Programmeeropdracht 2 — Stenen leggen
Algoritmiek voorjaar 2024, Universiteit Leiden

Inleiding

Alex is (tijdelijk?) op een zijspoor gerangeerd, en heeft van zijn oude vriend Michael een spelletje gekregen om de tijd te doden. Het spel bestaat uit een rechthoekig bord met $h \times b$ vierkante vakjes, en stenen van allerlei vormen. Een steen is precies zo groot dat hij een geheel aantal vakjes van het bord bedekt. Alle stenen bij elkaar zouden precies het complete bord moeten bedekken. De vraag is alleen: welke steen moet waar? Om zijn kwaliteiten te tonen, wil Alex de stenen netjes op het bord hebben liggen, als hij zich weer bij de club meldt. Maar er zijn zoveel mogelijkheden voor elk van de stenen! Kun jij Alex helpen?

Specificatie

Het speelbord van Alex kent h (met $2 \leq h \leq 20$) rijen en b (met $2 \leq b \leq 20$) kolommen met vakjes. Rijen zijn van boven naar beneden genummerd van 0 tot en met $h-1$; kolommen van links naar rechts van 0 tot en met $b-1$, zie Figuur 1. Het speelbord wordt gerepresenteerd als een tweedimensionaal array van integers. Bij het begin van het spel is het bord leeg.

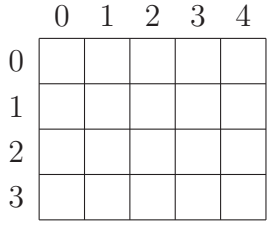

Figure 1: Een leeg 4×5 bord, met rij- en kolomnummers aangegeven.

Bij het speelbord horen S stenen, met vaste nummers $1, 2, \ldots, S$, in de volgorde waarin de stenen in de invoer staan (straks meer over de invoer). Een mogelijke verzameling stenen is weergegeven in Figuur 2.

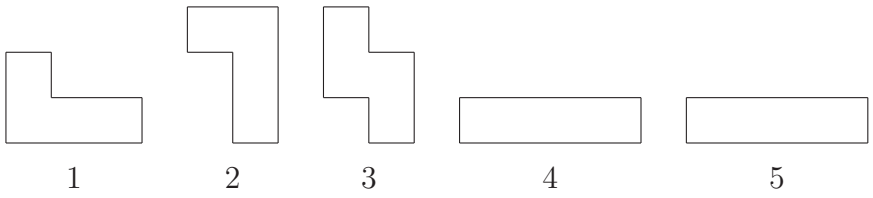

Figure 2: Een voorbeeld van vijf stenen, met vaste nummers $1, 2, 3, 4, 5$.

Stenen kunnen in acht oriëntaties op het bord gelegd worden: vier mogelijke rotaties en ook nog gespiegeld (alsof je hem ondersteboven legt).

De oriëntaties hebben een vaste nummering: Oriëntatie 0 is de oorspronkelijke oriëntatie, zoals die in de invoer staat. Oriëntaties 1, 2 en 3 ontstaan door rotatie over 90 graden, 180 graden en 270 graden, met de klok mee. Oriëntatie 4 ontstaat door de oorspronkelijke oriëntatie in de horizontale as te spiegelen. Oriëntaties 5, 6 en 7 ontstaan dan weer door rotatie over 90 graden,

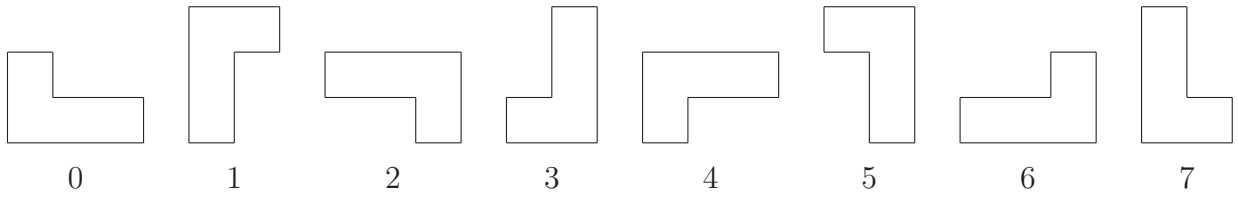

Figure 3: De acht mogelijke oriëntaties van steen 1 uit Figuur 2.

180 graden en 270 graden, met de klok mee. Ter illustratie bevat Figuur 3 de oriëntaties voor steen 1 uit Figuur 2.

