Test Bash

Voor deze test lever je een aantal scripts af in de map `/root/testbash`. Vul nu alvast in alle scripts in deze map je naam in! Enkel deze scripts worden in acht genomen bij de verbetering. Wanneer je klaar bent, log je uit, maar laat je de computer aanstaan.

Inleiding

Iedereen kent hem wel: de **schuifpuzzel**, een vierkant of rechthoekig raam met daarin **tegels** die door elkaar worden geschoven tot ze in een welbepaalde volgorde liggen. Steeds is één **uitsparing** voorzien, zodat bij elke beurt moet gekozen worden uit maximaal vier verschuivingen: de tegel boven, onder, links van of rechts van de uitsparing neemt de plaats van de uitsparing in, en de uitsparing komt bij de volgende beurt op de plaats van die tegel te liggen. Voor deze test is het de bedoeling dat je een aantal Bash-scripts aflevert, die samen een willekeurige vierkante schuifpuzzel oplossen (indien een oplossing bestaat).

Een relatief eenvoudige manier om zoekproblemen zoals dit op te lossen, is door middel van **breedte-eerst zoeken**. Hierbij beschouwt men eerst de kandidaat-oplossing die wordt verkregen door geen enkele actie te ondernemen, vervolgens alle kandidaat-oplossingen die ontstaan door één actie, dan die door twee acties, enzovoort. Het zoeken stopt zodra een oplossing wordt gevonden of nadat alle kandidaat-oplossingen werden beschouwd. In het eerste geval weet men zeker dat de gevonden oplossing ook uit het kleinst mogelijke aantal acties bestaat. In het tweede geval bestaat er simpelweg geen oplossing. Soms wordt het zoeken ook vroegtijdig afgebroken, bijvoorbeeld wegens beperkingen qua geheugen of tijd.

Toegepast op de schuifpuzzel betekent dit dat eerst wordt gecontroleerd of alle tegels meteen al op de gewenste plaats liggen, vervolgens of de oplossing wordt gevonden door één tegel te verschuiven, dan door een tweede tegel te verschuiven, enzovoort, tot de puzzel is opgelost. Mits een oplossing ook daadwerkelijk bestaat, wordt op deze manier gegarandeerd een oplossing gevonden die uit het kleinst mogelijke aantal verschuivingen bestaat. Werden alle mogelijke verschuivingen uitgeprobeerd en leidde geen enkele hiervan tot een geldige oplossing, dan is de puzzel gegarandeerd onoplosbaar.

Bij breedte-eerst zoeken wordt doorgaans gebruikgemaakt van een **wachtrij**, waarin de nog te beschouwen **toestanden** van het probleem worden geplaatst. Voor de schuifpuzzel komt hierin initieel de **starttoestand**: de op te lossen puzzel. Die halen we ook meteen weer van de wachtrij om hem te beschouwen. Bevinden alle tegels zich op de juiste plaats, dan is de **doeltoestand** bereikt en stopt het zoeken. Indien niet, dan bepalen we alle mogelijke verschuivingen vanuit deze toestand. Steeds kunnen twee, drie of vier tegels op de plaats van de uitsparing komen. De wachtrij groeit bijgevolg aan met twee, drie of vier **opvolgers** van de starttoestand; hun onderlinge volgorde is niet van belang.

We halen nu (opnieuw) het eerste element van de wachtrij; met ‘eerste’ bedoelen we hier het element dat als eerste werd toegevoegd aan de wachtrij. Hierbij hoort een toestand die werd bekomen uit de starttoestand door één tegel te verschuiven. Ofwel komt deze toestand overeen met de doeltoestand en hebben we een oplossing bestaande uit één verschuiving gevonden, ofwel moeten we nog minstens één bijkomende tegel verschuiven. In het eerste geval zijn we uiteraard klaar met zoeken.

In het tweede geval bepalen we alle *opvolgers* van deze toestand, die allemaal overeenkomen met een kandidaat-oplossing bestaande uit twee verschuivingen. Opnieuw beschouwen we tot vier mogelijke verschuivingen, namelijk van de tegels die grenzen aan de uitsparing. Herinner je echter dat het onze bedoeling was om eerst alle kandidaat-oplossingen bestaande uit één verschuiving te onderzoeken. Zo blijven er nog maximaal drie over, en die staan nog steeds in de wachtrij. De nieuwe kandidaten, bestaande uit twee verschuivingen, voegen we daarom achteraan toe, na de reeds aanwezige.

