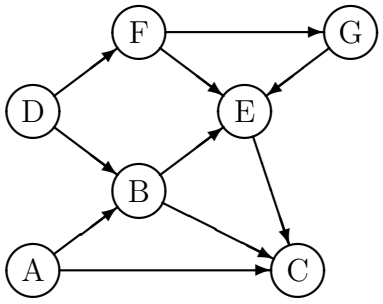
Hertentamen Algoritmiek
Maandag 6 juli 2020, 14.15 – 17.15 uur

Waneer er in een opgave gevraagd wordt om uitleg, toelichting of motivatie van je antwoord, is het belangrijk om die ook te geven.
De aantallen punten die bij het begin van elke opgave vermeld worden, zijn indicatief. Ze kunnen dus nog iets wijzigen.

---

1. [18 pt] Beschouw de volgende gerichte acyclische graaf G:



Bij onderdelen (a) en (b) gaan we respectievelijk een breadth-first search en een depth-first search uitvoeren in G. In beide gevallen starten we de zoektocht in knoop A. Als een knoop meerdere buren heb, lopen we die in alfabetische volgorde af. Als de zoektocht vanuit knoop A klaar is, en er zijn nog onbezochte knopen, dan starten we een zoektocht in de eerstvolgende, nog niet bezochte knoop in alfabetische volgorde. Enzovoort.

(a) Voer een breadth-first search (BFS) uit in G. Teken als je antwoord op dit onderdeel de resulterende BFS boom / bos. Geef daarin aan in welke volgorde de knopen in de rij zijn gezet.

(b) Voer een depth-first search (DFS) uit in G. Teken als je antwoord op dit onderdeel de resulterende DFS boom / bos. Geef daarin aan in welke volgorde de knopen voor de eerste keer worden bereikt (en op de stapel gezet), en in welke volgorde ze helemaal zijn afgehandeld (van de stapel worden gehaald).

(c) Een van de twee volgordes van de knopen die je vindt bij een depth-first search in een gerichte acyclische graaf kun je gebruiken om een topologische ordening van de graaf te bepalen. Welke van de twee volgordes is dat in het algemeen?

Geef ook de resulterende topologische ordening voor onze graaf G.

2. [24 pt] Bij het knapzakprobleem hebben we een verzameling objecten i ($1 \leq i \leq N$), elk met een gewicht w_i en een waarde v_i. Verder hebben we een knapzak met een capaciteit W. In deze knapzak kunnen we objecten uit onze verzameling stoppen, zolang het totaalgewicht niet groter wordt dan W. Doel is om een deelverzameling van (verschillende) objecten in de knapzak te stoppen met een zo groot mogelijke totale waarde.

   (a) We hebben tijdens het college Algoritmiek verschillende ontwerptechnieken toegepast op het knapzakprobleem. Beschrijf (per onderdeel in een of enkele zinnen)

      i. een kenmerkend verschil tussen exhaustive search en backtracking voor het knapzakprobleem,

      ii. een kenmerkend verschil tussen backtracking en brand-and-bound voor het knapzakprobleem,

      iii. een kenmerkend verschil tussen een gretig algoritme enerzijds, en de andere drie hierboven genoemde technieken anderzijds, voor het knapzakprobleem.

   Een voorbeeld van een knapzakprobleem met vier objecten is:

object i	w_i	v_i	v_i/w_i
1	9	81	9
2	8	64	8
3	3	21	7
4	7	42	6

   W = 16

   Tijdens het college hebben we een branch-and-bound algoritme voor het knapzakprobleem behandeld. Hierbij wordt gebruikt dat de objecten gesorteerd zijn op de gemiddelde-waarde-per-gewicht v_i/w_i, van groot naar klein. Iedere stap van het algoritme wordt een object wel of juist niet aan de knapzak toegevoegd, te beginnen bij object 1.

   Uitgaande van een deeloplossing, is een bovengrens voor de waarde van elke (complete) uitbreiding van de deeloplossing als volgt te berekenen:

   - Bij aanvang, in de begintoestand: $W * (v_1/w_1)$.
   - Bij een algemene deeloplossing (als we net object i wel of niet hebben gekozen, met $1 \leq i \leq N-1$): $v + (W - w) * (v_{i+1}/w_{i+1})$, met v de totaalwaarde van de reeds gekozen objecten en w het totaalgewicht daarvan.