Een bord met een aantal stenen daarop wordt weergegeven met het nummer van elke gelegde steen op de door die steen bedekte vakjes. **Je kunt de volgorde waarin de stenen zijn gelegd hier niet meer uit aflezen!** Elke steen kan hoogstens één keer op het bord liggen. Een voorbeeld van een gedeeltelijke bedekking van een 4×5 bord met stenen uit Figuur 2 is te zien in Figuur 4.

4			3	1
4		3	3	1
4		3	1	1
4				

Figure 4: Een 4×5 bord met drie stenen uit Figuur 2 erop.

Het is in deze opdracht onder andere de bedoeling om de gebruiker het spel te laten spelen. De gebruiker moet daartoe aan kunnen geven welke steen hij in welke oriëntatie waar op het bord wil neerleggen. Daarbij **moet** je de notatie gebruiken die we nu gaan uitleggen:

- Elke steen heeft op het bord een bounding box van $m_i \times n_i$ vakjes. Bijvoorbeeld steen 1 op het bord van Figuur 4 heeft een bounding box met vakjes $(0,3),(0,4),(1,3),(1,4),$ $(2,3),(2,4)$.
- We noteren nu een steen die op het bord gelegd wordt met vier getallen (rij,kolom,steennr, orient), waarbij (rij,kolom) het linkerbovenvakje is in de bounding box waar de steen terecht komt, steennr het nummer van de steen is, en orient de gebruikte oriëntatie. Steen 1 in het bord van Figuur 4 noteren we dus met $(0,3,1,3)$.

Wanneer de gebruiker met de beschreven notatie aangeeft dat hij ergens een steen wil leggen, moet je programma dat verwerken.

Behalve dat de gebruiker zelf stenen kan leggen op het bord, moet je programma ook, uitgaande van de huidige stand, (zo mogelijk) een complete oplossing van het spel (een bedekking van het bord met alle stenen) en het aantal verschillende complete oplossingen kunnen bepalen.

Invoer

De stenen van het spel worden ingelezen uit een tekstbestand. Dit tekstbestand ziet er als volgt uit:

- Een regel met een integer S: het aantal stenen.
- Vervolgens de beschrijving van S stenen. Voor elke steen i:
  - Een regel met twee integers m_i en n_i, gescheiden door een spatie: het aantal rijen en kolommen in de beschrijving.
  - m_i regels met elk n_i karakters: `X` (als dat vakje door de steen bedekt wordt) of `.` (als dat vakje niet door de steen bedekt wordt).

Er zou moeten gelden dat $1 \leq S \leq 20$ en dat $1 \leq m_i \leq 5$ en $1 \leq n_i \leq 5$. Je mag ervanuitgaan dat de bedekte vakjes een samenhangend geheel vormen: ze sluiten of horizontaal of verticaal op elkaar aan. Ook bevat de beschrijving geen overbodige rijen of kolommen, dat wil zeggen: geen rijen of kolommen met allemaal onbedekte vakjes.

Steen 1 in Figuur 2 wordt dus als volgt beschreven:

```
2 3
X..
XXX
```

Voor u beschikbaar

Op de website van het vak

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

is een skeletprogramma beschikbaar, waarin een klasse `Steen` en een klasse `Stand` worden gedefinieerd, die door het hoofdprogramma gebruikt worden. Het programma wordt gecompileerd met het commando `make`. Vervolgens kun je het met het commando `./Spel` runnen. Na het vastleggen van de dimensies h en b van het bord en het inlezen van een bestand met stenen, krijg je dan een menu aangeboden, met de keuze om een zet te doen of oplossingen te bepalen of te tellen. Bedoeling is om de TODO's in de gegeven bestanden uit te voeren.

Bij het skeletprogramma zitten ook drie voorbeeld tekstbestanden `stenen1.in`, `stenen2.in` en `stenen3.in`. Het eerstgenoemde tekstbestand bevat de stenen uit Figuur 2.

De meeste functies die je moet implementeren worden toegelicht in het skeletprogramma, speciaal in `steen.h` en `stand.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 Stand::leesInStenen (...)`

  Deze functie leest een verzameling stenen 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, en of het aantal stenen en de dimensies van de stenen binnen de hierboven genoemde grenzen vallen.