De wachtrij is nu aangegroeid, maar vooraan staan hoe dan ook nog steeds de resterende kandidaat-oplossingen (minimaal één, maximaal drie stuks) bestaande uit één verschuiving. Opnieuw halen we dus het eerste element van de wachtrij en verwerken we dit. Analoog worden hierdoor twee tot vier nieuwe kandidaat-oplossingen bestaande uit twee verschuivingen toegevoegd aan de wachtrij.

Wanneer alle kandidaat-oplossingen bestaande uit één verschuiving werden afgehandeld, maar niet tot een opgeloste puzzel leidden, weten we dat een geldige oplossing minimaal twee verschuivingen moet tellen. De bewuste kandidaat-oplossingen bevinden zich vooraan in de wachtrij, dus we gaan verder met de verwerking daarvan. Opnieuw behandelen we het eerste element. Zo verkrijgen we ofwel een geldige oplossing die bestaat uit twee verschuivingen, ofwel kunnen we nieuwe kandidaat-oplossingen genereren die bestaan uit drie verschuivingen. Ook die plaatsen we weer achter in de wachtrij, zodat deze per kandidaat-oplossing alweer aangroeit met maximaal vier elementen.

Dit mechanisme herhaalt zich tot de puzzel is opgelost. Naarmate meer kandidaat-oplossingen in de wachtrij worden behandeld, stijgt ook het minimale aantal verschuivingen nodig om tot een geldige oplossing te komen. Bovendien produceren we voor elke kandidaat-oplossing twee tot vier nieuwe wachtrijelementen, zodat de wachtrij snel aangroeit. Het probleem wordt daardoor snel complex.

Merk op: wanneer een toestand wordt onderzocht, komt een van zijn opvolgers steeds overeen met de verschuiving die de vorige ongedaan maakt. We houden hier (voorlopig) echter geen rekening mee. Immers, een verschuiving ongedaan maken leidt tot een oplossing die uit minstens twee extra verschuivingen bestaat, en breedte-eerst zoeken beschouwt de meest optimale kandidaat-oplossingen eerst. Aangezien we geïnteresseerd zijn in de oplossing die uit de minste verschuivingen bestaat, nemen we er vrede mee dat een aantal kandidaat-oplossingen meermaals worden onderzocht.

Wat moet er nu juist in de wachtrij komen te staan? Er zijn meerdere mogelijkheden. Wij kiezen ervoor om in elk wachtrijelement steeds twee zaken bij te houden:

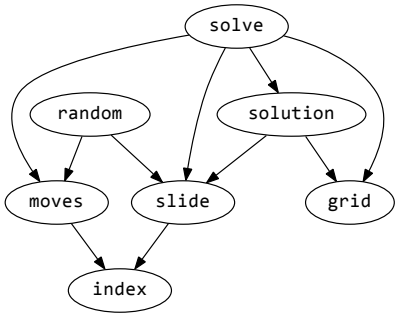
- de **toestand** t_i (met $i \geq 0$): waar elke tegel zich bevindt na de laatst uitgevoerde verschuiving;
- het **zoekpad** p_i: de opeenvolgende verschuivingen die de starttoestand t_0 transformeren in t_i.

Het zoekpad p_i hebben we nodig om na het zoeken de volledige oplossing te kunnen reconstrueren. Initieel wordt de starttoestand t_0 in de wachtrij geplaatst; het eerste zoekpad p_0 is dan ook leeg.

Tot slot maken we nog enkele afspraken:

- We beschouwen enkel vierkante puzzels. Het aantal rijen en kolommen wordt dus bepaald door één enkel natuurlijk getal r, de **rang** van de puzzel.
- We nummeren de **tegels** van 1 tot en met $r^2 - 1$. De *uitsparing* duiden we aan met 0.
- We nummeren de **posities**, d.w.z. de plaatsen waar tegels kunnen voorkomen in het raam, van 0 tot en met $r^2 - 1$, rij per rij, van links naar rechts. Voor een puzzel met $r = 3$ staan bovenaan dus van links naar rechts posities 0, 1 en 2, daaronder posities 3, 4 en 5, en tot slot onderaan posities 6, 7 en 8. *Verwar de posities niet met de tegelnummers!*
- Een puzzel is **opgelost** wanneer tegel nummer n zich op positie $n - 1$ bevindt, voor $1 \leq n \leq r^2 - 1$.

