Eksamen i IN2010 høsten 2021

10. Desember 2021

Om eksamen

- Eksamen består av en bitteliten oppvarming, etterfulgt av to hoveddeler.
- Den første delen består av små oppgaver, som rettes automatisk. Det gjøres ingen forskjell mellom ubesvart og feil svar; det betyr at det lønner seg å svare på alle oppgavene.
- Den andre delen består av litt større oppgaver hvor du i større grad må programmere (pseudokode), skrive og resonnere.
- Alle besvarelser skal skrives inn i Inspera og det er ingen mulighet for opplasting av håndskrevede svar.
- Ingen hjelpemidler er tillatt.

Kommentarer og tips

- Det er lurt å lese raskt gjennom eksamen før du setter i gang. Hele oppgavesettet er lagt ved som PDF.
- Det kanskje viktigste tipset er *å lese oppgaveteksten svært nøye*.
- Pass på at du svarer på nøyaktig det oppgaven spør om.
- Pass på at det du leverer fra deg er klart, presist og enkelt å forstå, både når det gjelder form og innhold.
- Hvis du står fast på en oppgave, bør du gå videre til en annen oppgave først.
- Alle implementasjonsoppgaver skal besvares med *pseudokode*. Det viktige er at pseudokoden er *lett forståelig, entydig og presis*.
- En *lett forståelig, entydig og presis* forklaring med naturlig språk, kan være mer poenggivende enn pseudokode som er vanskelig å forstå, tvetydig eller upresis.

Sensorveiledning

Sensorveiledningen inneholder et løsningsforslag på alle oppgaver. I tillegg gis det veiledning på hvordan poeng kan fordeles, men sensorene står fritt til å gi eller trekke poeng etter eget skjønn. Det legges alltid vekt på at besvarelsene er presise og enkle å forstå, både med hensyn til form og innhold.

Oppvarming 2 poeng

(a) Hva er en algoritme? Svar kort (maks fire setninger).

(b) Hva er en datastruktur? Svar kort (maks fire setninger).

Vi er ikke ute etter et «fasitsvar». Vi er ute etter å høre din forståelse av begrepene, og alle rimelige svar gir full uttelling.

Her finnes det utrolig mange riktige svar! Vi er ikke ute etter en formell definisjon (selv om det selvfølgelig vil være poenggivende), men vil bare få studentene i gang med å tenke på algoritmer og datastrukturer. Her er mitt svar (som på ingen måte bør være førende for hvordan vi tildeler poeng):

- Algoritmer er idéene bak de programmene vi skriver. De angir hva som må gjøres for å løse et gitt problem på en presis og entydig måte. Følger du en algoritme for å løse et problem så skal du også få riktig svar til slutt, og helst på rimelig tid.
- Datastrukturer er måter å organisere data på. Stort sett vil vi organisere dataene etter hva vi oftest ønsker å gripe tak i, slik at vi kan få mer effektive algoritmer.

Vi gir full uttelling for alle rimelige svar. For å *ikke* få full uttelling må man enten la oppgaven stå ubesvart, eller skrive noe som er direkte feil. Det kan trekkes poeng dersom deler av svaret inneholder vesentlige feil.

Stabilitet 3 poeng

Anta at arrayet `A` er usortert og inneholder personobjekter som alle har et felt for alder. Anta videre at personobjektene som ligger på `A[3]` og `A[42]` begge er 22 år gamle.

Arrayet `A` blir sortert etter alder. Etter sorteringen får du vite at:

- Personobjektet som lå på `A[3]` før sorterting, ligger nå på `A[9]`
- Personobjektet som lå på `A[42]` før sorterting, ligger nå på `A[7]`

(a) Var sorteringen stabil?

> Nei.

(b) Hva er alderen til personobjektet som ligger på `A[8]` etter sortering?

> 22.

(c) Dersom du får vite at ingen personer i `A` er eldre enn 100 år gammel. Hvilken sorteringsalgoritme bør da benyttes, med hensyn til kjøretidseffektivitet?

> Bucket sort.

Litt sortering 10 poeng

For hver av påstandene nedenfor kan du anta at `A` er et array med n elementer, og at i er et heltall $0 \leq i < n$.

Bubble sort

(a) Etter i iterasjoner av den ytre loopen i Bubble sort, er de i *første* elementene sortert.

Usant

(b) Etter i iterasjoner av den ytre loopen i Bubble sort, er de i *siste* elementene sortert.

Sant

(c) Bubble sort bytter kun elementer som står direkte ved siden av hverandre.