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

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;
```

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.

- `bool Stand::bepaalOplossing (...)`

  Deze functie moet **met behulp van backtracking**, proberen om het bord vanuit de huidige stand vol te leggen met de resterende stenen. Zonder deze functie zal je oplossing voor de opdracht **nooit** voldoende worden, hoe goed de rest ook is.

  De functie moet op alle mogelijke manieren, steen voor steen op het bord leggen. Wanneer een deeloplossing niet meer kan uitgroeien tot een complete oplossing, moet de zoektocht in deze richting worden afgebroken, en een stap terug worden gedaan in de opbouw van de oplossing.

  De functie `bepaalOplossing` moet ook het aantal standen bepalen dat tijdens de zoektocht wordt bekeken. `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 `bepaalOplossing` als *wrapper-functie* te gebruiken, met een recursieve hulpfunctie waar het echte rekenwerk gebeurt.

  Tenslotte moet de functie, als er een oplossing wordt gevonden, deze in parameter `oplossing` teruggeven. Dat is een tweedimensionaal array, waarbij `oplossing[i][j]` het nummer is van de steen die het vakje in rij i en kolom j bedekt, net zoals dat in Figuur 4 voor een deeloplossing het geval is.

Symmetrieën

Bij het zoeken naar oplossingen moet je voorkomen dat je exact dezelfde stand meerdere keren genereert door de neergelegde stenen in verschillende volgordes op het bord te leggen. De stand in Figuur 4 kun je bijvoorbeeld krijgen door eerst steen 1, dan steen 3 en dan steen 4 neer te leggen, maar ook via andere volgordes. Beperk je dan tot een van die volgordes.

Ook kunnen verschillende oriëntaties van dezelfde steen feitelijk hetzelfde zijn. Dit is bijvoorbeeld het geval met oriëntaties 0 en 2 van steen 3 in Figuur 2. Probeer van dergelijke 'dubbele' oriëntaties van dezelfde steen slechts één oriëntatie te gebruiken.

Daarnaast kunnen verschillende stenen in de invoer feitelijk hetzelfde zijn: de ene is een oriëntatie van de ander. Een voorbeeld is te zien in Figuur 5(a). Als je ook met deze laatste vorm van symmetrie rekening houdt bij het zoeken naar oplossingen, en tevens uitlegt in het verslag hoe je dat doet, kun je 0.5 punt bonus verdienen.

Als je het bord een halve slag draait of spiegelt, krijg je een ander bord dat feitelijk op hetzelfde neerkomt. Verder kun je dezelfde stenen op verschillende manieren neerleggen en toch dezelfde bedekking van vakjes op het bord krijgen, zodat de mogelijke vervolgstanden ook hetzelfde zijn. Een eenvoudig voorbeeld is gegeven in Figuur 5(b). Met deze twee zaken hoef je geen rekening te houden bij het zoeken naar oplossingen.

4			3	2
4		3	3	2
4		3	2	2
4				

(a)

4			3	3
4		3	3	1
4		1	1	1
4				

(b)

Figure 5: Twee varianten van de gedeeltelijke bedekking van het 4×5 bord uit Figuur 4. (a) Steen 1 vervangen door steen 2. (b) Stenen 1 en 3 anders neergelegd.

Dom zoeken en slim zoeken

Bij het zoeken naar een complete oplossing van het spel, of het tellen van het aantal complete oplossingen, kun je 'dom' of 'slim' te werk gaan. 'Dom' is zoals hierboven beschreven: recursief stenen neerleggen, waarbij er nog wel voor wordt gezorgd dat dezelfde stand niet meerdere keren wordt gegenereerd.

Bij 'slim' zoeken doe je na elke steen die je neerlegt een extra test, waarna je de zoektocht soms al sneller kunt afkappen. Stel bijvoorbeeld dat elke steen vier vakjes bedekt, zoals in ons voorbeeld. Door een aantal stenen op het bord te leggen, kunnen de resterende, lege vakjes opgesplitst worden in verschillende, losse deelgebieden. Wanneer een deelgebied niet uit een veelvoud van vier vakjes bestaat, zal het nooit meer lukken om het bord te bedekken. Dit wordt geïllustreerd door Figuur 6. Bij slim zoeken moet je dus na iedere steen die je neerlegt kijken

	3			1
	3	3		1
		3	1	1

Figure 6: Door stenen 1 en 3 neer te leggen is een deelgebied van drie lege vakjes en een deelgebied van negen lege vakjes ontstaan. Die kunnen niet meer bedekt worden met stenen van elk vier vakjes.

of er deelgebieden ontstaan zijn (bijvoorbeeld met een depth-first search of een breadth-first search) en (in ons voorbeeld) kijken of die allemaal uit een veelvoud van vier vakjes bestaan.

