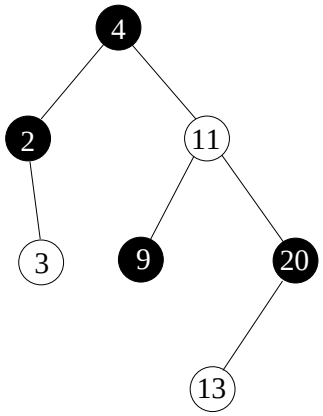
Examen Datastructuren en Algoritmen II

Naam : .....

- Lees de hele oefening zorgvuldig voordat je begint ze op te lossen!
  Als je niet goed verstaat wat de vraag of taak is, vraag het aan de lesgever! Voor oefeningen die fout verstaan zijn, kunnen geen punten gegeven worden!
- Schrijf leesbaar. Oplossingen die niet leesbaar zijn, kunnen ook niet beoordeeld worden.
- Als technieken toegepast moeten worden, toon altijd voldoende tussenstappen om te kunnen zien wat er gebeurt en dat de technieken goed verstaan zijn.
- Stellingen uit de les mogen natuurlijk altijd gebruikt worden zonder dat het bewijs opnieuw gegeven moet worden (behalve in gevallen waar het expliciet anders staat)!
- Geef alleen dan een antwoord als je denkt dat je de oplossing kent. Verspil geen tijd met de poging gewoon lange teksten te schrijven waarin zekere sleutelwoorden opduiken – zoals dat vaak geprobeerd wordt. Dergelijke oplossingen halen nooit punten en voor bijzonder slechte oplossingen worden punten afgetrokken!

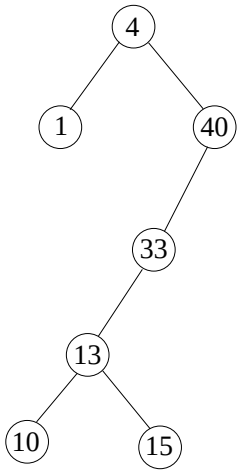
1. **Zoekbomen 2.25 pt**

   - Voeg sleutel 14 toe aan de volgende rood-zwart boom. Gebruik daarbij de bewerkingen die enkel rekening houden met de kleuren van toppen op het doorlopen pad en niet met de kleuren van kinderen die niet op het pad liggen.



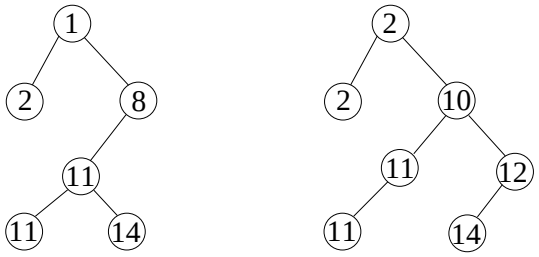
- Bewijs één van de volgende twee uitspraken:
  - Er bestaat een constante c zodat voor alle rood-zwart bomen T en alle toppen $x \in T$ met l zwarte toppen in het linkerdeelboom en r zwarte toppen in het rechterdeelboom van x geldt $l \leq c * r$.
  - Er bestaat geen constante c zodat voor alle rood-zwart bomen T en alle toppen $x \in T$ met l zwarte toppen in het linkerdeelboom en r zwarte toppen in het rechterdeelboom van x geldt $l \leq c * r$.

- Pas de bewerking “verwijder 12” toe op de volgende semi-splay boom.

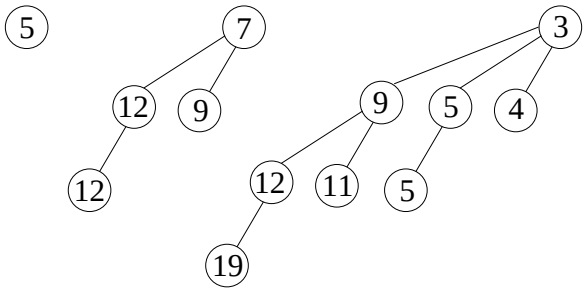


2. **Heaps 2.25 pt**

- Merge de volgende twee skew heaps met de recursieve skew merge bewerking:



- Voeg de sleutels, 1, 5 en 3 in deze volgorde toe aan de volgende binomiale wachtlijn.