Sant

(d) Bubble sort garanterer et minimalt antall bytter.

Usant

Selection sort

(e) Etter i iterasjoner av den ytre loopen i Selection sort, er de i *første* elementene sortert.

Sant

(f) Etter i iterasjoner av den ytre loopen i Selection sort, er de i *siste* elementene sortert.

Usant

(g) Selection sort bytter kun elementer som står direkte ved siden av hverandre.

Usant

(h) Selection sort garanterer et minimalt antall bytter.

Sant

Insertion sort

(i) Etter i iterasjoner av den ytre loopen i Insertion sort, er de i *første* elementene sortert.

Sant

(j) Etter i iterasjoner av den ytre loopen i Insertion sort, er de i *siste* elementene sortert.

Usant

(k) Insertion sort bytter kun elementer som står direkte ved siden av hverandre.

Sant

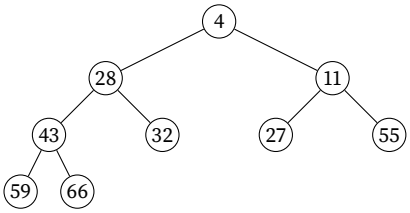
(l) Insertion sort garanterer et minimalt antall bytter.

Usant

Det gis 1 poeng for riktig svar og -1 for feil eller ubesvart. Maksimalt får man 10 poeng, som vil si at man kan få full uttelling med én feil. Man kan ikke få mindre enn 0 poeng totalt.

Binære heaps 6 poeng

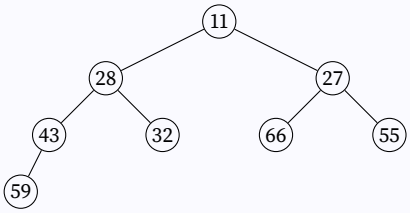
Du skal ta utgangspunkt i følgende binære heap H_1, hvor både tre-representasjonen og array-representasjonen er gitt:



4	28	11	43	32	27	55	59	66
0	1	2	3	4	5	6	7	8

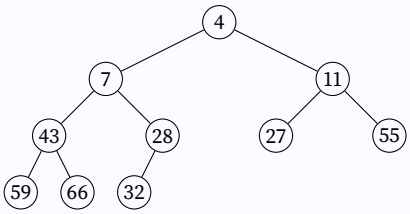
Du skal fylle inn tabellene som svarer til hver deloppgave i Inspera. Hver tabell svarer til array-representasjonen av H_1 etter de oppgitte operasjonene. Deloppgavene er uavhengig av hverandre, altså refererer H_1 til heapen som er gitt ovenfor, og endringer vi gjør i en deloppgave følger ikke med til neste deloppgave.

(a) Hvordan ser heapen ut etter ett kall på H_1.`RemoveMin`()?



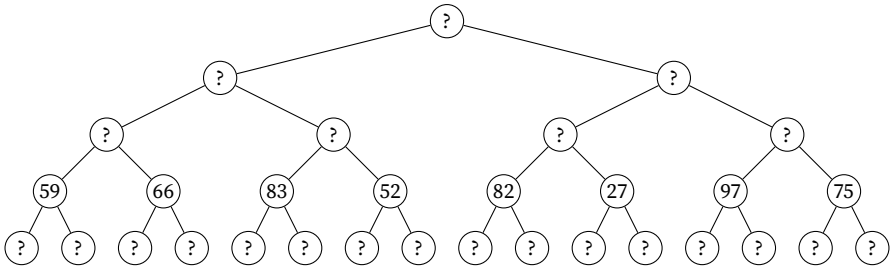
11	28	27	43	32	66	55	59
0	1	2	3	4	5	6	7

(b) Hvordan ser heapen ut etter ett kall H_1.`Insert`(7)?



4	7	11	43	28	27	55	59	66	32
0	1	2	3	4	5	6	7	8	9

I en annen binær min-heap H_2 får du kun gitt verdiene på dybde 3:



(c) Hva er den minste verdien som kan ligge på dybde 4 av H_2?

27. En heap tillater duplikater, og det ligger en node med verdi 27 på nivå 3 kan den noden ha et barn med samme verdi, altså 27.

Det gis 2 poeng per riktig svar (og 0 for feil svar). På oppgave (c) gis det 1 poeng for svaret 28.

