

Innehåll

F10 - Mängd, graf

5DV149 Datastrukturer och algoritmer
Kapitel 13.1–13.2, 17

Niclas Börlin
niclas.borlin@cs.umu.se

2024-04-23 Tis

- Mängd
- Graf
- OU 4

Tenta

- Anmälningarfönstret stänger 2024-04-29

Blank

Mängd

Mängd

- Modell:
 - En **påse** (men man kan inte ha två likadana element)
- Organisation:
 - En **oordnad** samling av element som är av samma typ
 - Grundmängden behöver inte vara ändlig (ex. heltal) men dataobjekten är **ändliga**
 - Kan inte innehålla två element med **likadana värden**
 - En mängd kan **inte innehålla mängder**
 - **Homogen** datatyp
- En mängd som kan innehålla **flera** element med samma värde kallas multimängd (**multiset**) eller påse (**bag**)
 - Vid textsökning betraktas ofta ett dokument som en påse av ord (*bag of words*), där bara ordens frekvens räknas (jfr. google)

Specifikation (1)

```
abstract datatype Set(val)
  Empty() → Set(val)
  Single(v: val) → Set(val)
  Insert(v: val, s: Set(val)) → Set(val)
  Union(s: Set(val), t: Set(val)) → Set(val)
  Intersection(s: Set(val), t: Set(val)) → Set(val)
  Difference(s: Set(val), t: Set(val)) → Set(val)
  Isempty(s: Set(val)) → Bool
  Member-of(v: val, s: Set(val)) → Bool
  Choose(s: Set(val)) → val
  Remove(v: val, s: Set(val)) → Set(val)
  Equal(s: Set(val), t: Set(val)) → Bool
  Subset(s: Set(val), t: Set(val)) → Bool
  Kill(s: Set(val)) → ()
```

Specifikation (2)

- Boken har tagit med de vanliga matematiska mängdoperationerna
- Alla behövs inte
- Följande operationer räcker:

```
abstract datatype Set(val)
  Empty() → Set(val)
  Isempty(s: Set(val)) → Bool
  Insert(v: val, s: Set(val)) → Set(val)
  Member-of(v: val, s: Set(val)) → Bool
  Choose(s: Set(val)) → val
  Remove(v: val, s: Set(val)) → Set(val)
  Kill(s: Set(val)) → ()
```

Konstruktion av mängd, dubletter

- ▶ De flesta konstruktioner måste kunna hantera att det inte får finnas dubletter i en mängd
- ▶ Mängd som Lista har två alternativ:
 - ▶ Se till att listan inte har dubletter (krav på `Insert` och `Union`)
 - ▶ Låt listan innehålla dubletter (krav på `Equal`, `Remove`, `Intersection`, `Difference`)

Konstruktion av mängd som lista

- ▶ Komplexitet:
 - ▶ Metoder som kräver **sökningar** i listan: $O(n)$
 - ▶ Binära mängdoperationerna mellan två listor med m och n element har komplexitet $O(mn)$
- ▶ Listan kan konstrueras på olika sätt
 - ▶ **Sorterad** lista är **effektivare** för de binära mängdoperationerna

Konstruktion av mängd som bitvektor (1)

- ▶ En **bitvektor** är en vektor med elementvärden av typen $\text{Bit} = \{0,1\}$
- ▶ Ofta tolkas 0=falskt och 1=sant, dvs Bit identifieras som datatypen Boolean
- ▶ Grundmängden måste ha en **diskret linjär ordning** av elementen (elementen kan *numreras*)
 - ▶ Bit k i bitvektorn motsvarar det k :te elementet i grundmängden
 - ▶ Biten är 1 om elementet ingår i mängden

Konstruktion av mängd som bitvektor (2)

- ▶ Veckoschema måndag-fredag timvis, 08-17 i 64-bitars heltal:

	Mon	Tue	Wed	Thu	Fri
08-09	X			X	
09-10	X			X	
10-11					
11-12					
12-13					
13-14	X			X	
14-15	X			X	
15-16					
16-17					

- [illegible]

Konstruktion av mängd som bitvektor (3)

- ▶ Sökoperationer och binära operationer har komplexitet $O(M)$, där M är antalet element i grundmängden
- ▶ Om bitvektorn finns implementerad som ett eller flera **ord** (*memory words*) kan man utnyttja maskinoperationer
 - ▶ Processorn gör operationen **samtidigt** på alla element i vektorn
 - ▶ Detta gör många metoder effektiva
 - ▶ Ex. för en ordlängd på 64 bitar=8 bytes så tar AND för 64 bitar samma tid som AND för 1 bit

