Programmeeropdracht 1 — Aapje Omino
Algoritmiek, voorjaar 2021

Inleiding

Femke en Lieke zijn ook buiten de baan onafscheidelijk. Femke heeft een aapjespuzzel (zie college 1 Algoritmiek) gekregen, en al gauw helpt ook Lieke mee om de puzzel op te lossen. Het valt echter niet mee, en dus gaan ze zoeken op internet. Daar ontdekken ze dat de aapjespuzzel héééél moeilijk is (om precies te zijn: NP-volledig), en dus besluiten ze om de stenen te gebruiken voor een variant van Triominos: Aapje Omino.

Spelverloop

Aapje Omino bestaat uit een rechthoekig bord met vierkante vakjes, en een aantal vierkante stenen. Een steen past precies op een vakje. Elke steen heeft aan elk van de zijden een getal staan. Stenen kunnen op aangrenzende vakjes op het bord gelegd worden, als de twee getallen aan weerszijden van de grenslijn gelijk zijn. Een voorbeeld van een geldige configuratie van stenen op het bord is te zien in Figuur 1. Als je nu een steen met getallen $(1, 7, 4, 5)$ (respectievelijk noord, oost, zuid en west) zou hebben, zou je die op vakje (1,1) neer kunnen leggen. Merk op dat stenen ook 90, 180 of 270 graden geroteerd mogen worden.

	0	1	2	3	4
0					
1	3 9 4 5		7 7 3 2	2 1 1 1	
2	3 4 3 5	5 4 8 6	6 3 2 6		
3			2 2 2 2		
4					

Figuur 1: Een voorbeeld van een Aapje Omino bord.

De eerste steen ligt op een willekeurig vakje op het bord. Femke en Lieke beginnen nu elk met een aantal stenen in de hand, terwijl de resterende stenen in een vaste volgorde in de pot zitten.

Femke is als eerste aan de beurt. Ze probeert een van haar stenen aan te leggen bij de stenen op het bord (in eerste instantie is dat dus alleen de beginsteen). Als ze meerdere mogelijkheden heeft, mag ze zelf kiezen. Als ze geen enkele steen aan kan leggen (en alleen dan) moet ze een steen uit de pot pakken. Als ze die steen alsnog aan kan leggen, moet ze dat ook direct doen. Als ze die steen niet aan kan leggen, gaat de beurt naar Lieke. Het is niet toegestaan om een steen bovenop een andere steen te leggen, of een steen op een leeg vakje te leggen dat niet in horizontale of verticale richting grenst aan een bezet vakje.

Vervolgens is Lieke aan de beurt, die volgens dezelfde regels een steen aan moet proberen te leggen. Dan Femke weer, enzovoort.

Net zo lang tot een van de vriendinnen geen steen meer over heeft, of totdat een van de twee niets meer kan doen: ze heeft nog wel stenen, maar kan die niet aanleggen, terwijl de pot al leeg is. Als we een van deze situaties bereiken, wordt gekeken hoeveel stenen beide spelers nog over hebben. Als bijvoorbeeld Femke nog twee stenen over heeft en Lieke nul, heeft Lieke een score van $+2$ en Femke een score van -2. Elke speler probeert voor zichzelf uiteraard een zo hoog mogelijke score te bereiken.

Omdat Lieke op de baan vaak verliest van Femke, wil ze nu bij dit spel graag winnen. Kun jij haar helpen om steeds de beste zet te kiezen?

Details

Het bord heeft een hoogte (aantal rijen) h en een breedte (aantal kolommen) b met $1 \leq h, b \leq 10$. Elk van de spelers begint met hetzelfde aantal stenen in de hand. De stenen hebben een vaste volgorde: steen 0 wordt direct (en zonder rotatie) op een opgegeven vakje op het bord gelegd. Steen 1 is voor Femke, steen 2 voor Lieke, steen 3 weer voor Femke, enzovoort, totdat beide spelers het juiste beginaantal stenen hebben. De overige stenen komen in de pot, en worden daar, wanneer nodig, in verder oplopende volgorde uitgehaald. Je weet dus precies welke steen je zult krijgen als je een steen uit de pot moet halen. Sterker nog: je weet ook welke stenen de andere speler in haar hand heeft.

De rijen van het bord zijn van boven naar beneden genummerd vanaf 0. De kolommen van links naar rechts vanaf 0. Een steen wordt beschreven door de vier getallen die erop staan, in de volgorde (noord, oost, zuid, west). Als je een steen neerlegt, moet je ook aangeven of je de steen roteert, met een getal uit $\{0, 1, 2, 3\}$. Hierbij is 0 = geen rotatie, 1 = rotatie van 90 graden tegen de klok in, 2 = rotatie van 180 graden, en 3 = rotatie van 270 graden tegen de klok in.