   (b) Pas het branch-and-bound algoritme met de hierboven gegeven bovengrens toe op het voorbeeld en teken de bijbehorende state-space-tree, met bij elke knoop (deeloplossing) de relevante informatie (waaronder ook de opbouw van de bovengrens). Geef ook aan in welke volgorde de knopen zijn aangemaakt, welke knopen gesnoeid worden en waarom.

   Wat is dus de optimale oplossing?

3. [30 pt] In een flatgebouw van breedte B zijn twee liften, aan de linker kant en aan de rechterkant van het gebouw, dus op afstand B van elkaar. Op elke verdieping i van het flatgebouw ($i \geq 1$) zijn twee adressen waar je een pakketje moet bezorgen, op respectievelijk afstand $A[i][0]$ en afstand $A[i][1]$ van de linker lift.

   Je moet alle pakketjes bezorgen, eerst de pakketjes op verdieping 1, dan de pakketjes op verdieping 2, enzovoort. Je wilt hierbij echter zo min mogelijk lopen: van lift, naar adres, naar volgend adres, naar lift, enzovoort. Op de begane grond hoef je niet te lopen, want je kunt je bakfiets met pakketjes bij elk van de liften voor de ingang parkeren. Zoals al aangegeven, hoef je op de begane grond ook geen pakketjes te bezorgen.

   De vraag is nu wanneer je de linker lift moet nemen om van verdieping $i-1$ naar verdieping i te gaan, en wanneer de rechter lift. En welke lift je moet nemen voor de weg terug naar beneden.

   Stel bijvoorbeeld dat $B = 10$, en dat array A er als volgt uit ziet:

i	$A[i][0]$	$A[i][1]$
4	5	9
3	7	6
2	3	8
1	2	4

   Als je steeds de linker lift neemt, moet je op verdieping 1 in totaal 8 meter lopen: van de lift, naar adres 2, naar adres 4, en weer terug naar de lift. Vervolgens loop je op verdieping 2 in totaal 16 meter, op verdieping 3 in totaal 14 meter, en op verdieping 4 in totaal 18 meter. Totaal: $8 + 16 + 14 + 18 = 56$ meter. Merk op dat $A[i][0]$ en $A[i][1]$ niet per se gesorteerd zijn.

   Als je steeds de rechter lift neemt, loop je in totaal: $16 + 14 + 8 + 10 = 48$ meter. Als je echter begint met de linker lift, en op verdieping 2 naar de rechter lift loopt, en vervolgens rechts blijft, loop je in totaal: $8 + 10 + 8 + 10 = 36$ meter. (We negeren hier dat je, eenmaal terug op de begane grond, weer naar je bakfiets bij de linker lift moet lopen.)

   Om de minimale loopafstand te bepalen maken we gebruik van de grootheden $L(i)$ en $R(i)$, die als volgt gedefinieerd zijn:

   $L(i)$ is de minimale loopafstand om de pakketjes op verdiepingen 1 t/m i te bezorgen en aan het eind (weer) bij de linker lift op verdieping i te zijn. $R(i)$ is de minimale loopafstand om de pakketjes op verdiepingen 1 t/m i te bezorgen en aan het eind (weer) bij de rechter lift op verdieping i te zijn.

   (a) Beredeneer dat $L(i)$ voldoet aan de volgende recurrente betrekking:

$$L(i) = \begin{cases} 0 & \text{als } i = 0 \\ \min \left\{ \begin{array}{c} L(i-1) + 2 * \max\{A[i][0], A[i][1]\}, \\ R(i-1) + B \end{array} \right\} & \text{als } i \geq 1 \end{cases}$$

   Hint: bedenk bij het basisgeval wat $L(0)$ voorstelt.
   Overigens voldoet $R(i)$ aan de volgende recurrente betrekking:

$$R(i) = \begin{cases} 0 & \text{als } i = 0 \\ \min \left\{ \begin{array}{c} L(i-1) + B, \\ R(i-1) + 2 * (B - \min\{A[i][0], A[i][1]\} \end{array} \right\} & \text{als } i \geq 1 \end{cases}$$

   Vanwege symmetrie hoef je dat echter niet te beredeneren.