Konstruktion av mängd som bitvektor (4)

- ▶ Grundmängden måste vara ändlig och i praktiken liten
- ▶ Reserverat minne proportionellt mot grundmängdens storlek
 - ▶ Ett veckoschema måndag-fredag, uppdelat i timmar 08-17 kräver $M = 45$ bitar (9h per dag, 5 dagar) = 6 bytes
 - ▶ Ett veckoschema 24/7 uppdelat i 5-minutersintervall kräver $M = 2016$ bitar ($7 \cdot 24 \cdot 12$)=252 bytes
 - ▶ En tabell över upptagna IP4-internetadresser kräver $M = 2^{32} = 4294967296$ bitar ≈ 0.5 Gbyte
- ▶ Få element per mängd utnyttjar minnet ineffektivt

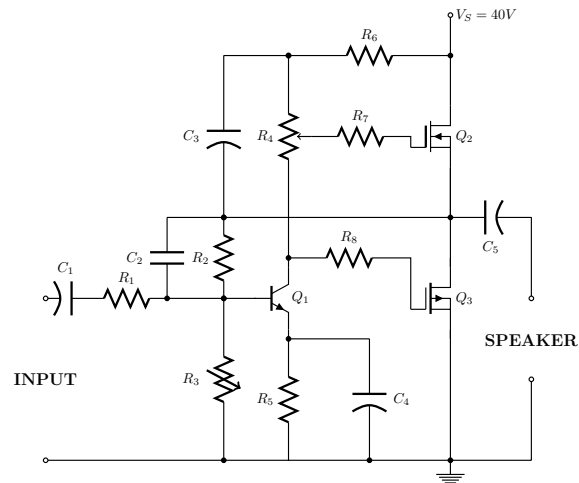
Graf

Graf

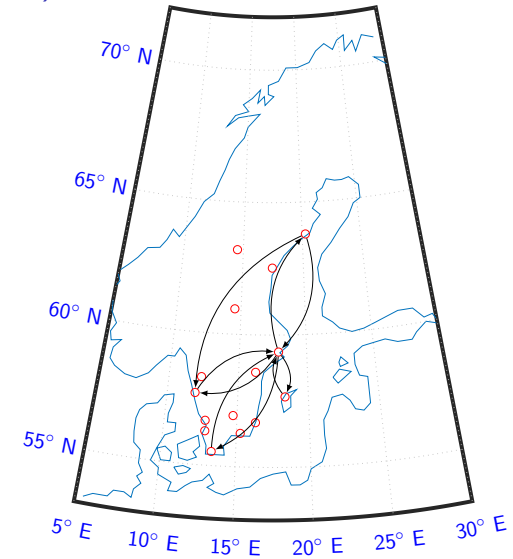
- ▶ Modell: Vägkarta med enkelriktade gator utritade.



Graf, tillämpningar — elektriska kretsar



Graf, tillämpningar — nätverk (gator, flygrutter, kommunikation)



Specifikation av Graf

- ▶ En graf är
 - ▶ en **sammansatt** datatyp — innehåller **noder**
 - ▶ en **homogen** datatyp — alla noder är av **samma typ**
 - ▶ **oordnad** — det finns ingen **första** nod
- ▶ En graf kan vara **riktad** och **oriktad**
- ▶ Specifikationen för Graf kan vara
 - ▶ **mängd-orienterad** eller
 - ▶ **navigerings-orienterad**

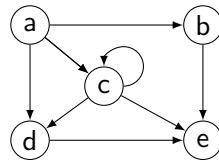
Blank

Mängdorierad specifikation, riktad graf

- ▶ Vanlig inom matematiken
- ▶ En graf $G = (V, E)$ består av:
 - ▶ en mängd V **noder** (*vertices*) och
 - ▶ en mängd E **bågar** (*edges*) som binder samman noderna i V
 - ▶ En **båge** $e = (u, v)$ är ett **ordnat par** av **noder**
 - ▶ Bågen har **riktning**, den går **från** u **till** v
 - ▶ Bågen gör noden v till **granne** till u , men inte tvärtom

▶ Exempel:

$V = \{a, b, c, d, e\},$
 $E = \{(a, b), (a, c), (a, d), (b, e),$
 $(c, c), (c, d), (c, e), (d, e)\}$

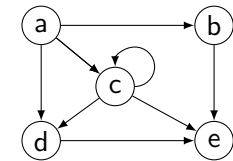


Navigeringsorienterad specifikation, riktad graf

- ▶ En graf är en mängd med **noder**
 - ▶ Till varje nod associeras en **grannskapsmängd** av noder som kallas **grannar**
- ▶ **Bågarna** utgör **ordnade par** av noder bestående av varje nod u och varje nod v i u 's grannskapsmängd
 - ▶ Bågarna är **riktade**:
 - ▶ om noden v ligger i u 's grannskapsmängd så går det en båge **från** u **till** v , men inte tvärtom
 - ▶ bågen gör noden v till **granne** till u , men inte tvärtom

▶ Exempel:

$G = \{ (a, \{b, c, d\})$
 $(b, \{e\})$
 $(c, \{c, e, d\})$
 $(d, \{e\})$
 $(e, \{\}) \}$



Gränsyta graf

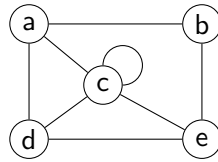
```
abstract datatype Graph(node, edge)
  Empty() → Graph(node, edge)
  Insert-node(n: node, g: Graph(node, edge))
    → Graph(node, edge)
  Insert-edge(e: edge, g: Graph(node, edge))
    → Graph(node, edge)
  Isempty(g: Graph(node, edge)) → Bool
  Has-no-edges(g: Graph(node, edge)) → Bool
  Choose-node(g: Graph(node, edge)) → node
  Neighbours(n: node, g: Graph(node, edge)) → Set(node)
  Delete-node(n: node, g: Graph(node, edge))
    → Graph(node, edge)
  Delete-edge(e: edge, g: Graph(node, edge))
    → Graph(node, edge)
  Kill(g: Graph(node, edge)) → ()
```

Tänkbar informell specifikation

- ▶ **Empty()** — konstruerar en **tom** graf utan noder och bågar
- ▶ **Insert-node**(n , g) — sätter in **noden** n i grafen g
- ▶ **Insert-edge**(e , g) — sätter in **bågen** e i grafen g
 - ▶ Det förutsätts att noderna i e **finns** i grafen
- ▶ **Isempty**(g) — **testar** om grafen g är **tom**, dvs. utan noder och bågar
- ▶ **Has-no-edges**(g) — testar om grafen g saknar **bågar**
- ▶ **Choose-node**(g) — väljer ut en **godtycklig** nod ur grafen g
- ▶ **Neighbours**(n , g) — returnerar mängden av alla **grannar** till noden n i grafen g
- ▶ **Delete-node**(n , g) — tar bort **noden** n ur grafen g
 - ▶ Det förutsätts att n inte ingår i nån båge
- ▶ **Delete-edge**(e , g) — tar bort **bågen** e ur grafen g
- ▶ **Kill**(g) — returnerar alla resurser som upptas av g

Oriktade grafer (1)

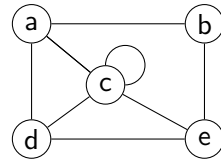
- ▶ I en **oriktad** graf utgör bågar en **mängd** av två noder
 - ▶ Noderna är grannar till **varandra**
 - ▶ Inga riktade bågar kan förekomma
- ▶ I den mångdorierade specifikationen **definierar vi om** bågar till att vara **mängder** $\{u, v\}$ av nodpar



- ▶ Exempel:

$$V = \{a, b, c, d, e\},$$

$$E = \{\{a, b\}, \{a, c\}, \{a, d\}, \{b, e\}, \{c, c\}, \{c, d\}, \{c, e\}, \{d, e\}\}$$



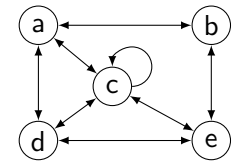
Oriktade grafer (2)