Opgave



Voor deze test lever je in totaal **zeven Bash-scripts** af, die samen een vierkante schuifpuzzel oplossen; met een *optioneel* achtste script kun je bonuspunten verdienen. Hiernaast staat afgebeeld hoe de zeven scripts elkaar oproepen. De meeste ervan vallen normaal kort uit. Schat de hoeveelheid werk in en besteed je tijd nuttig! **Maak enkel gebruik van in de labo's geziene stof.**

In de map `bin` vind je een gecompileerd equivalent van elk script. Je kunt bijgevolg op elk moment overschakelen tussen beide door het achtervoegsel `.sh` weg te laten uit de opdracht. Zo kun je `grid.sh` bijvoorbeeld vervangen door `bin/grid`. *Vergelijk de uitvoer hiervan met jouw versie!*

INDEX.SH

Dit hulpscript bepaalt de positie waarop een gegeven zoekstring voor het eerst voorkomt in een lijst met stringwaarden. De posities worden geteld vanaf nul. Voor de zoekstring `bar` en de lijst met waarden `"foo bar 42" baz bar bar` geeft het script dus als uitvoer het getal `2`. Wordt de zoekstring niet gevonden, dan is de uitvoer `-1` en de *exit status* `1`. Het script wordt opgeroepen als volgt:

```
$ ./index.sh zoekstring waarde0 waarde1 waarde2 ...
```

De zoekstring moet verplicht worden opgegeven, maar mag wel de lege string zijn. Ontbreekt deze, dan geeft het script een foutmelding. De lijst met stringwaarden mag eveneens leeg zijn (dus uit nul elementen bestaan); in dit geval is de (standaard)uitvoer van het script steeds `-1`.

GRID.SH

Dit script verdeelt tekst gelijkmatig over het scherm door een vast ***`aantal_waarden_per_regel`*** af te drukken. Elke waarde neemt **exact** ***`breedte`*** karakters in. De syntaxis is als volgt:

```
$ ./grid.sh aantal_waarden_per_regel breedte waarde1 waarde2 ...
```

Hieronder vind je een voorbeeld van een `grid.sh`-opdracht en zijn uitvoer. De grijze markeringen dienen louter ter verduidelijking. Merk op dat te korte waarden *rechts* worden uitgelijnd (door toevoeging van spaties vooraan), terwijl te lange waarden worden afgeknot tot ***`breedte`*** karakters. Tussen de waarden staan geen extra spaties. De uitvoer wordt afgesloten met exact één *newline*-karakter.

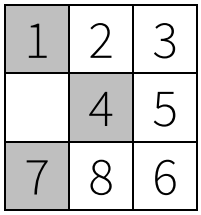
```
$ ./grid.sh 5 3 abcdef GH ijklm N opq RST uvwx YZ
abc GHijk  Nopq
RSTuvw YZ
```

De eerste twee argumenten zijn verplicht, bestaan uitsluitend uit cijfers en zijn strikt positief. Indien niet, geeft het script een foutmelding. De lijst met waarden mag wel leeg zijn.

MOVES.SH

Dit script bepaalt de mogelijke tegelverschuivingen voor een gegeven toestand van de schuifpuzzel. Als uitvoer geeft het de *posities* van de tegels die verschoven kunnen worden, op één regel, gescheiden door spaties. Het script produceert dus als uitvoer minimaal twee, maximaal vier getallen, die elk een positie grenzend aan de uitsparing aanduiden. De volgorde hiervan is niet van belang.

De invoer van `moves.sh` gebeurt door middel van opdrachtlijnargumenten. Eerst volgt de *rang* van de puzzel. Vervolgens komen de nummers van de *tegels*, rij per rij, van links naar rechts. Voor de puzzel met *rang* 3 die hiernaast staat afgebeeld krijg je dus volgende in- en uitvoer:



```
$ ./moves.sh 3 1 2 3 0 4 5 7 8 6
0 4 6
```

Om de tegels grenzend aan de uitsparing te bepalen, heb je natuurlijk eerst de positie van de uitsparing nodig. Maak gebruik van `index.sh` om de index van het cijfer `0` in de lijst te bepalen.

Ga ervan uit dat alle argumenten correct worden opgegeven. Invoercontrole is niet nodig.

SLIDE.SH

Dit script bepaalt de toestand van een puzzel na het verschuiven van een tegel. Zoals bij `moves.sh` bestaat de argumentenlijst uit de *rang* van de puzzel gevolgd door de opeenvolgende nummers van de tegels; hieraan wordt tot slot ook de *positie* van de te verplaatsen tegel toegevoegd. Als uitvoer komen de nummers van de tegels in hun nieuwe volgorde. Indien men bij bovenstaande puzzel de tegel met nummer 7 op positie 6 wenst te verplaatsen, worden de in- en uitvoer dus:

```
$ ./slide.sh 3 1 2 3 0 4 5 7 8 6 6
1 2 3 7 4 5 0 8 6
```

Net als `grid.sh` moet ook `slide.sh` de positie van de uitsparing vooraf bepalen. Maak daarom ook in dit script gebruik van `index.sh` om de index van het cijfer `0` te bepalen.

