variabele `paar` een pair voorstelt, dan kun je met respectievelijk `paar.first` en `paar.second` de afzonderlijke onderdelen van het pair benaderen.
- Het is verstandig om eerst na te denken over hoe je de inhoud van het spel (pot, schalen, rijen op de borden) op kunt slaan, en vervolgens enkele eenvoudige functies te implementeren, bijvoorbeeld de getters, `leesInSpel` en `drukAf`.
- Als je bij de vereiste controles in de memberfuncties een fout ontdekt, geef dan ook een passende foutmelding aan de gebruiker.
- Om je programma te testen (voordat je het inlevert) komen er mogelijk nog nadere instructies, op de website van het vak.

Let op: het is niet toegestaan om de headers van de gegeven public memberfuncties in `tegelspel.h` te veranderen. Dan zou onze automatische test (met een ander main programma) bij de beoordeling namelijk niet goed kunnen werken. Je mag wel functies toevoegen. Verder kan het inleveren van lelijke, niet-elegante, of erg inefficiënte code 0.5 punt aftrek opleveren.

Algemene opmerkingen

- Maak / behoud een verstandige klassenstructuur.
- Je klassen mogen alleen `public` membervariabelen en methoden hebben die buiten de klasse bekend moeten zijn. Andere membervariabelen en methoden moeten `private` zijn.
- Als er in je klassen dynamisch geheugen wordt gealloceerd, denk dan ook aan een destructor.
- Functies mogen niet te lang zijn (maximaal 35 regels).
- Gebruik constantes waar dat zinvol is. Kijk bijvoorbeeld in bestand `constantes.h` in het skeletprogramma.

- Het werkende programma mag er op het scherm eenvoudig uitzien, maar moet wel duidelijk zijn. De enige te gebruiken headerfiles zijn in principe `iostream`, `sstream`, `iomanip`, `fstream`, `cstring`, `string`, `vector`, `utility`, `set`, `unordered_set`, `cstdlib`, `ctime` en `climits`. Wil je een andere headerfile gebruiken, vraag de docent. De headerfile `algorithm` mag in ieder geval niet gebruikt worden.

- Boven elke functie moet een commentaarblokje komen met daarin een korte beschrijving van wat de functie doet. Noem daarin tevens de gebruikte parameters: geef hun betekenis en geef aan hoe ze eventueel veranderd worden door de functie. Geef bij memberfuncties ook aan wat deze met de membervariabelen van het object doen. Let verder op de layout (consequent inspringen) en op het overige commentaar bij de programmacode (zinvol en kort).

- Het programma moet onder Linux bij LIACS getest zijn en werken. Dat kan vanuit huis bijvoorbeeld op de huisuil, zie de instructie op de website.

Verslag

Het verslag moet getypt zijn in LaTeX, en moet bevatten:

- Een korte introductie, met uitleg over het spel en de opdracht.

- Een paragraaf waarin je uitlegt hoe je met behulp van de functie `besteScore` bepaalt wat de eindscore voor de huidige speler wordt als beide spelers optimaal verder spelen. Werk je met doezet/undoezet of maak je gebruik van een kopie, en waarom heb je die keuze gemaakt? Doorzoek je de complete toestand-actie-boom, of spaar je toch nog ergens werk uit?

- Een paragraaf met enkele resultaten van je programma. Wat is de uitkomst (score + beste zet) van je functie `besteScore` voor het voorbeeldspel in tekstbestand `spel1.txt`. Hoeveel standen worden voor dit voorbeeld door de functie bekeken, en hoeveel tijd vergt dat? Wat is de uitkomst van de functie bepaalGoedeScore voor dit zelfde voorbeeldspel?

  Beantwoord dezelfde vragen voor het voorbeeldspel in tekstbestand `spel3.txt`.

  Welke conclusie kun je uit bovenstaande trekken?