(b) Geef een C++ functie `int afstand (int n, int B, int A[][2])` die met behulp van bottom-up dynamisch programmeren, en met gebruikmaking van de bij (a) gegeven recurrente betrekkingen, de minimale loopafstand bepaalt bij een flat van breedte B, met hoogste verdieping n, en bezorgadressen in array A. Je mag aannemen dat $n \geq 1$ en dat de adressen in A groter zijn dan 0 en kleiner zijn dan B. Maak gebruik van een passende tabel / passende tabellen om resultaten van deelproblemen in op te slaan. Je mag, desgewenst, gebruik maken van functies `min(...)` en `max(...)`, die het minimum, respectievelijk maximum van twee getallen bepalen.

N.B.: omdat op de begane grond niet bezorgd hoeft te worden, hebben $A[0][0]$ en $A[0][1]$ geen bruikbare waarde.

(c) Voer het algoritme van onderdeel (b) met de hand uit voor het geval dat $n = 5$, $B = 10$ en het volgende array A:

i	$A[i][0]$	$A[i][1]$
5	8	5
4	2	4
3	1	2
2	9	7
1	3	8

Vul de complete tabel(len) met resultaten van deelproblemen in, en maak duidelijk (voor de complete flat) wat de minimale loopafstand is.

(d) Leid stap voor stap uit je antwoord bij onderdeel (c) voor elke verdieping i af, welke lift je moet nemen van verdieping $i - 1$ naar verdieping i, en met welke lift je aan het eind weer naar beneden gaat. Noem niet alleen de liften die je gebruikt, maar maak ook duidelijk hoe je daaraan komt.

N.B.: het is de bedoeling dat je lineair door je tabel(len) loopt.

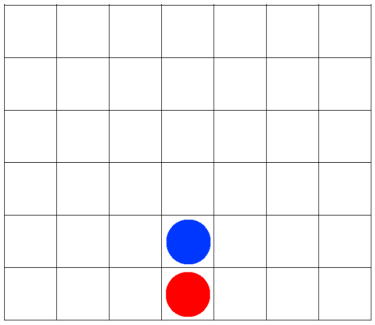
(e) Bij dit onderdeel hoef je je antwoorden **niet** te motiveren.

Wat is de tijdcomplexiteit (als functie van het aantal verdiepingen n) van je algoritme bij onderdeel (b), met bottom-up dynamisch programmeren dus?

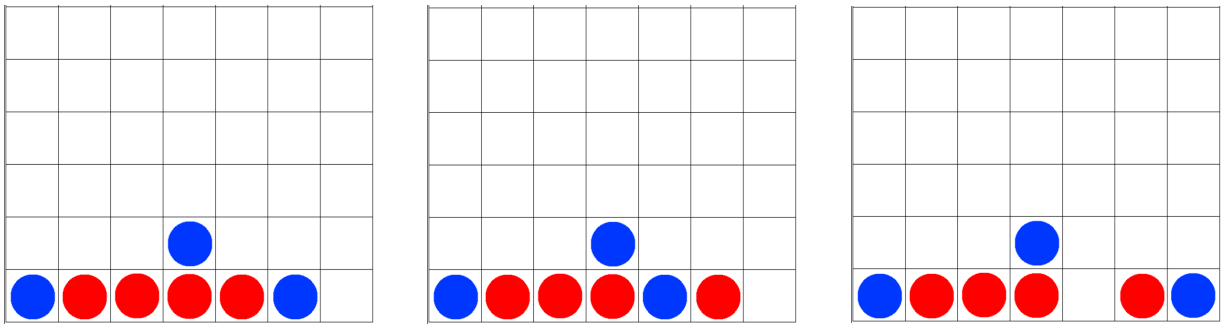
Wat zou de tijdcomplexiteit worden, als we, in plaats van dynamisch programmeren, de recurrente betrekkingen rechtstreeks zouden omzetten in recursieve functies L en R, waarbij dus deelresultaten meerdere keren berekend zouden worden?