Huffmantrær 5 poeng

En tekststreng vi vil komprimere består av 5 ulike tegn a,b,c,d og e. De relative frekvensene er gitt av følgende frekvenstabell:

a	b	c	d	e
2	1	12	1	4

(a) Hva er den lengste kodelengden for et symbol i det tilhørende huffmantreet?

4.

(b) Hvor mange bits blir koden for tegnet e?

2.

(c) Hvor mange bits brukes for å kode strengen aaabbcdee?

26.

Kodelengdene for de ulike symbolene er:

a	b	c	d	e
3	4	1	4	2

Vi har 3 a-er, 2 b-er, 1 c, 1 d og 2 e-er, som gir $3 \cdot 3 + 2 \cdot 4 + 1 + 4 + 2 \cdot 2 = 9 + 8 + 1 + 4 + 4 = 26$.

Nå skal vi ikke jobbe med noe konkret huffmantre for et en gitt frekvenstabell, men heller tenke på generelle huffmantrær.

(d) Hvis en frekvenstabell består av 8 forskjellige tegn, hvor mange noder har det tilhørende huffmantreet?

15. Det vil ha 8 løvnoder og 7 interne noder.

(e) I et huffmantre som har 8 løvnoder, hva er den lengste kodelengden et symbol kan ha?

7.

Det gis 1 poeng per deloppgave.

Litt om beregnbarhet og kompleksitet 8 poeng

For alle spørsmålene nedenfor, svar **Sant** eller **Usant**.

(a) Det er bevist at $P = NP$.

Usant.

(b) Det er bevist at $P \neq NP$.

Usant.

(c) Alle NP-komplette problemer kan polynomtidsreduseres til hverandre.

Sant.

(d) Alle problemer i P er også i NP.

Sant.

(e) Alle avgjørelsesproblemer er enten i P eller NP.

Usant.

(f) Alle problemer hvor JA-instanser kan verifiseres i polynomiell tid, er i NP.

Sant.

(g) Hvis noen klarer å løse avgjørelsesproblemet **Hamiltonsykel**, **Knapsack** eller **Sudoku** i polynomiell tid, så har de bevist at $P = NP$.

Sant.

(h) Alle avgjørelsesproblemer kan løses med en rask nok datamaskin.

Usant.

Det gis 1 poeng for riktig svar og -1 for feil eller ubesvart. Maksimalt får man 8 poeng og minimalt 0 poeng.

Korteste avstander i grafer 8 poeng

For hver graftype, velg den raskeste algoritmen som finner korteste avstander fra en startnode til alle andre noder i grafen.

	Bredde-først søk	Dijkstra	Topologisk Sortering	Bellman-Ford
Ingen negative kanter				
Vektet DAG				
Uvektet				
Ingen negative sykler				

- Uvektet: en uvektet graf. Grafen er enten rettet eller urettet.
- Vektet DAG: en vektet, rettet asyklisk graf. Kantene kan ha negativ vekt.
- Ingen negative kanter: grafen er vektet, men ingen kanter har negativ vekt. Grafen er enten rettet eller urettet.
- Ingen negative sykler: grafen er vektet, men inneholder ingen sykler med negativ vekt. Grafen er enten rettet eller urettet.

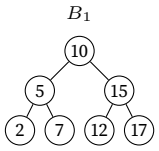
	Bredde-først søk	Dijkstra	Topologisk sortering	Bellman-Ford
Ingen negative kanter		×		
Vektet DAG			×	
Uvektet	×			
Ingen negative sykler				×

Det gis 2 poeng per riktig svar.

Intervall i et binært søketre 10 poeng

Du er gitt et binært søketre B og to heltall a og b der $a \leq b$. Du skal gi en algoritme som skriver ut verdiene innenfor intervallet $[a, b]$ i *sortert* rekkefølge (fra minst til størst). Det vil si at hvis x er verdien i en node, så skal denne noden skrives ut hvis og bare hvis $a \leq x \leq b$.

```
Input: Et binært søketre B og to heltall a og b der a ≤ b
Output: Skriver ut verdiene innenfor intervallet [a, b] i sortert
        rekkefølge
1 Procedure InRange(B, a, b)
  | // ···
```



For det binære søketreet B_1 (vist i eksempelet ovenfor) skal for eksempel `InRange(`B_1`, 2, 7)` skrive ut 2, 5 og 7, og `InRange(`B_1`, 10, 20)` skrive ut 10, 12, 15 og 17.

(a) Oppgi algoritmen din (med psuedokode). Du kan gjøre rimelige antagelser om representasjonen av B. En mer effektiv algoritme er mer poenggivende. Du trenger ikke tenke på formateringen på utskriften.