- Geef ook de beschrijving van een stand, met pot, schalen, twee borden en de speler-aan-beurt, waarvoor de aanroep van `besteScore` door jouw programma meer dan een miljard standen bekeek, maar die nog wel binnen een uur uit te rekenen was. Vermeld ook het aantal bekeken standen en de benodigde rekentijd. Lever bij je inzending een invoerbestand `spel2.txt` mee dat met dit bord overeenkomt.

- Een paragraaf waar je ingaat op een ander algoritme om een zet te bepalen. Je zou altijd kunnen kiezen voor een zet die een rij zo vol mogelijk maakt, bij voorkeur helemaal vol. Laat zien dat dit *gretige* algoritme niet per se een optimale zet oplevert. Doe dit door een toestand te beschrijven waar dit aan de orde is, met de mogelijke zetten voor de speler die aan de beurt is. Laten we deze speler A noemen. Welke zet levert het gretige algoritme in dit geval op? Beredeneer wat de besteScore voor speler A wordt nadat zij deze gretige zet heeft uitgevoerd. Wat zou een betere zet voor speler A geweest zijn, en wat wordt de besteScore voor haar na deze zet?

Onder 'beredeneren' verstaan we in principe niet dat je puur verwijst naar de uitkomst van `besteScore`, maar dat je laat zien wat je vanuit welke toestand kunt bereiken. Dat gaat het gemakkelijkst als de toestand die je beschrijft dicht bij een eindstand is.

- Beschrijf het experiment zoals dat wordt uitgevoerd door memberfunctie `doeExperiment`. Geef vervolgens ook de resultaten van de functie voor de voorbeeldspellen in tekstbestanden `spel4.txt` en `spel5.txt`. De resultaten die je moet geven zijn de aantallen bekeken standen en de benodigde tijd voor besteScore voor elk aantal zetten dat ongedaan is gemaakt. Zet deze resultaten in een overzichtelijke tabel. Als de aanroep van `besteScore` voor een bepaald aantal zetten meer dan vijf minuten duurt, mag je het experiment voor dat spel afbreken.[2]

  Welke conclusie(s) trek je uit de resultaten van het experiment

Je hoeft in je verslag geen 'appendix' met je complete programma (alle .cc/.h bestanden) op te nemen. We printen het toch niet meer uit.

Aanvullingen / tips / vragen

Eventuele verdere aanvullingen of tips bij de programmeeropdracht komen op de website van het vak

`https://liacs.leidenuniv.nl/~vlietrvan1/algoritmiek/`

Daar is ook een subpagina met onder andere een template voor het LaTeX-verslag. De behaalde cijfers komen te zijner tijd in Brightspace te staan.

Heb je vragen over de opdracht, dan kun je die uiteraard stellen tijdens de practicumbijeenkomsten van het vak. Je kunt ook emailen naar algoritmiek@liacs.leidenuniv.nl

In te leveren

Via Brightspace:

- je programma (alle .cc/.h bestanden en Makefile voor Linux bij LIACS)
- en een PDF van je verslag

samen in één .zip, .tgz of .tar.gz bestand. Vermeld overal duidelijk de namen van de makers.

Uiterste (!) inleverdatum: donderdag 21 maart 2024, 23.59 uur. Je kunt (vanwege de tentamenweek) maximaal drie weken te laat inleveren, maar dan verlies je wel punten. Als je uiterlijk twee weken na de deadline inlevert, gaat er 1 punt van je cijfer af. Als je meer dan twee weken, maar uiterlijk drie weken na de deadline inlevert, gaan er 2 punten van het cijfer af.

Normering: werking 5.5 punten; commentaar en layout 1 punt; modulaire opbouw en OOP 1 punt; verslag 2.5 punten.

Het is niet toegestaan om je opdracht te laten maken door een large language model als ChatGPT. Bij opvallende inzendingen kunnen de makers worden uitgenodigd om hun oplossing toe te lichten.

[2] Mocht dit al heel snel gebeuren, zodat je erg weinig resultaten hebt, dan kan dit puntenaftrek opleveren.