Utsatt eksamen i IN2010 høsten 2021

13. Januar 2022

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.

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.

Litt sortering **10 poeng**

(a) Heap sort er *stabil.*

(b) Merge sort er *stabil.*

(c) Quicksort er *stabil.*

(d) Radix sort er *stabil.*

(e) Heap sort er *in-place.*

(f) Merge sort er *in-place.*

(g) Quicksort er *in-place.*

(h) Radix sort er *in-place.*

(i) Verste tilfelle kjøretidskompleksitet for Quicksort er $O(n \cdot \log(n))$.

(j) Merge sort har bedre verste kjøretidskompleksitet enn Quicksort.

(k) Å sette alle elementer inn i et AVL-tre, etterfulgt av en in-order traversering gir en $O(n \cdot \log(n))$ sorteringsalgoritme.

(l) Heap sort er spesielt godt egnet for *nesten* sortert input.

Litt hashing **3 poeng**

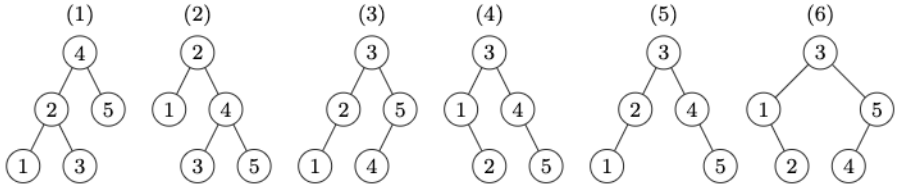
(a) Seperate chaining er mer effektivt med hensyn til minnebruk enn linear probing.

(b) Både separate chaining og linear probing vil gi riktig svar for innsetting, oppslag og sletting, dersom man bruker en hashfunksjon som alltid returnerer 0.

(c) Separate chaining kan kombineres med AVL-trær, og få $O(\log(n))$ på innsetting, oppslag og sletting i *verste tilfelle*.

Balanserte søketrær — 6 poeng

Følgende er alle AVL-trær som inneholder tallene 1 til 5.



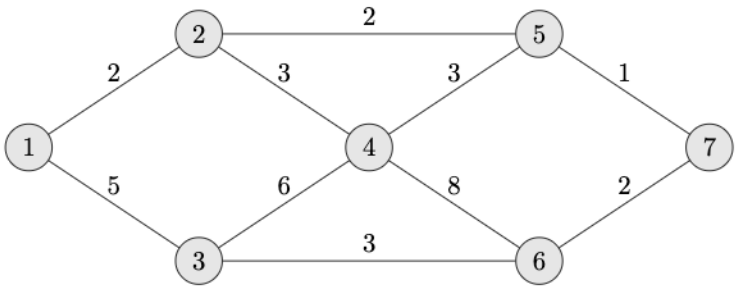
(a) Hvilket av AVL-trærne ovenfor får man hvis man legger inn tallene 1 til 5 i rekkefølgen $2, 4, 3, 1, 5$? Oppgi svaret som et tall mellom 1 og 6.

(b) Hvor mange av trærne ovenfor kan fargelegges som et rød-svart tre?

(c) Hvis vi legger til 6 i alle AVL-trærne ovenfor, i hvilke trær vil det forekomme rotasjoner? Oppgi svaret som summen av alle trærne det gjelder. Altså, hvis det kun forekommer rotasjoner i det første treet, er svaret 1, og hvis det forekommer rotasjoner i alle trærne, er svaret 21 (fordi $1 + 2 + 3 + 4 + 5 + 6 = 21$).

Vektede grafer **5 poeng**

Ta utgangspunkt følgende grafen G:



(a) Hva er vekten på det minimale spenntreet til G?

(b) G er to-sammenhengende. Det finnes noder som kan fjernes fra G som gjør at G ikke lenger er to-sammenhengende. Hvilke noder er det? Oppgi svaret som summen av alle noder det gjelder. Altså, hvis det kun gjelder node 1, er svaret 1, og hvis det gjelder alle nodene, er svaret 28 (fordi $1+2+3+4+5+6+7 = 28$).