I de neste deloppgavene skal du anta at B har n noder, er balansert og ikke inneholder noen duplikater.

(b) Anta at a er den minste verdien i B og b er den største verdien i B. Oppgi kjøretidskompleksiteten på algoritmen din med hensyn til n.

(c) Anta at $a = b$. Oppgi kjøretidskompleksiteten på algoritmen din med hensyn til n.

Vi gir ingen føringer på hvordan det binære treet skal være implementert. Studenten må selv gjøre rimelige antagelser her (som at det en node har et element x en venstre- og en høyrepeker). Her antar vi at B er representert ved rotnoden, og tar denne noden som input. Hver node har en venstre, høyre og en x. Vi representerer det tomme treet ved **`null`**.

(a)

```
Input: Et balansert binært søketre B og to heltall a og b der a ≤ b
Output: Skriver ut verdiene innenfor intervallet [a, b] i sortert rekkefølge
1 Procedure InRange(v, a, b)
2   if v = null then
3     return null
4   if a < v.x then
5     InRange(v.left, a, b)
6   if a ≤ v.x ≤ b then
7     print(v.x)
8   if v.x < b then
9     InRange(v.right, a, b)
```

Denne algoritmen har kjøretidskompleksitet $O(\log n + \min(n, (b-a)))$, men dette trenger ikke studenten å oppgi.

(b) Algoritmen kjører i $O(n)$ dersom a og b er henholdsvis det største og minste tallet i B.

(c) Dersom $a = b$ er kjøretidskompleksiteten $O(\log n)$.

For full uttelling må studenten ha en god fremstilling av en algoritme som har $O(\log n)$ når $a = b$, og oppgi riktig kjøretidskompleksitet. En $O(n)$ algoritme kan få opp til 8 poeng. En $O(n \cdot \log n)$ algoritme kan få opp til 5 poeng. En $O(n^2)$ algoritme kan få opp til 2 poeng.

Flest forekomster — 8 poeng

Strategi

Input: Et array A med n positive heltall
Output: Tallet som forekommer flest ganger i A

```
1  Procedure MostFrequent1(A)
2      H ← new HashMap()
3      for x in A do
4          H.put(x, 0)
5      m ← A[0]
6      for x in A do
7          f ← H.get(x)
8          H.put(x, f + 1)
9          if H.get(x) > H.get(m) then
10             m ← x
11     return m
```

Strategi

Input: Et array A med n positive heltall
Output: Tallet som forekommer flest ganger i A

```
1  Procedure MostFrequent2(A)
2      k ← maximum element in A
3      B ← array of size k + 1
4      for i ← 0 to k do
5          B[i] ← 0
6      m ← A[0]
7      for x in A do
8          B[x] ← B[x] + 1
9          if B[x] > B[m] then
10             m ← x
11     return m
```

Ovenfor ser du to (korrekte) algoritmer som begge tar et array A med positive heltall som input, og returner tallet som forekommer flest ganger. Drøft fordeler og ulemper ved de to ulike strategiene. Du bør oppgi både kjøretidskompleksiteten i *verste* tilfelle, og hva som er den *forventede* kjøretidskompleksiteten, med hensyn til n og k. Den bør begrunnes *kort*. Du bør også oppgi hvilke situasjoner **Strategi 1** bør foretrekkes over **Strategi 2**, og vice versa, med hensyn til konkret kjøretid. Gjøre rede for eventuelle antagelser du gjør underveis.

Dette er en oppgave som går ut på å gjøre rimelige antagelser og anvende dem til å si noe fornuftig om hvilken fremgangsmåte man bør bruke. Dette er den oppgaven fra eksamenssettet hvor sensoren er nødt til å anvende mest skjønn i poengsettingen. Tanken bak oppgaven er å få studenten til å reflektere over hvilken fremgangsmåte man bør velge i lys av kjøretidskarakteristikkene til de to strategiene. Det bør legges vekt på at teksten er godt skrevet (dette betyr ikke at det skal trekkes poeng for enkle skrivefeil, men at teksten må være tydelig og enkel å forstå; en fin tommelfingerregel er at en setning man ikke forstår etter to gjennomlesninger ignoreres). Under lister vi noen refleksjoner vi ønsker å se.