- In de les werd er meerdere keren op gewezen, dat alle heaps die je tijdens jouw opleiding hebt gezien voor- en nadelen hebben. Geef voor elk van de volgende doelstellingen één heap die daarvoor optimaal presteert en leg **precies** uit wat het voordeel van deze heap is vergeleken met één andere die op dat vlak minder goed presteert.
  - klein geheugengebruik
  - snel het kleinste element vinden
  - snel toevoegen – vooral als een hele reeks van elementen wordt toegevoegd, die niet al op voorhand gekend is

3. **Branch and bound 1 pt**

Als je geen heuristiek zoals first fit dalend of best fit dalend op het inpakprobleem wil toepassen, maar gegarandeerd een optimale oplossing nodig hebt, dan is een branch and bound algoritme zoals het volgende een mogelijkheid.

Gegeven de verzameling $G = \{g[1], \ldots, g[n]\}$ van gewichten die allemaal ten hoogste 1 zijn. Als een deelverzameling $H \subseteq G$ van gewichten is gegeven die al op vrachtwagens geplaatst zijn, geeft de functie `ondergrens(H)` een ondergrens voor het aantal vrachtwagens die door een oplossing gebruikt worden die de gewichten uit H op de gegeven manier plaatst.

```
plaats( index, aantal_vrachtwagens )
{

als (index==n+1)
  { als (aantal_vrachtwagens < beste_oplossing)
          beste_oplossing = aantal_vrachtwagens;
    return;
  }

for (vw_index=1; vw_index <= aantal_vrachtwagens; vw_index++)
  { als (g[index] past nog op vrachtwagen[vw_index])
       { plaats g[index] op vrachtwagen[vw_index];
         als (ondergrens(g[1],...,g[index]) < beste_oplossing)
             plaats( index+1, aantal_vrachtwagens );
         verwijder g[index] van vrachtwagen[vw_index];
       }
  }

plaats g[index] op vrachtwagen[aantal_vrachtwagens+1];
als (ondergrens(g[1],...,g[index]) < beste_oplossing)
             plaats( index+1, aantal_vrachtwagens+1 );
verwijder g[index] van vrachtwagen[aantal_vrachtwagens+1];
}
```

Het algoritme wordt opgestart met `plaats(1,0)` en `beste_oplossing=oneindig`.

Geef een gemakkelijk voorstel om dit algoritme te verbeteren (denk aan de oefening over het kleuren van grafen) en leg uit waarom volgens jou het algoritme achteraf sneller zou moeten draaien.

4. **Geamortiseerde complexiteit 2.5 pt**

   Het belangrijkste deelresultaat in het bewijs van de geamortiseerde complexiteit van skew heaps was het volgende:

   Deelresultaat 4: Als een skew merge bewerking op twee skew heaps S_1, S_2 in een verzameling D van skew heaps wordt toegepast en het resultaat is D' dan geldt: $\Phi(D') - \Phi(D) \leq g - s$ waarbij g het aantal goede toppen en s het aantal slechte toppen is die op het rechterpad van S_1 of op het rechterpad van S_2 liggen.

   Daarbij was een top *slecht* als hij meer kinderen in zijn rechterdeelboom had dan in zijn linkerdeelboom en anders *goed*. Wij hadden gedefinieerd:

   $$\Phi(D) := |\{v \in D | v \text{ is slecht}\}|$$

   Deelresultaat 4 is in deze vorm niet meer juist, als je het op de recursieve versie van de skew merge bewerking toepast. Geef een herformulering van Deelresultaat 4 die wel juist is voor de recusieve merge bewerking en gebruikt kan worden om te bewijzen dat de gewijzigde kost van een recursieve skew merge bewerking $O(\log n)$ is. Geef ook het bewijs van deze grens.

5. **Dynamisch programmeren 2.5 pt**

Gegeven een rij $a_1, a_2, \ldots, a_n$ van rationale getallen die allemaal groter dan 0 zijn. Wij kijken nu naar de uitdrukking $a_1/a_2/\ldots/a_n$ en willen daar haakjes zetten zodat de evaluatievolgorde door de haakjes uniek bepaald wordt en de waarde maximaal onder alle evaluatievolgordes is.

Als de rij $\frac{1}{4}, \frac{2}{3}, \frac{3}{5}$ is, dan is
$(\frac{1}{4}/\frac{2}{3})/\frac{3}{5} = \frac{5}{8} = \frac{25}{40}$
en de andere manier om haakjes te zetten geeft
$\frac{1}{4}/(\frac{2}{3}/\frac{3}{5}) = \frac{9}{40}$
– de eerste manier geeft dus de maximale waarde.