Litt av hvert **8 poeng**

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

(a) Alle trær er lister.

(b) Alle trær er grafer.

(c) Dybde-først søk har lavere kjøretidskompleksitet enn bredde-først søk.

(d) Det er mulig å gjøre et usortert array om til en binær heap på lineær tid.

(e) Djikstras algoritme for korteste stier har samme kjøretidskompleksitet som Prims algoritme for minimale spenntær.

(f) For en vektet graf finnes det alltid et *unikt* minimalt spenntre.

(g) Å sortere et array med heltall, der hvert tall er mellom 0 og 2^{32}, kan gjøres på lineær tid.

(h) Det kan være en sykel mellom tre noder i en rettet graf som tilhører tre ulike sterkt sammenhengende komponenter.

Kjøretid på grafalgoritmer **8 poeng**

For hver grafalgoritme, kryss av på den (laveste) korrekte kjøretidskompleksiteten.

	$O(1)$	$O(\|V\| + \|E\|)$	$O((\|V\| + \|E\|) \cdot \log(\|V\|))$	$O(\|V\| \cdot \|E\|)$
Bredde-først søk				
Dijkstra				
Bellman-Ford				
Kosaraju				

Korte beskrivelser av algoritmene:

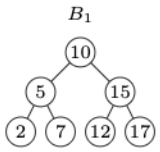
- Bredde-først søk: Traverserer grafen lag for lag.
- Dijkstra: Finner korteste stier fra én til alle andre noder.
- Bellman-Ford: Finner korteste stier fra én til alle andre noder.
- Kosaraju: Finner sterkt sammenhengende komponenter.

Finn nærmeste i binære søketrær — 10 poeng

Du er gitt et binært søketre B og et heltall x. Du skal gi en algoritme som finner den *nærmeste* verdien til x. Det vil si at du skal finne verdien y i B slik at $|x - y|$ er så liten som mulig. Dersom det finnes to verdier i B som er like nærme x spiller det ingen rolle hvilken av verdiene som returneres. Du kan anta at B inneholder minst ett element.

Input: Rotnoden v av et binært søketre B og et heltall x
Output: Returnerer verdien y i treet som er nærmest x

```
1 Procedure FindClosest(v, x)
  | // ···
```



For det binære søketreet B_1 (vist i eksempelet ovenfor) skal for eksempel `FindClosest(`B_1`, 5)` returnere 5, `FindClosest(`B_1`, 9)` returnere 10, `FindClosest(`B_1`, 13)` returnere 12 og `FindClosest(`B_1`, −100)` returnere 2.

(a) Oppgi algoritmen din (med psuedokode). En mer effektiv algoritme er mer poenggivende. Hint: Du kan bruke ∞ (eller `inf`) som en verdi som er lenger unna x enn noen annen verdi i B.

(b) Hvis B er et vilkårlig binært søketre med n elementer, hva er kjøretidskompleksiteten på algoritmen din?

(c) Hvis B er et AVL-tre med n elementer, hva er kjøretidskompleksiteten på algoritmen din?

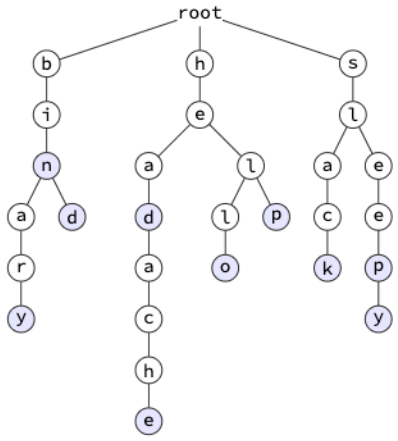
Auto complete — 10 poeng

De fleste teksteditorer har funksjonalitet for «auto completion», altså der det dukker opp en liten boks med mulige måter å fullføre et påbegynt ord. Nedenfor gir vi to mulige datastrukturer som kan brukes for å implementere en enkel mekanisme for å fullføre et påbegynt ord.