Strategi 1 benytter seg av et hashmap, som i verste tilfelle har lineærtid på innsetting og oppslag. Den gjør $O(n)$ innsettinger og oppslag i H, som tilsammen utgjør $O(n^2)$ kjøretidskompleksitet i verste tilfelle. (Merk at vi her antar et hashmap implementert med linear probing eller separate chaining, slik vi har sett i kurset, og ikke for eksempel Java sine hashmaps, som bruker separate chaining, hvor hver «bøtte» er et rød-svart tre og gir logaritmisk tid på alle operasjoner i verste tilfelle.) Hashmaps har derimot *forventet* konstant tid på innsetting og oppslag, og dermed får $O(n)$ forventet kjøretidskompleksitet.

Strategi 2 bruker kun arrayer, som har konstant tid på alle operasjoner. I motsetning til **Strategi 1** finner vi det største elementet k, og lager et array B med plass til k elementer, og initialiserer det med verdien 0. Å finne det største elementet er i $O(n)$, og opprettelsen og initialiseringen av B er i $O(k)$. Den siste løkken er også i $O(n)$. Tilsammen gir dette $O(n + k)$ i både verste og forventet kjøretidskompleksitet (fordi vi ikke benytter oss av hashing).

Den mest vesentlige forskjellen mellom **Strategi 1** og **Strategi 2** er at **Strategi 2** er treg dersom k er stor. Men hvis dette *ikke* er tilfelle (som for eksempel kan bety at $k < n$) så vil **Strategi 2** være mer effektiv enn **Strategi 1**, fordi en hashing har en del høyere konstantfaktorer som følge av rehashing og kollisjonshåndtering. I tillegg er det alltid en (svært liten) sjanse for at vi får mange kollisjoner, som gjenspeiles i analysen for verste tilfelle.

Merk: Det var en liten feil i **Strategi 2**, der vi lot størrelsen på B må være k, snarere enn $k + 1$.

Finn par som summerer til x 10 poeng

Du er gitt et array med unike heltall `A` og et heltall x. Du skal skrive en prosedyre som skriver ut alle par av heltall y og z i arrayet, der $y + z = x$. Tallet på en gitt posisjon i arrayet kan maksimalt inngå i én utskrift. Rekkefølgen parene (y, z) skrives ut i spiller ingen rolle, og rekkefølgen innad i parene spiller heller ingen rolle.

```
Input: Et array A av heltall, og et heltall x
Output: Skriver ut alle par y, z ∈ A slik at y + z = x
1 Procedure FindSummands(A, x)
  | // ···
```

For eksempel skal et kall på `FindSummands(`$[0, 2, 4, 6, 8, 10]$`, 10)` skrive ut parene $(0, 10)$, $(2, 8)$ og $(4, 6)$.

For begge deloppgaver: Gi pseudokode for en algoritme som løser problemet **og** oppgi kjøretidskompleksiteten på algoritmen (den trenger ikke begrunnes). Lavere kjøretidskompleksitet er mer poenggivende.

(a) Skriv en prosedyre `FindSummands` som beskrevet over, med antagelsen om at `A` er *sortert fra minst til størst*.

(b) Skriv en prosedyre `FindSummands` som beskrevet over, men du kan *ikke* anta at `A` er sortert. Hint: Du kan anta $O(1)$ for innsetting, oppslag og sletting i hashbaserte datastrukturer.

Både oppgave (a) og (b) kan enkelt løses i kvadratisk tid, slik:

```
1  Procedure FindSummands(A, x)
2      for i ← 0 to |A| − 1 do
3          for j ← i + 1 to |A| − 1 do
4              if A[i] + A[j] = x then
5                  print (A[i], A[j])
```

I oppgave (a) kan vi oppnå lineærtid ved å holde styr på to indekser i og j, der i initielt er 0 og økes, og j initielt er $|A|$ og minkes.

```
1  Procedure FindSummands(A, x)
2      i ← 0
3      j ← |A| − 1
4
5      while i < j do
6          y ← A[i] + A[j]
7          if x < y then
8              j ← j − 1
9          else if x > y then
10             i ← i + 1
11         else
12             print(A[i], A[j])
13             i ← i + 1
```

I oppgave (b) kan vi oppnå $O(n)$ ved å sortere med radix-sort først, og deretter bruke løsningen fra (a), eventuelt $O(n \cdot \log(n))$ ved merge sort eller lignende. Vi kan også oppnå forventet linærtid ved å lagre elementer vi har sett i en mengde (implementert ved hashing), og for hvert element i $y \in A$ sjekke om $x - y$ er i mengden.

```
1  Procedure FindSummands(A, x)
2      H ← new HashSet()
3      for y in A do
4          if x − y is in H then
5              print(y, x − y)
6              H.remove(y)
7          else
8              H.add(y)
```

For full uttelling forventer vi $O(n)$ i forventet kjøretidskompleksitet for (a) og (b). Dersom noen har en $O(n)$ algoritme i oppgave (b), med en kommentar om at den kan brukes for (a), så gir det full uttelling. Man kan også få full uttelling for a bruke løsningen fra (a) og bruke radix sort først i (b).