4. [28 pt] Virgil en Georginio hebben een nieuw spel ontdekt: Vier op een lijn. Het wordt gespeeld op een bord van M rijen en N kolommen, genummerd van 0 t/m $M-1$, respectievelijk 0 t/m $N-1$. Omdat het bord rechtop staat, vallen stenen, wanneer ze worden losgelaten boven een kolom, automatisch naar het laagste lege vakje in die kolom. Virgil speelt met de rode stenen, Georginio met de blauwe stenen. Om de beurt kiezen ze een van de kolommen (die nog niet vol zijn) en laten ze daarin een steen van hun kleur vallen.

   Beschouw het volgende bord (zes rijen, zeven kolommen), waar eerst Virgil en vervolgens Georginio een steen in de middelste kolom heeft laten vallen:



   De speler die er als eerste in slaagt om vier van zijn stenen op een lijn (horizontaal of verticaal, NIET diagonaal) te krijgen, wint. Het spel is dan afgelopen. De vier stenen moeten in dezelfde rij of kolom liggen, met ten hoogste 1 vakje ertussen dat geen steen van die kleur bevat. Hieronder drie voorbeelden van borden met vier rode stenen op een lijn, zodat Virgil wint:



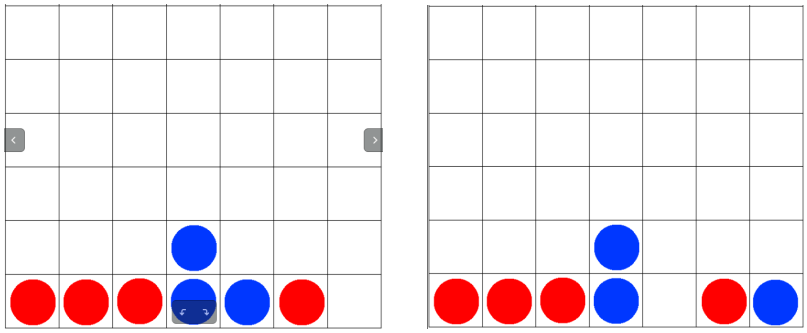
(a) (b) (c)

(a) De vier rode stenen liggen naast elkaar op de onderste rij. (b) De vier rode stenen liggen op de onderste rij, met 1 blauwe steen ertussen. (c) De vier rode stenen liggen op de onderste rij, met een leeg vakje ertussen.

Merk op dat bij deze drie voorbeelden de vier stenen van dezelfde kleur steeds horizontaal liggen, maar dat verticaal natuurlijk ook mogelijk is.

Z.O.Z.

Nu volgen twee voorbeelden van borden waar nog geen van de spelers heeft gewonnen:



(a) (b)

(a) Er liggen twee blauwe stenen tussen de vier rode stenen. (b) Er ligt een blauwe steen en een leeg vakje tussen de vier rode stenen.

(a) Geef een C++ functie `bool inlijn (int bord[M][N], int sp)` die controleert of er op bord `bord` vier stenen op een lijn liggen van speler `sp`.

De parameter `bord` is dus een 2-dimensionaal array met integers, met M rijen en N kolommen. De onderste rij van het bord is rij 0. De entries zijn getallen 0 (het vakje is nog leeg), 1 (een rode steen, van Virgil) of 2 (een blauwe steen, van Georginio). De parameter `sp` is 1 (Virgil) of 2 (Georginio).

(b) Geef een recursieve C++ functie `bool winnend (int bord[M][N}, int sp)` die met behulp van brute force bepaalt of speler `sp`, bij optimaal spel van beide spelers, het spel wint, uitgaande van de stand op bord `bord`, waarbij speler `sp` (in die stand) aan de beurt is.

De parameters `bord` en `sp` zijn van dezelfde vorm als bij onderdeel (a).

Je mag ervan uitgaan dat parameter `bord` een geldig bord voorstelt, waarbij nog geen enkele speler vier stenen op een lijn heeft. Onder een geldig bord verstaan we onder andere dat de stenen in elke kolom zo ver mogelijk naar beneden liggen (er zijn geen gaten onder).

Als het bord vol is, terwijl geen enkele speler vier op een lijn heeft, dan verliest de speler die aan de beurt is.

(c) Beschrijf (in woorden, niet in code) een gretig algoritme dat een zet (kolom) kiest voor de speler die in een stand aan de beurt is. Als hij uit meerdere zetten kan kiezen, welk criterium / welke criteria gebruikt hij dan om een keuze te maken?