Tip: Als je weet wat de laatste divisie is die je moet doen – bv. die die na a_i staat – dan krijg je het grootste getal door de **maximaal** mogelijke waarde voor de linkerkant en de **minimaal** mogelijke waarde voor de rechterkant te nemen.

Geef een algoritme dat dit probleem met dynamisch programmeren oplost en een complexiteit heeft van $O(n^3)$. Geef de pseudocode en voldoende uitleg. Het algoritme moet alleen de maximale waarde geven en hoeft de evaluatievolgorde niet bij te houden.

6. **Gretige heuristieken 1 pt**

Geef een gretig algoritme voor het volgende probleem. Schrijf expliciet waarom jouw algoritme een *gretig algoritme* is en geef uitleg hoe goed de optimale oplossing door dit algoritme wordt benaderd:

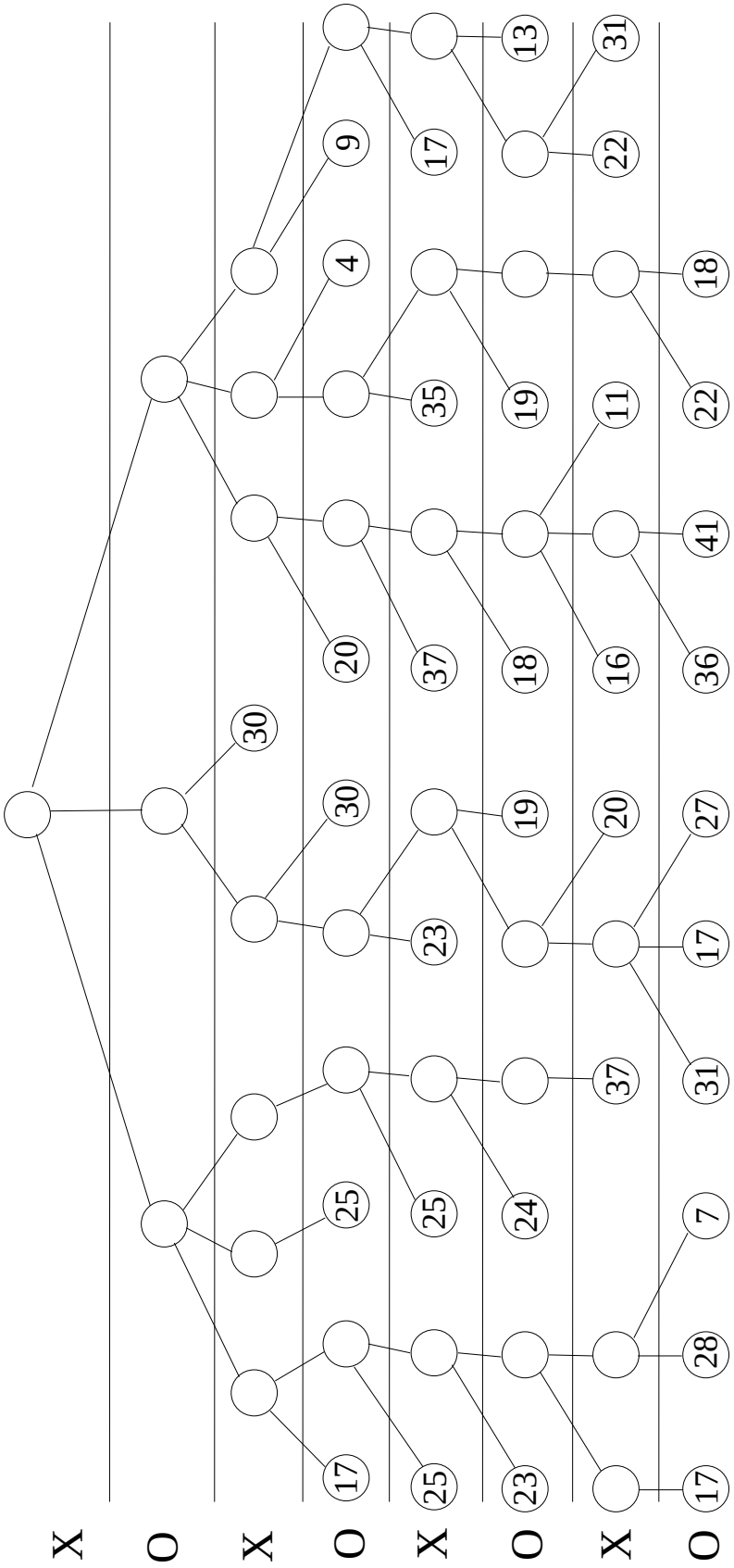
In een bedrijf moeten gaten op een groot aantal moederborden geboord worden. De gaten zijn op posities $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$. De tijd die de machine nodig heeft om van positie p_i naar positie p_j te gaan is $t(p_i, p_j) = t(p_j, p_i)$. Omdat soms elementen die al op het moederbord geplaatst zijn in de weg staan, kan dat heel lang duren – de tijd kan dus *in principe* elk getal zijn en niet noodzakelijk een veelvoud van de Euklidische afstand tussen de punten.