- ▶ I bägge representationerna kan vi implementera en oriktad graf som en **riktad** genom att modifiera **Insert-edge** och **Remove-edge**
 - ▶ Ett anrop till **Insert-edge**(g, a, b) skulle då sätta in **både** bågen från **a** till **b** och från **b** till **a**

$$V = \{a, b, c, d, e\},$$

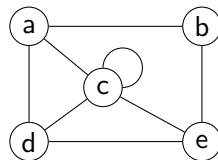
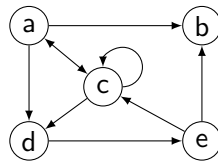
$$E = \{(a, b), (a, c), (a, d), (b, e), (b, a), (c, a), (d, a), (e, b), (c, c), (c, d), (c, e), (d, e), (d, c), (e, c), (e, d), \}$$

$$G = \{ (a, \{b, c, d\}), (b, \{e, a\}), (c, \{c, e, a, d\}), (d, \{e, a, c\}), (e, \{d, c, b\}) \}$$



Gradtal

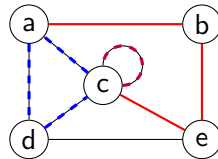
- ▶ Noder i riktade grafer har två gradtal:
 - ▶ Ingradtalet: antalet bågar som går **till** noden.
 - ▶ Utgradtalet: antalet bågar som startar i noden och går till en **annan** nod.
- ▶ Noder i oriktade grafer har ett gradtal:
 - ▶ Gradtalet = Antalet bågar till **grannar** (inklusive sig själv)



Blank

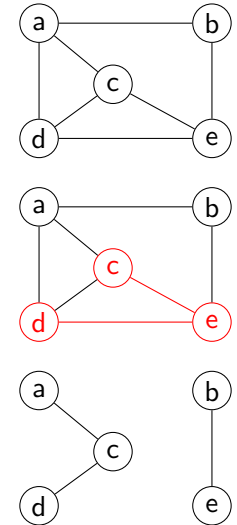
Terminologi, vägar

- ▶ **Väg / stig (path)**: En **sekvens** av noder v_1, v_2, \dots, v_n så att v_i och v_{i+1} är **grannar**
 - ▶ Sekvenserna $a - b - e - c$ och $a - c - d - a$ är vägar
- ▶ **Enkel väg (simple path)**: Inga noder förekommer två gånger i vägen
 - ▶ Vägen $a - b - e - c$ är en enkel väg
- ▶ **Cykel (cycle)**: En väg där den sista noden i sekvensen är densamma som den första
 - ▶ Vägen $a - c - d - a$ är en cykel
 - ▶ Vägen $c - c$ är en cykel



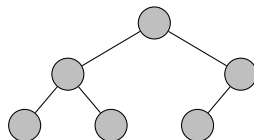
Terminologi, sammanhängande

- ▶ **Sammanhängande (connected) graf**:
 - ▶ Varje nod har en väg till varje annan nod
- ▶ **Delgraf (subgraf)**:
 - ▶ En **delmängd** av noderna och kanterna som **formar en graf**
- ▶ **Icke sammanhängande** med sammanhängande komponenter
 - ▶ Till höger finns **en** graf med **två** sammanhängande komponenter



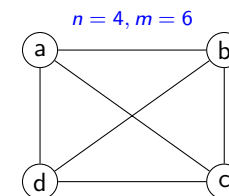
Träd som specialfall av graf

- ▶ Ett **träd** är en sammanhängande, oriktad graf **utan cykler**



Terminologi, närbarhet (Connectivity) (1)

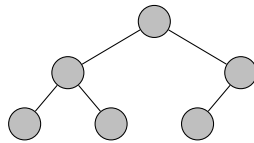
- ▶ Låt n = antalet **noder** och m = antalet **bågar**
- ▶ En **komplett graf (complete graph)** får man när **alla** noder är grannar till **alla andra**
 - ▶ I en komplett, riktad graf är $m = n(n - 1)$
 - ▶ I en komplett, oriktad graf är $m = n(n - 1)/2$



Terminologi, närhet (*Connectivity*) (2)

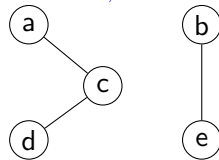
- För ett **träd** gäller $m = n - 1$

$$n = 6, m = 5$$



- Om $m < n - 1$ så kan grafen inte vara sammanhängande

$$n = 5, m = 3$$

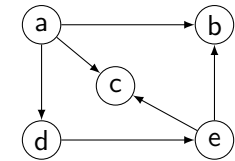


- För en sammanhängande graf varierar m från att vara $O(n)$ (träd) till att vara $O(n^2)$ (komplett graf)