In het algemeen hoeven de stenen in het spel niet per se allemaal hetzelfde aantal vakjes te bedekken. In dat geval kun je nog wel slim zoeken door naar de grootste gemene deler en het minimum van de aantallen vakjes per steen te kijken. Als je bijvoorbeeld stenen hebt die óf vier óf zes vakjes bedekken, dan zal het aantal vakjes van elk deelgebied even moeten zijn (want ggd(4,6)=2) en minstens 4. De bestanden `standaard.h` en `standaard.cc` bevatten al een functie `ggd (m,n)`.

Algemene opmerkingen

Enkele tips bij het programmeren

- Kijk in het skeletprogramma waar allemaal `TODO` staat, voor met name de functies die geïmplementeerd moeten worden.
- Het is verstandig om eerst na te denken over hoe je de stenen met hun oriëntaties op kunt slaan, en vervolgens enkele eenvoudige functies te implementeren, bijvoorbeeld de constructoren, de memberfuncties van klasse `Steen`, en de memberfuncties `getVakje`, `drukAf`, `leesInStenen` en `legSteenNeer` van klasse `Stand`.
- 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 `stand.h` en `steen.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. In dit geval houdt dat in het bijzonder in: verdeel de functionaliteit over de klassen `Steen` en `Stand`. Een object van de klasse `Steen` kan (bij een reeds geopende `ifstream`) de gegevens van een steen inlezen, en die opslaan in eigen membervariabelen. Vervolgens kan dit object deze gegevens via eigen memberfuncties toegankelijk maken voor een object van de klasse `Stand`.
- Je klassen mogen alleen `public` memberfuncties en -variabelen hebben die buiten de klasse bekend moeten zijn. Andere memberfuncties en -variabelen 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. Je hoeft niet de complete spelregels uit te leggen, en als je gebruik maakt van de notatie uit deze specificatie, hoef je die ook niet te herhalen.
- Een paragraaf waarin je uitlegt hoe je, uitgaande van een stand op het bord, alle mogelijke vervolgstanden opbouwt, zonder standen dubbel te genereren (zie onder het kopje Symmetrieën).
- Een paragraaf waarin je uitlegt hoe je bij het 'slim' zoeken de aantallen vakjes in de ontstane deelgebieden bepaalt.
- Een paragraaf waarin je beredeneert hoeveel verschillende oplossingen er zijn op het 4×5 bord met de stenen uit Figuur 2 (Je hoeft niet formeel te bewijzen dat er geen andere oplossingen zijn!)
- Een paragraaf met resultaten van je zoektocht voor de volgende instanties:
  - $m = 4$, $n = 5$ en `stenen1.in`,
  - $m = 6$, $n = 10$ en `stenen2.in`,
  - $m = 3$, $n = 20$ en `stenen2.in`,
  - $m = 5$, $n = 8$ en `stenen3.in`,
  - $m = 10$, $n = 4$ en `stenen3.in`,
  - een eigen instantie waarvoor het mogelijk is om in meer dan tien seconden, maar binnen enkele minuten met slim zoeken het aantal mogelijke oplossingen te tellen.

  Geef voor elk van deze instanties in een overzichtelijke tabel de volgende resultaten, voor zover die binnen redelijke tijd (zeg enkele minuten) door je programma te vinden zijn:
  - het aantal bekeken standen en de benodigde tijd voor het dom bepalen van een oplossing,
  - het aantal bekeken standen en de benodigde tijd voor het slim bepalen van een oplossing,
  - het aantal oplossingen,
  - het aantal bekeken standen en de benodigde tijd voor het dom tellen van het aantal oplossingen,

– het aantal bekeken standen en de benodigde tijd voor het slim tellen van het aantal oplossingen.

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.

Het is aan te raden om na het inlezen van de stenen te controleren of het totaal aantal vakjes dat door die stenen bedekt kan worden gelijk is aan het aantal vakjes op het bord. Zo voorkom je zinloze zoektochten.

Het is verder verstandig om de acht mogelijke oriëntaties direct na inlezen te bepalen, en alle oriëntaties voor alle stenen op te slaan in bijvoorbeeld een tweedimensionaal array. Dan hoef je tenminste niet tijdens het spel steeds opnieuw de verschillende oriëntaties te genereren.

Bedenk dat Alex eerst een aantal stenen op het bord kan leggen, en je daarna kan vragen om te zoeken naar een / alle oplossingen vanuit de ontstane stand. De stenen die hij al op het bord heeft gelegd, zijn niet per se de eerste stenen uit de invoer.

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 de gebruikte invoertekstbestanden
- 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 2 mei 2024, 23.59 uur. Je kunt maximaal twee weken te laat inleveren, maar dan gaat er wel per (deel van een) week een punt van het cijfer af. Zie de website voor de exacte regeling.

Normering: werking 5 punten; commentaar en layout 1 punt; modulaire opbouw en OOP 1 punt; verslag 3 punten; bonus als je ermee rekening houdt dat verschillende stenen feitelijk hetzelfde kunnen zijn: 0.5 punt.