For en $O(n \cdot \log n)$ løsning på (b) kan man få opp til 8 poeng.

For kvadratisk tid på begge løsninger kan det gis opp til 3 poeng.

To-fargelegge en graf 10 poeng

Du er gitt en *urettet og sammenhengende* graf $G = (V, E)$. Du skal lage en algoritme som fargelegger nodene i V med to farger R(rød) og B(blå), slik at ingen naboer $(u, v) \in E$ har samme farge; hvis dette *ikke er mulig*, så skal algoritmen skrive ut at det ikke er mulig.

Du kan anta at hver node i V har et felt $v.\texttt{color}$ som initielt er satt til `null`, og at du har tilgang på en funksjon `FlipColor` der `FlipColor(R)` $= B$ og `FlipColor(B)` $= R$.

Input: En sammenhengende graf $G = (V, E)$
Output: En graf der nodene er fargelagt hvis det er mulig

```
1 Procedure TwoColor(G)
  | // ···
```

Å fargelegge en graf med to farger kan gjøres enkelt med et dybde-først (eller bredde-først) søk, der vi flipper fargen mellom hvert rekursive kall i søket. Dersom vi når en node, som allerede har en satt farge, som er *ulik* den som skal settes, så kan vi avbryte med melding om at det ikke er mulig å fargelegge grafen med to farger. Grunnen til at denne strategien fungerer (riktig nok bare for to farger), er at hvert valg av farge tvinges av foreldrenoden i DFS-spenntreet.

Input: En sammenhengende graf $G = (V, E)$
Output: En graf der nodene er fargelagt hvis det er mulig

```
1  Procedure TwoColor(G)
2  |  v ← arbitrary v ∈ V
3  |  TwoColorDFS(G, v, R)
4
5  Procedure TwoColorDFS(G, u, C)
6  |  if u.color = null then
7  |  |  u.color ← C
8  |  |  for (u, v) ∈ E do
9  |  |  |  TwoColorDFS(G, v, FlipColor(C))
10 |  else if C ≠ u.color then
11 |  |  Abort with message "impossible"
```

For full uttelling må det gis en godt presentert løsning med dybde- eller bredde først søk (eller en annen god løsning vi ikke har tenkt på enda). For en god beskrivelse av idéen kan det gis full uttelling, hvis den er svært presist formulert (altså, like presist som pseudokode).

Det gis opp til 7 poeng for riktig idé, men med vesentlige feil i pseudokoden.

Det gis ingen poeng for idéer som ikke vil lede til et korrekt svar. For idéer som gir riktig svar, men som vil ha veldig dårlig kjøretid kan det gis opp til 3 poeng.

Whops!-oppgjør 10 poeng

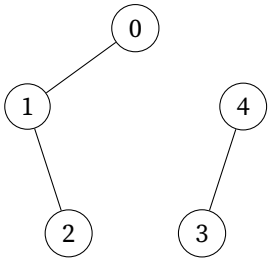
Ansatte på Institutt for informatikk har vært på instituttseminar hvor det har sklidd helt ut med utlån av små beløp mellom hverandre. De skal alle gjøre opp for seg ved å overføre penger via tjenesten Whops!. Dessverre har det oppstått et virvar av interne stridigheter mellom de ansatte, hvor mange har endt opp med å blokkere hverandre på Whops!, som gjør det vanskelig å få oppgjøret til å gå opp. Administrasjonen ser fortvilet på situasjonen, og skjønner at dette problemet vil oppstå igjen i årene fremover, hvor det også skal ansettes vanvittig mange. De har rutiner på plass for å få oversikt over hvor mye hver ansatt har lagt ut eller skylder totalt. Nå ber de om hjelp fra studentene i IN2010 (som på dette tidspunktet har blitt eksperter på algoritmer og datastrukturer) til å finne en generell måte å sjekke om oppgjøret kan gjennomføres gjennom Whops! eller ikke, som også vil skalere etterhvert som Institutt for Informatikk får ubegripelig mange ansatte.