Vi viser eksempler på hvordan datastrukturene ser ut dersom de lagrer ordene: `bin`, `binary`, `bind`, `head`, `headache`, `hello`, `help`, `slack`, `sleep`, `sleepy`.

Strategi 1

Den første strategien er å bruke et *prefiks-tre* som holder på strenger. Et prefiks-tre har en rot. Hvis en streng s er lagt inn i treet, har rotnoden et barn med s sin første bokstav. Den noden har igjen en peker til s sin andre bokstav, og så videre. Den siste bokstaven i s er markert som en *terminal* node (terminalnodene er fargelagt blå i eksempelet). Det vil si at hvis du følger en sti fra rotnoden til en terminalnode, så har du stavet et ord som er lagt inn i treet.



Fra et prefiks-tre er det enkelt å finne alle måter å fullføre et påbegynt ord. Det gjøres ved å returnere alle stier fra rotnoden som begynner med bokstavene fra et gitt ord og ender i en terminalnode.

Strategi 2

Den andre strategien er å bruke et hashmap, der vi lar hver prefiks av et ord mappe til en liste av alle mulige måter å fullføre ordet. Hvis H er et hashmap, og s er en streng som skal legges til, ser vi på hver prefiks p av s og legger s til i listen H[p]. I eksempelet ved siden av ser dere hvordan hashmapet kan se ut dersom alle eksempelordene er lagt (merk at noen av listene er kortet ned, indikert ved «...»).

Nøkkel	Verdi
`""`	`"bin", ..., "sleepy"`
`"b"`	`"bin", "binary", "bind"`
`"bi"`	`"bin", "binary", "bind"`
`"bin"`	`"bin", "binary", "bind"`
`"bina"`	`"binary"`
`"binar"`	`"binary"`
`"binary"`	`"binary"`
`"bind"`	`"bind"`
`"h"`	`"head", ... "help"`
`"he"`	`"head", ... "help"`
`"hea"`	`"head", "headache"`
`"head"`	`"head", "headache"`
`"headа"`	`"headache"`
`"headac"`	`"headache"`
`"headach"`	`"headache"`
`"headache"`	`"headache"`
`"hel"`	`"hello", "help"`
`"hell"`	`"hello"`
`"hello"`	`"hello"`
`"help"`	`"help"`
`"s"`	`"slack", "sleep", "sleepy"`
`"sl"`	`"slack", "sleep", "sleepy"`
`"sla"`	`"slack"`
`"slac"`	`"slack"`
`"slack"`	`"slack"`
`"sle"`	`"sleep", "sleepy"`
`"slee"`	`"sleep", "sleepy"`
`"sleep"`	`"sleep", "sleepy"`
`"sleepy"`	`"sleep"`

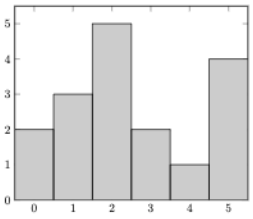
Fra et slikt hashmap er det svært enkelt å finne alle måter å fullføre et påbegynt ord. Det gjøres ved å slå opp det påbegynte ordet i hashmapet og returnere listen.

Drøft fordeler og ulemper ved de to ulike strategiene.