Voor u beschikbaar

Op de website van het vak

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

is een skeletprogramma beschikbaar, waarin een klasse `Aapje Omino` wordt gedefinieerd, die door het hoofdprogramma gebruikt wordt om een bord in te lezen, het spel te spelen en experimenten te doen. Er is ook al een klasse `Zet` beschikbaar, om een neergelegde steen te beschrijven met vier getallen: het nummer van de steen, de rotatie, en de rij en kolom van het vakje waar de steen is neergelegd. Het programma wordt onder Linux gecompileerd met het commando `make`. Vervolgens kun je het met het commando `./AapjeOmino` runnen. Je krijgt dan een menu aangeboden, met de keuze om het spel te spelen of experimenten te doen. Bedoeling is om de `TODO`'s in de gegeven bestanden, in eerste instantie `aapjeomino.cc` en `aapjeomino.h`, maar ook `zet.cc` en `main.cc`, uit te voeren.

In de bestanden `standaard.h` en `standaard.cc` zijn twee standaardfuncties uitgewerkt: `integerInBereik` en `randomGetal`. Je kunt die gebruiken om te testen of een integer tussen bepaalde grenzen ligt, en om een random getal tussen bepaalde grenzen te genereren.

Een aantal functies die je moet implementeren spreken voor zich, of worden toegelicht in het skeletprogramma. 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 AapjeOmino::leesIn (...)`

  Deze functie leest een Aapje Omino spel in vanuit een tekstbestand. Dit bestand begint (altijd) met drie regels met elk twee integers: eerst de hoogte en de breedte van het bord, dan het totaal aantal stenen N in het spel en het beginaantal stenen van de spelers (elke speler begint met dit aantal), daarna de positie (rij en kolom) van steen 0 op het bord. Vervolgens komt er voor elke steen een regel (in totaal N regels dus) met vier getallen: de getallen op de steen in de volgorde (noord, oost, zuid, west). Getallen die op dezelfde regel in het tekstbestand staan, worden gescheiden door een spatie.

  Je mag ervanuitgaan dat een tekstbestand er inderdaad zo uitziet. Je moet wel controleren dat de zes getallen op de eerste drie regels aan de specificaties voldoen.

  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 AapjeOmino.

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

  ```
  fin >> getal;
  ```

  waarbij `getal` bijvoorbeeld gedeclareerd is als:

  ```
  int getal;
  ```

- `vector<Zet> AapjeOmino::bepaalMogelijkeZetten ()`

  Deze methode moet een vector van mogelijke zetten retourneren voor de speler die op dit moment aan de beurt is: stenen die zij met een bepaalde rotatie op een bepaald vakje kan leggen. Het is hierbij niet nodig om equivalente zetten te filteren: als je bijvoorbeeld een steen met getallen (2,2,2,2) neerlegt maakt de rotatie niet uit. Het is geen probleem als je dan alle vier de rotaties als aparte zetten ziet.

- `vector<Zet> bepaalGoedeZetten ()`

  Je zou kunnen verwachten dat het een goede zet is, als je een steen kunt neerleggen die aan meerdere stenen op het bord grenst, zoals $(1, 7, 4, 5)$ op vakje (1,1) in Figuur 1, die dan aan drie stenen zou grenzen. Deze functie vraagt je om niet (per se) alle mogelijke zetten te bepalen, maar alleen de zetten waarbij de steen aan zoveel mogelijk bestaande stenen op het bord zou grenzen. Afhankelijk van de toestand kan dit maximale aantal buurstenen overigens ook gewoon 1 zijn.

- `int AapjeOmino::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.

  Per toestand wordt gekeken wat de mogelijkheden zijn voor de speler die aan de beurt is: welke zetten kan zij doen, eventueel nadat ze een steen uit de pot heeft gehaald. Bij elke mogelijkheid wordt de resulterende vervolgtoestand bepaald, waarna (recursief!) voor die vervolgtoestand wordt bepaald wat de eindscore zou worden voor de andere speler, die dan aan de beurt is. Voor een globale opzet van deze functie, zie de slides van college 4.

- Bij diverse memberfuncties wordt in `aapjeomino.h` gespecificeerd dat je eerst bepaalde controles moet uitvoeren. Je moet die controles dan daadwerkelijk in de functies uitvoeren, zelfs als er in `main.cc` al op wordt gelet. In onze automatische test bij de beoordeling zullen we namelijk met een andere main proberen wat er gebeurt met de functies als niet aan de voorwaarden is voldaan. Bijvoorbeeld als we bij `doeZet` een ongeldige zet proberen.

- `bool AapjeOmino::genereerRandomSpel (...)`

  Genereert een random spel, of eigenlijk random stenen voor een nieuw te beginnen spel. Handig voor het doen van experimenten.

- `doeexperimenten()`

  Code voor het experiment dat gedaan moeten worden voor het verslag. Dan hoef je dat experiment niet handmatig te doen, en hoeven de nakijkers ook niet alles handmatig te controleren. Zie verderop.

Enkele tips bij het programmeren

- Kijk in het framework waar allemaal `TODO` staat, voor met name de functies die geïmplementeerd moeten worden.

- In het skeletprogramma wordt al gebruik gemaakt van `vector` en `pair`. Als je niet weet hoe je die gebruikt, zoek het op in de C++ reference. Een handige functie om een pair te maken, is de functie `make_pair`. Je kunt de twee onderdelen van een pair benaderen met `first` en `second`.