Du setter flittig i gang, og formaliserer problemet som et grafproblem. Du lar $G = (V, E)$, der hver ansatt være representert ved en node $v \in V$, og lar det være en (urettet) kant $\{u, v\} \in E$ mellom to ansatte dersom de ikke har blokkert hverandre på Whops!. For hver ansatt $v \in V$ kan du slå opp i en tabell `T`, der `T[v]` er et heltall som angir hvor mye v skylder eller har lagt ut totalt. Dersom `T[v]` er negativ, betyr det at v skylder penger, men dersom `T[v]` er positiv betyr det at v har lagt ut penger. Du vet at summen av alle tallene i `T` er nøyaktig 0. Merk at det betyr at dersom alle kunne utveksle penger mellom hverandre gjennom Whops! (muligens via andre ansatte) ville oppgjøret alltid gått opp.

Du skal gi en algoritme som tar G og `T` som input og svarer **`true`** hvis oppgjøret kan gjøres gjennom Whops!, og **`false`** dersom det ikke er mulig.

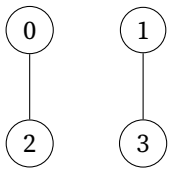
Her er to eksempler på hvordan en G og `T` kan se ut. For eksempelet med G_1 og T_1 kan oppgjøret gjøres gjennom Whops!, men for G_2 og T_2 er det ikke mulig å gjøre oppgjøret gjennom Whops!.

G_1



v	$\mathsf{T}_1[v]$
0	500
1	−200
2	−300
3	−50
4	50

G_2



v	$\mathsf{T}_2[v]$
0	300
1	−250
2	0
3	−50

Innsikten som trengs for å løse denne oppgaven er at dersom summen (med hensyn til T) i hver av komponentene av grafen G er 0, så går oppgjøret opp med Whops!. Vi kan samle komponenten som tilhører en node v, ved å gjøre et dybde-først (eller bredde-først) fra v. Ved et fullt dybde-først søk samler vi komponentene for hver node $v \in V$. Til slutt kan vi gå gjennom hver komponent, og sjekke at summen av `T[`v`]` for alle v i komponenten er 0.

I denne oppgaven er det viktigere at studenten klarer å kommunisere idéen bak algoritmen godt, enn å skrive pseudokode som veldig enkelt lar seg oversette til et konkret programmeringsspråk. Det er helt greit om studenten gjør antagelser om representasjonen av grafen (for eksempel en nabomatrise, nabolister eller en mer objekt-orientert stil). Løsningsforslaget nedenfor er ganske detaljert, men binder seg ikke til noen spesifikk representasjon av grafen.

Input: En graf $G = (V, E)$ og en tabell av heltall T
Output: **true** hvis oppgjøret går opp i Whops!, **false** ellers.

```
1  Procedure ComponentOf(s, G)
2      component ← empty set
3      stack ← [s]
4      while stack is not empty do
5          v ← pop stack
6          add v to component
7          for (v, u) ∈ E do
8              if u ∉ component then
9                  add u to stack
10     return component
11
12 Procedure ComponentsOf(G)
13     components ← empty set
14     visited ← empty set
15     for v ∈ V do
16         if v ∉ visited then
17             C ← ComponentOf(v, G)
18             add C to components
19             add every c ∈ C to visited
20     return components
21
22 Procedure Settleable(G, T)
23     for C in ComponentsOf(G) do
24         if Σ_{v∈C} T[v] ≠ 0 then
25             return false
26     return true
```

En setning som tilsvarer «Sjekk at alle komponenter summerer til 0 med hensyn til T » gir opp til 5 poeng. De resterende 5 poengene fordeles etter hvor gode og presise beskrivelser studenten gir av hvordan dette skal beregnes (altså, hvordan DFS kan brukes for å beregne komponentene, og hvordan vi sjekker summene med hensyn på T).

Whops!-logger 10 poeng

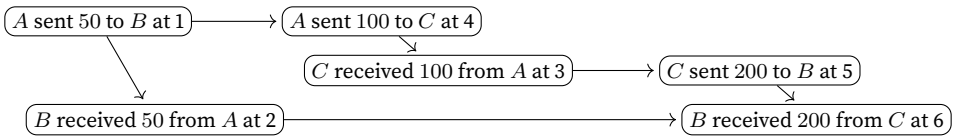
Tjenesten Whops! lar brukere overføre penger mellom hverandre gjennom en app. De prøver i så stor grad som mulig å unngå å lagre informasjon om brukerene sine sentralt, og lar heller informasjonen lagres på brukeren sin egen mobiltelefon.