Ga ervan uit dat alle argumenten correct worden opgegeven. Invoercontrole is niet nodig.

SOLUTION.SH

Dit script krijgt een puzzel en een lijst met opeenvolgende verschuivingen mee, en drukt de toestand van de puzzel na elke verschuiving af. Het script `solve.sh`, dat in de volgende alinea wordt besproken, maakt gebruik van `solution.sh` om de opeenvolgende stappen van een oplossing te beschrijven.

Zoals bij `moves.sh` bestaat de argumentenlijst van `solution.sh` uit de *rang* van de puzzel gevolgd door de lijst met nummers van de tegels. Hierop volgt dan het volledige *zoekpad* van de starttoestand t_0 naar de eindtoestand, voorgesteld als een lijst met *posities* van te verschuiven tegels. Deze lijst wordt gescheiden door spaties, zodat elk element ervan een afzonderlijk opdrachtlijnargument vormt.

Om de toestand na elke verschuiving te bepalen, doet `solution.sh` een beroep op `slide.sh`. Dit hulpscript wordt dus opgeroepen voor elke verschuiving in het zoekpad.

Als uitvoer produceert `solution.sh` een netjes uitgelijnde weergave van de toestand van de puzzel na elke verschuiving. Verwezenlijk dit door bij het uitschrijven gebruik te maken van `grid.sh`, en dit met een vaste kolombreedte van 4 karakters.

De verwachte uitvoer van `solution.sh` neemt veel plaats in beslag en nemen we daarom niet op. In plaats daarvan raden we aan om onderstaande opdrachten uit te voeren en de werking te bestuderen. De eerste opdracht schrijft een starttoestand uit; de tweede beschrijft twee verschuivingen.

```
$ bin/grid     3 4 1 2 0 4 5 3 7 8 6
$ bin/solution 3   1 2 0 4 5 3 7 8 6 5 8
```

Ga ervan uit dat alle argumenten correct worden opgegeven. Invoercontrole is niet nodig.

SOLVE.SH

Dit script lost een gegeven puzzel op door gebruik te maken van *breedte-eerst zoeken*, zoals beschreven in de inleiding. Het steunt hiervoor op `grid.sh`, `moves.sh`, `slide.sh` en `solution.sh`.

Het script wordt steeds opgeroepen zonder opdrachtlijnargumenten. De starttoestand t_0 wordt via *standaardinvoer* doorgegeven. Op de eerste regel staat de *rang* van de puzzel. Vervolgens komen de nummers van de tegels op de rijen ervan, gescheiden door spaties; elke regel van het bestand beschrijft één rij. Ziehier een voorbeeld van een puzzel en de daarbij horende invoer van `solve.sh`:

1	2	3
	4	5
7	8	6

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

Ga na of de invoer geldig is: de *rang* van de puzzel bestaat enkel uit cijfers en is strikt positief, het aantal kolommen op elke regel komt hier telkens mee overeen, elk tegelnummer is een strikt positief natuurlijk getal kleiner dan r^2, en elk van de tegelnummers komt *exact* één keer voor in de invoer. Wordt niet aan één of meer vereisten voldaan, dan sluit het script af met een gepaste foutmelding.

Druk na het inlezen van de starttoestand de puzzel netjes uitgelijnd af. Maak hiervoor gebruik van `grid.sh` met een vaste kolombreedte van 4 karakters, net als bij `solution.sh`.

Vervolgens initialiseer en verwerk je de *wachtrij* tot de eerste geldige oplossing wordt gevonden óf tot alle mogelijke verschuivingen werden uitgeprobeerd. Neem de inleiding grondig door en bepaal zelf een gepaste Bash-gegevensstructuur voor de wachtrij en haar elementen. **Maak vooraf een schets van de gegevensstructuur en beschrijf deze kort in commentaar.**