De taak is nu een manier te vinden de gaten zo snel mogelijk te boren – dus een reeks $p_{i_1}, \ldots, p_{i_n}$ te vinden zodat de som $\sum_{k=1}^{n-1} t(p_{i_k}, p_{i_{k+1}})$ van de afstanden minimaal is. De tijd om te boren is natuurlijk voor alle reeksen dezelfde. Omdat je bij het volgende moederbord de reeks gewoon in de andere richting kan doorlopen, moet je niet van p_{i_n} terug naar p_{i_1} gaan en telt de tijd $t(p_{i_n}, p_{i_1})$ dus ook niet mee.

7. α–β–**snoeien 2 pt**

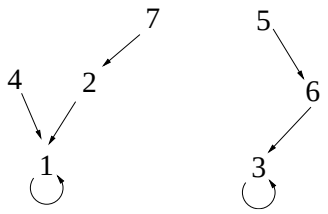
- Bewijs de volgende uitspraak of geef een tegenvoorbeeld:
  Stel dat `get_waarde(t)` de functie uit de les is die α–β–snoeien implementeert.
  Als *t* een top in een spelboom is die op een constant pad ligt en `get_waarde()` wordt op de wortel toegepast, dan wordt `get_waarde(t)` ook opgeroepen.
  Informeel zou je dus kunnen zeggen dat toppen die op een constant pad liggen en dus de waarde van de wortel kunnen bepalen, moeten geëvalueerd worden en – omdat `get_waarde()` juist werkt – worden die ook geevalueerd.

- Pas α–β–snoeien op de volgende spelboom toe om de waarde van het spel te berekenen. Evalueer altijd eerst de linkertakken. Schrijf gebruikte grenzen **aan** de toppen en de teruggegeven waarden **in** de toppen. Plaats streepjes door de bogen die naar deelgrafen leiden die je niet evalueert omdat je snoeit.



8. **Verzamelingen 1.5 pt**

- Pas eerst de relatie $1 \equiv 5$ en dan de relatie $7 \equiv 3$ toe op de volgende union-find datastructuur. Gebruik union by size met path compression.



- Ons universum is de verzameling $\{0,\ldots,n\}$ met $n < 60$ en wij werken met een computer waar de lengte van een woord 64 bit is. M en M' zijn twee verzamelingen die als bitvectoren voorgesteld zijn en i is een element van $\{0,\ldots,n\}$. Beschrijf (zoals in de les) zo efficiënt mogelijke uitdrukkingen die het volgende betekenen:
  - Er is maar één element dat in M en M' zit en dat is i.
  - i is een element van precies één van de verzamelingen M, M'
  - niet elk element $0 \leq j < 60$ komt in M of M' voor.

9. **Gerandomiseerde algoritmen 1 pt**

In de les hadden wij de volgende definitie:

Gegeven een oneven getal x. Dan kunnen wij $x-1$ schrijven als $x-1=2^t u$ met een oneven getal u en $t \geq 1$.

Als nu $b \in \{1,\ldots,x-1\}$ en

$b^u (\mod x) \notin \{1,-1\}$ **en** $\forall 0 < s < t : b^{2^s u} \not\equiv -1 (\mod x)$

dan noemen wij b een speciale getuige voor x.

Bewijs dat als een speciale getuige voor een oneven getal x bestaat, het getal x zeker niet priem is.

Dat kan bv. met een redenering die analoog is met de redenering in de les die toont dat het Miller-Rabin algoritme getuigen *"herkent"*.

Gebruik hier niet gewoon dat het Miller-Rabin algoritme voor getuigen *"niet priem"* geeft en dat dit antwoord altijd juist is.

NOG NIET OMDRAAIEN !