Andra grafer

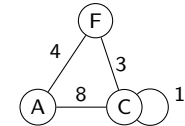
- DAG = *Directed Acyclic Graph*:

- En riktad graf **utan** **cykler**



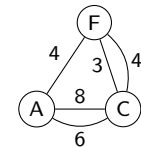
- Viktad graf:

- En graf där bågarna har **vikter**



- Multigraf:

- Tillåtet med **flera** **bågar** mellan två noder
- Bågarna har olika egenskaper som måste lagras



Konstruktion av grafer

- Det finns många sätt att konstruera en graf
- Vi kommer att fokusera på två:
 - Graf med **förbindelsematris** (*adjacency matrix*)
 - Graf som **Fäلت av Lista**

Graf med förbindelsematris (*adjacency matrix*)

- Bågarna representeras av **ettor** i en matris

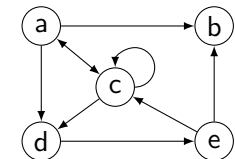
- Rad i visar vilka bågar man kan nå **från** nod i
- Kolumn j visar från vilka noder det kommer bågar **till** nod j

- Enkel att implementera

- Passar också med **vikter** på bågar

- Matrisen kan bli stor

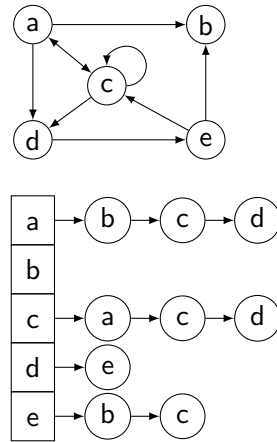
- Minne: $O(n^2)$



	a	b	c	d	e
a	0	1	1	1	0
b	0	0	0	0	0
c	1	0	1	1	0
d	0	0	0	0	1
e	0	1	1	0	0

Graf som Fält av Lista

- ▶ Fältelementen innehåller **en nod** och en lista — **grannskapslistan**
- ▶ Antalet noder fixt (Fält), antalet grannar per nod variabelt (Lista)
- ▶ Inte lika utrymmeskrävande som en matris
 - ▶ Utrymmet $O(m + n)$



Introduktion till OU4

OU4 — Grafer och grafalgoritmer

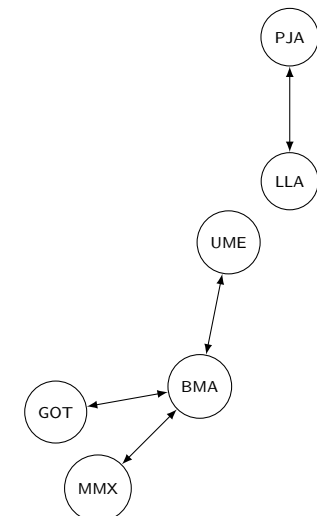
- ▶ Skriv ett program som:
 1. Läser in en beskrivning av en **riktad graf** (textfil)
 2. Besvarar frågor om det finns någon **väg mellan noderna**
 - ▶ För att svara på frågan så ska en bredden-först-traversering göras i grafen (nästa föreläsning)
- ▶ Programmet ska ta namnet på en fil med grafbeskrivningen som **kommandoradsparameter**
- ▶ Frågorna ska ställas **interaktivt** till användaren

Kartfil

- ▶ Exempel på kartfil

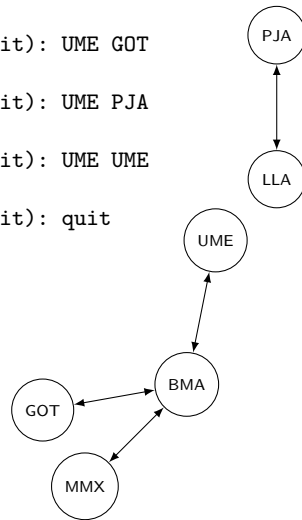
```
# Some airline network
8
UME BMA # Umea-Bromma
BMA UME # Bromma-Umea
BMA MMX # Bromma-Malmo
MMX BMA # Malmo-Bromma
BMA GOT # Bromma-Goteborg
GOT BMA # Goteborg-Bromma
LLA PJA # Lulea-Pajala
PJA LLA # Pajala-Lulea
```