- Het is verstandig om eerst goed na te denken over hoe je het bord, de stenen en de zetten representeert, en vervolgens enkele eenvoudige functies te implementeren, bijvoorbeeld `leesIn`, `drukAf`, `eindstand`, `wisselSpeler` en `doeZet` (misschien eerst zonder alle controles).

- Als je bij de vereiste controles in de memberfuncties een fout ontdekt, geef dan ook een foutmelding aan de gebruiker.

- Om je programma te testen zijn er vier .txt bestanden met in te lezen spellen beschikbaar: `ao1.txt` tot en met `ao4.txt`. De stenen in `ao4.txt` komen overeen met het bord in Figuur 1. Op de website van het vak komen enkele resultaten beschikbaar bij deze vier spellen.

Let op: het is niet toegestaan om de headers van de gegeven public memberfuncties in `aapjeomino.h` en `zet.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 0.25 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, zoals ook in het skeletprogramma al gebeurt.
- Het werkende programma mag er op het scherm eenvoudig uitzien, maar moet wel duidelijk zijn. De enige te gebruiken headerfiles zijn in principe iostream, iomanip, fstream, cstring, string, vector, set, unordered_set, utility, cstdlib en ctime. Wil je een andere headerfile gebruiken, vraag de docent. De headerfile algorithm mag in ieder geval niet gebruikt worden.
- Boven elke functie moet een commmentaarblokje 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 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, voor de functie `bepaalMogelijkeZetten`, bepaalt wat de mogelijke zetten voor de huidige speler zijn: wat loop je allemaal af, wat controleer je, en in welke volgorde?

- 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. Vertel bijvoorbeeld wat je doet als er (ook na het uit de pot halen van een steen) geen zetten mogelijk zijn, wat je doet als er wel zetten mogelijk zijn. Werk je met doezet/undoezet of maak je gebruik van een kopie, en waarom heb je die keuze gemaakt?

- Een paragraaf waarin je met behulp van een voorbeeld laat zien dat een 'goede zet' (zoals we die gedefinieerd hebben voor `bepaalGoedeZetten`) soms slechter uitpakt dan een 'niet-goede zet.' Teken daarbij een stukje van een toestand-actie-boom, dat begint in de toestand waarin je beide zetten kunt doen, en werk die volledig uit tot de resulterende eindtoestanden met eindscores. **Dit stukje toestand-actie-boom moet niet meer dan tien toestanden bevatten.**

- Een paragraaf met een beschrijving en de resultaten van een eenvoudig experiment. Bij dit experimenten gaan we bekijken tot hoeveel stenen het berekenen van de beste score nog lukt binnen een redelijke tijd, te beginnen bij acht stenen tot maximaal dertig stenen. En of we ook bij random spellen (niet alleen bij een geconstrueerd voorbeeld, zie het vorige puntje) zien dat 'goede zetten' toch slechter kunnen zijn dan de beste zet.

  Genereer daartoe voor elk aantal stenen $N = 8, 9, 10, \ldots, 30$ tien keer een random spel, met de volgende andere parameters:

  - een 7×7 bord,
  - grofweg $N/4$ stenen in de hand van elk van de spelers bij het begin van het spel, dus grofweg $N/2$ stenen in de pot,
  - positie (3,3) voor de startsteen,
  - op elke steen vier random getallen tussen 1 en N, inclusief 1 en N zelf (zo komt elk getal gemiddeld vier keer voor).

  Na het genereren van een random spel vraag je om de beste score voor speler 1 bij optimaal spel van beide spelers. En je kijkt wat de score voor speler 1 wordt, als zij steeds een willekeurige 'goede zet' (bijvoorbeeld de eerste de beste 'goede zet') speelt, terwijl speler 2 steeds een beste zet speelt, bepaald met `besteScore`.

  Bereken van de tien runs voor een aantal stenen N, gemiddeldes voor

  - de eindscore voor speler 1 wanneer beide spelers optimaal spelen,
  - het aantal standen dat bekeken is om deze beste score te berekenen
  - de hoeveelheid tijd die daarvoor nodig was
  - de score voor speler 1 wanneer zij steeds een willekeurige 'goede zet' speelt, terwijl speler 2 steeds een beste zet speelt.

  Dit (complete) experiment moet uitgevoerd worden met de functie `doeexperimenten` in `main.cc`.

  Zet deze gemiddeldes in een overzichtelijke tabel in het verslag, met een toelichting. Kun je inderdaad zien dat een 'goede zet' niet per se optimaal is?

  Als experimenten voor een bepaalde waarde van N te lang duren, mag je de experimenten afbreken. Geef dan in het verslag resultaten voor de experimenten die je hebt kunnen

uitvoeren. Als je hierdoor erg weinig resultaten hebt, kan dat overigens wel puntenaftrek opleveren.

- Een 'appendix' met je complete programma (alle .cc/.h bestanden).

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 en de vragenuren 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 (inclusief het programma!)

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

Uiterste (!) inleverdatum: maandag 29 maart 2021, 23.59 uur.

Normering: werking 6 punten; commentaar en layout 1 punt; modulaire opbouw en OOP 1 punt; verslag 2 punten.