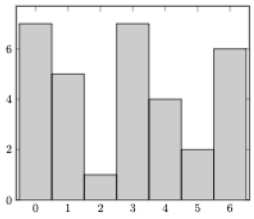
Fyll et histogram med vann — 10 poeng

Her er to histogrammer:

H_1

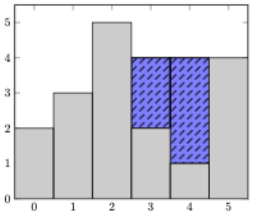


H_2

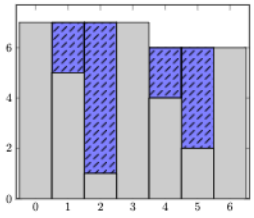


Nå heller vi vann over histogrammene til det blir så fullt som mulig. Vannet som blir liggende i histogrammet forteller oss hvor mye vann histogrammet rommer:

H_1



H_2



Du skal gi en algoritme som finner ut hvor mye vann et histogram rommer. Input er et array H med positive heltall, der $\mathsf{H}[i]$ representerer hvor høy søylen på plass i er.

Input: Et histogram, representert som et array H av positive heltall med størrelse n
Output: Returnerer et heltall som sier hvor mye vann histogrammet rommer

```
1 Procedure HistogramCapacity(H)
  | // ···
```

For H_1 er array-representasjonen $[2, 3, 5, 2, 1, 4]$ og `HistogramCapacity(`H_1`)` skal returnere 5. For H_2 er array-representasjonen $[7, 5, 1, 7, 4, 2, 6]$ og `HistogramCapacity(`H_2`)` skal returnere 14.

(a) Oppgi en algoritme som finner hvor mye vann et histogram rommer (med pseudokode). Lavere kjøretidskompleksitet er mer poenggivende.

(b) Oppgi kjøretidskompleksiteten for algoritmen din.

Garbage collection — 8 poeng

Mange moderne programmeringsspråk har en *garbage collection*, som er en prosedyre som frigjør minne som garantert ikke vil brukes i programmet lenger. Du skal utlede en enkel algoritme for garbage collection.

Vi kan anta at alt som lagres er *objekter*, der et objekt kan referere til andre objekter. Vi lar hvert objekt være representert som en node i en objektgraf $G = (V, E)$, der V er alle objektene som er opprettet, og en (rettet) kant fra u til v betyr at objektet u har en referanse til objektet v. I tillegg har vi en mengde R med alle objektene som kan refereres til direkte (typisk objekter som refereres til av programvariabler). Alle objekter i R er også i V. Objekter det *ikke* finnes en referanse til er garantert å ikke bli brukt i programmet, og skal derfor frigjøres.

Det betyr at ingen av objektene i R skal frigjøres, og dersom et objekt i R har en referanse til et objekt, så skal dette objektet heller ikke frigjøres, etc...

Du skal gi en prosedyre `GarbageCollect` som tar en graf $G = (V, E)$ og en mengde R som input, og frigjør alle objekter det *ikke* finnes en referanse til. Du kan anta at du har en prosedyre `Free` som frigjør et objekt.

Merk at garbage collection kjøres på objektgrafen flere ganger i løpet av programmets levetid.

Input: En objektgraf $G = (V, E)$ og en mengde R med objekter
Output: Frigjør alle objekter det ikke finnes en referanse til

```
1 Procedure GarbageCollect(G, R)
  | // ···
```

Whops!-nettverk **10 poeng**

Tjenesten Whops! lar brukere overføre penger mellom hverandre gjennom en app. På hver mobiltelefon har brukeren en bekjentliste, som angir hvilke Whops!-brukere som brukeren kan utveksle penger med.

Ved hver overføring følger det med en liten avgift som er avhengig av hvilke banker de to respektive brukerene har. For eksempel kan brukere som har samme bank kunne overføre penger mellom hverandre helt uten ekstra kostnad.

Nettverket av brukere er gitt som en *urettet* og *vektet* graf $G = (V, E)$. Hver bruker er representert som node i V, og det er en urettet kant $\{u, v\}$ mellom u og v dersom de er i bekjentlistene til hverandre. I tillegg angir $w(u, v)$ kostnaden av å overføre penger mellom u og v.

Ved en overføring mellom to brukere s og t kan Whops! dele opp overføringen i flere ledd, der den totale avgiften fra hver overføring belastes hos brukeren s.