- ▶ Motsvarande graf



Exempelkörning

```
> ./isConnected airmap1.map
Enter origin and destination (quit to exit): UME GOT
There is a path from UME to GOT.
Enter origin and destination (quit to exit): UME PJA
There is no path from UME to PJA.
Enter origin and destination (quit to exit): UME UME
There is a path from UME to UME.
Enter origin and destination (quit to exit): quit
Normal exit.
```



Design av grafen

- ▶ Ni ska **designa och implementera** grafen själva
- ▶ Ni får **gränssytan** specificerad i en fil `graph.h`
- ▶ Ni **måste följa gränssytan**
- ▶ Ni ska implementera **två** versioner av grafen (i filerna `graph1.c` och `graph2.c`)
 - ▶ Den ena designen är **fri**
 - ▶ Den andra måste använda en **grannskapsmatris**
- ▶ Ni ska också skriva ett huvudprogram i `is_connected.c`
 - ▶ Funktionerna i `is_connected.c` använder någon av graf-versionerna
 1. inläsning av filen,
 2. uppbyggnad av grafen,
 3. interaktionen med användaren och
 4. sökalgoritmen
- ▶ Implementerat korrekt kommer `is_connected.c` att gå att kompilera **utan förändringar** tillsammans med `graph1.c` eller `graph2.c` och fungera korrekt

Obligatorisk uppgift 4

- ▶ Börja **i tid** med uppgiften!
- ▶ Redan nu kan ni t.ex. börja fundera på **inläsning från fil** och hur ni vill **konstruera grafen**
 - ▶ Sikta på att ha klarat av detta **före tentan**
- ▶ Algoritmerna för **traversering** kommer att gås igenom på nästa föreläsning
- ▶ Ni får jobba i **enskilt**, eller i grupper på upp till **tre** personer
 - ▶ Bästa pedagogiska effekt får ni om ni jobbar i **grupper om tre**
 - ▶ En person fokuserar på `graph1.c`
 - ▶ En person fokuserar på `graph2.c`
 - ▶ En person fokuserar på `is_connected.c`
 - ▶ Alla ska förstå allas lösningar

Feedback från LP3 (1)

- ▶ Student 1
 - ▶ "This assignment has been a challenging one, **starting with it early helped me tremendously**. As per usual working with C, the memory handling has been frustrating, but in this assignment I felt that a lot of what I learned from the previous ou3 assignment translated over to working with ou4. Especially when it came to solving double frees and segmentation faults. To my surprise the part of the assignment I got stuck on the longest was doing the breadth-first traversal. The reason for that boiled down to the fact that I had not understood my code fully. I was completely perplexed to as why I was receiving segmentation fault. What I thought was a very logical solution ended up being completely illogical to how I had implemented my graph. **I did not find the flaw until I used a debugging program** and carefully observing what happened to the memory addresses."

Feedback från LP3 (2)

► Student 2+3

- "Den största utmaningen med denna övning var att komma igång, det kändes motigt att gå igenom och labb-specifikationen som är väldigt omfattande. När det väl var gjort och vi kände att vi hade greppat vad som behövde göras gick det lättare. Det var dock även en stor utmaning att implementera algoritmerna för programmet och få alla pusselbitar att falla på plats. *I efterhand inser vi att vi hade sparat oss en del lidande om vi hade bitit i det sura äpplet och tagit oss igenom labb-specifikationen lite tidigare i arbetsförloppet.*"

Feedback från LP3 (3)

► Student 4 (jobbade i par)

- Jag tycker den här uppgiften var spännande och lärorik. Den var krävande på ett annat sätt än de tidigare uppgifterna, eftersom vi fick bestämma själva hur grafen skulle implementeras, vilket var kul. *Planeringsfasen var den viktigaste delen. Vi upptäckte tidigt brister i våra initiala ideér, och kunde snabbt avfärda dem, utan en rad kod. Det gjorde att när vi väl kom till utvecklingsfasen så gick det snabbt framåt. Båda visste exakt hur programmet skulle fungera, så det fanns inga tvetydligheter.* Användandet av git underlättade också avsevärt. Vi kunde lugnt delegera vad som skulle göras, och låta git sköta sammanfogningen av vår kod. Anton är väldigt rolig att jobba med, eftersom vi har samma mål med programmet, och båda tycker det är väldigt kul med programmering, vilket förstås underlättar mycket.

► Student 5+6

- ... *Denna uppgift har man verkligen behövt läsa igenom noga för att inte slösa massvis med tid på ofunktionella implementationer.*