Baseer je implementatie van breedte-eerst zoeken op onderstaande pseudocode. Maak daarbij gebruik van de scripts die in de rechterkolom vermeld staan.

```
element ← { toestand: starttoestand, zoekpad: [ ] }
wachtrij ← [ element ]

while wachtrij niet leeg
    element ← shift( wachtrij )
    t ← element.toestand
    p ← element.zoekpad
    if t = doeltoestand
        schrijf-oplossing-uit( t, p )          d.m.v. solution.sh
        break
    end if
    foreach v ∈ mogelijke-verschuivingen( t )   d.m.v. moves.sh
        t' ← pas-verschuiving-toe( t, v )       d.m.v. slide.sh
        p' ← push( p, v )
        opvolger ← { toestand: t', zoekpad: p' }
        wachtrij ← push( wachtrij, opvolger )
    end foreach
end while
```

Wanneer geen geldige oplossing werd gevonden, d.w.z. indien de `break`-instructie niet werd bereikt, schrijft het script een gepaste melding uit en sluit het af met als *exit status* 1.

In de submap `output` vind je een aantal opgeloste puzzels, waarmee je je script(s) kunt testen. Je vindt er per *rang* een submap, met daarin opnieuw een submap per aantal verschuivingen in de optimale oplossing. Zo bevat de map `output/6/3` dus informatie over een puzzel met rang 6 die oplosbaar is in 3 verschuivingen. Telkens staat de starttoestand, dus de invoer van `solve.sh`, in `puzzle.txt` en de oplossing, dus de uitvoer, in `solution.txt`. Elke oplossing is uniek.

Doordat breedte-eerst zoeken een eerder naïeve manier is om schuifpuzzelproblemen te lijf te gaan, kan het oplossen van complexe puzzels erg veel tijd in beslag nemen. De uitvoeringstijd zal *niet* in acht worden genomen bij de beoordeling; een *correcte* implementatie is dus voldoende. In het bestand `output/timing.txt` vind je niettemin per probleem de tijd (gemeten in seconden) terug die onze bescheiden testmachine nodig had om de gegeven puzzels op te lossen.

RANDOM.SH

Om de voorgaande scripts uitvoerig te testen, kunnen willekeurige puzzels handig van pas komen. Daarom construeert `random.sh` bij elke uitvoering één willekeurige, oplosbare puzzel.

Als opdrachtlijnargumenten krijgt dit script de *rang* van de te genereren puzzel en het maximale aantal verschuivingen *m* om tot de oplossing van deze puzzel te komen. Volgende opdracht produceert dus een puzzel met rang 8, die oplosbaar is in maximaal 20 verschuivingen:

```
$ ./random.sh 8 20
```

Het script vertrekt van de oplossing en voert hier vervolgens *m* willekeurig gekozen verschuivingen op uit. Daarbij kan het voorkomen dat een voorgaande verschuiving toevallig ongedaan wordt gemaakt, wat het aantal verschuivingen in de optimale oplossing zal verminderen. Dit is ook de reden waarom *m* het *maximale* aantal verschuivingen tot de doeltoestand voorstelt en niet noodzakelijk het exacte.

Om de mogelijke verschuivingen te bepalen, doet het script een beroep op het hulpscript `moves.sh`. Vervolgens selecteert het hieruit telkens willekeurig één verschuiving, die meteen ook wordt toegepast door `slide.sh` op te roepen. Deze stappen worden *m* keer herhaald.

De uitvoer van het script beantwoordt aan het invoerformaat van `solve.sh`: op de eerste regel komt (enkel) de *rang* van de puzzel, en op de daarop volgende regels staan de nummers van de tegels op de overeenkomstige rijen, gescheiden door spaties.

Ga ervan uit dat beide argumenten correct worden opgegeven. Invoercontrole is niet nodig.

BONUSVRAAG: SOLVE2.SH

Indien je besluit om ook deze vraag op te lossen, vertrek dan van een **kopie** van `solve.sh` genaamd `solve2.sh`. Zo willen we vermijden dat je een werkende versie kwijtspeelt. Bijgevolg mag je beide scripts indienen; deze zullen afzonderlijk worden geëvalueerd.

Zoals vermeld in de inleiding, beschouwt breedte-eerst zoeken een tamelijk groot aantal kandidaat-oplossingen meermaals. Bedenk daarom een manier om ervoor te zorgen dat elk van deze kandidaat-oplossingen maximaal één keer in de wachtrij belandt. Door deze optimalisatie zou je script aanzienlijk sneller moeten werken dan de eerste versie. De oplossing wijzigt hierdoor echter niet! De uitvoerings-tijden van onze implementatie van `solve2.sh` vind je in `output/timing2.txt`.

Controleer nogmaals of je je naam hebt ingevuld in alle scripts in `/root/testbash`!

Klad