(a) Gi en algoritme (eller gjengi en algoritme fra kurset) som finner den laveste samlede kostnaden assosiert med å overføre et beløp mellom to noder s og t. Dersom du bruker en algoritme kjent fra kurset skal du også beskrive hvordan den virker.

En ny forskrift gjør at Whops! nå er nødt til å opprette en avtale mellom hvert par av banker det skal kunne overføres penger mellom. Hver slik avtale koster penger å vedlikeholde, så naturligvis ønsker de å inngå så få avtaler som mulig, men der det fremdeles er mulig for alle brukere å overføre penger mellom hverandre.

De har igjen tilgang på en graf, der hver bank er en node, og mellom hver node er det en urettet kant med en vekt som angir avgiften som er assosiert med å overføre et beløp mellom de to bankene.

(b) Foreslå en algoritme fra kurset som lar Whops! bergene hvilke avtaler de bør opprette, slik at alle brukere fremdeles kan overføre penger mellom hverandre, der de både oppretter et minimalt antall avtaler og får så lave avgifter som mulig. Forslaget bør begrunnes *kort*.

Whops!-gjeld **10 poeng**

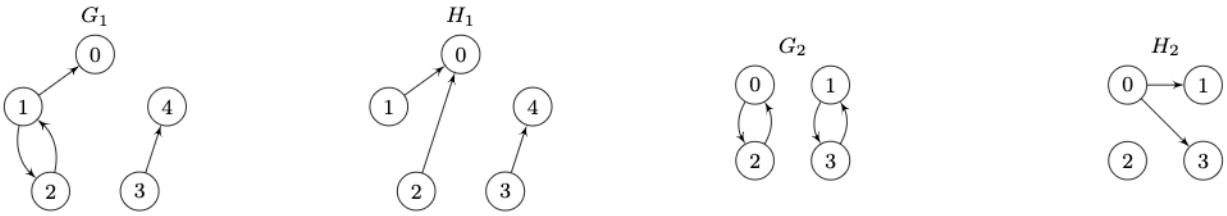
> Merk at konteksten for denne oppgaven er lik en oppgave som ble gitt på den ordinære eksamen. Input for algoritmen er *ikke* likt, så det er viktig å forholde seg til oppgaven, slik den er gitt her. Merk også at denne oppgaven er helt uavhengig fra forrige oppgave.

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 hvilke ansatte som skylder hverandre penger. 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 $G = (V, E_G)$, der hver ansatt er representert ved en node $v \in V$, og lar det være en *rettet* kant $(u, v) \in E_G$ dersom u kan overføre penger til v (altså har u *ikke* blokkert v på Whops!). Så konstruerer du en gjelds-graf $H = (V, E_H)$, der det er en rettet kant $(u, v) \in E_H$ mellom to ansatte hvis u skylder v penger. Merk at G og H består av de samme nodene. Dersom alle kunne utveksle penger mellom hverandre gjennom Whops! (muligens via andre ansatte) ville oppgjøret alltid kunne gjennomføres med Whops!.

(a) Du skal gi en algoritme som tar G og H 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 H kan se ut. For eksempelet med G_1 og H_1 kan oppgjøret gjøres gjennom Whops!, men for G_2 og H_2 er det ikke mulig å gjennomføre oppgjøret med Whops!.



(b) Du får nå vite at Whops! har begynt å stenge kontoer som er blokkert av mer enn k andre brukere (under antagelsen at dette må være spam eller bare høyst urimelige kunder), der k er veldig liten til sammenligning med antall noder. Med andre ord vil det være en kant mellom de aller fleste noder dersom G er stor, og inngraden til node i G er minst $|V| - k - 1$. Du kan fremdeles anta at G og H består av de samme nodene.

Beskriv (med naturlig språk) hvordan du kan bruke denne informasjonen til å gjøre algoritmen mer effektiv.