En gang i blant har de behov for å anskaffe en komplett logg av overføringer for en periode for å kunne feilsøke systemene sine. Da henter de inn *lokale* logger fra noen utvalgte mobiltelefoner i systemet. En lokal logg er en liste av hendelser, der hver hendelse beskriver enten at et beløp er sent eller mottatt. I tillegg inneholder hver hendelse et tidsstempel som forteller når beløpet ble sendt eller motatt. For enkelhets skyld er tidsstemplene representert som positive heltall. Her er et lite eksempel på tre logger fra mobiltelefoner A, B og C.

A	B	C
A sent 50 to B at 1	B received 50 from A at 2	C received 100 from A at 3
A sent 100 to C at 4	B received 200 from C at 6	C sent 200 to B at 5

Dessverre er tidsstemplene fra mobiltelefonene upålitelige. De observerer ofte at et mottak av et beløp forekommer *før* beløpet ble sendt, i følge tidsstemplene. I eksempelet ovenfor mottar C et beløp på 100 *før* A har sendt beløpet til C.

For å approksimere en riktig logg lar de hver hendelse fra de lokale loggene være noder i en rettet graf $G = (V, E)$. Dersom to hendelser u og v kommer direkte etter hverandre i en lokal logg, så finnes det en rettet kant (u, v) i grafen. For hver hendelse u som representerer at et beløp er sendt, så finnes det en rettet kant (u, v) der v er hendelsen for det korresponderende mottaket av det beløpet. Her er grafen for eksempelet ovenfor:



Den ønskede komplette loggen for dette eksempelet skal se slik ut:

A sent 50 to B at 1
B received 50 from A at 2
A sent 100 to C at 4
C received 100 from A at 3
C sent 200 to B at 5
B received 200 from C at 6

Hver node v kan aksessere tidsstempelet fra loggen ved $v.\texttt{ts}$. Skriv en algoritme som tar rettet graf $G = (V, E)$ som input, og skriver ut alle noder i V i en rekkefølge slik at:

- Dersom det finnes en kant fra u til v så skal u skrives ut før v.
- u skal skrives ut før v hvis $u.\texttt{ts} < v.\texttt{ts}$, så lenge det ikke bryter med det første kravet.

Denne oppgaven kan løses med en litt modifisert topologisk sortering. I en standard topologisk sortering av nodene plasserer alle noder med inngrad 0 på en stack. For hver node u som besøkes, så fjerner vi alle kanter (u, v), og plasserer v på stacken dersom den får inngrad 0 som en konsekvens av å ha besøkt u. Dette er tilstrekkelig for å garantere at dersom det er en sti fra u til v, så skrives u ut før v. Merk at det ikke spiller noen rolle for den topologiske sorteringen om vi bruker en stack, en kø, eller en annen datastruktur med konstant tid på innsetting og uttak.

Det andre kravet sier at dersom det *ikke* er en sti fra u til v, så skal u skrives ut før v hvis $u.\texttt{ts} < v.\texttt{ts}$; med andre ord vil vi *sortere* nodene etter tidsstempelet, enn så lenge det ikke bryter med det første kravet. Som allerede nevnt, er det ikke viktig hva slags datastruktur vi bruker for å lagre nodene som er klare for å bli besøkt for at den topologiske sorteringen skal være korrekt. Dermed kan vi bruke en *heap* som ordnes etter tidsstemplene, og dermed garantere at vi hele tiden besøker den noden med det minste tidsstempelet, uten å bryte med den topologiske ordningen.

Input: En rettet graf $G = (V, E)$
Output: Skriv ut nodene i V slik kantene i grafen respekteres, og tidsstemplene minimeres.

```
1  Procedure TopSortLogs(G = (V, E))
2      H ← empty heap, ordered by ts
3      for v ∈ V do
4          if indegree of v is 0 then
5              add v to H
6      while H is not empty do
7          u ← pop smallest from H
8          print(u)
9          for (u, v) ∈ E do
10             remove u from the incoming edges of v
11             if indegree of v is 0 then
12                 add v to H
```

En umodifisert topologisk sortering gir opp til 5 poeng. Sortering etter tidsstemplene, uten å ta høyde for kantene i grafen gis opp til 2 trøstepoeng.

En modifisert topologisk sortering, som løser problemet i kvadratisk tid gis opp til 8 poeng. For full uttelling må algoritmen kjøre i $O((|V| + |E|) \log |V|)$.