

Algoritmer og datastrukturer for Norsk Informatikk Olympiade (NIO)

fredrik.anfinsen@gmail.com

23. november 2014

Innhold

1	Introduksjon	2
1.1	Om NIO-oppgavene	3
1.2	Installasjon av C++ editor og kompilator	4
1.3	Forkunnskaper	11
1.4	Oppgaver i heftet	11
1.5	Oppgaver	12
2	Binærsøk	14
2.1	Binærsøk i STL	15
2.1.1	lower_bound	16
2.1.2	upper_bound	16
2.1.3	binary_search	16
2.2	NIO oppgaver med binærsøk	17
2.3	Oppgaver	18
3	Kompleksitet	19
3.1	Big-Oh notasjon	19
4	Matematikk	21
4.1	Binære tall	21
4.2	Binære operasjoner	22
4.2.1	AND	22
4.2.2	OR	23
4.2.3	XOR	23
4.2.4	NOT	24
4.2.5	Bit-manipulasjon	24
4.3	Fakultet	26
4.4	Binomialkoeffisient	26
4.5	Utvalg (2^n)	29
4.6	Permutasjoner	31
4.7	Modulo	32
4.8	Potenser	33
4.9	Printall	34
4.10	Diverse	35

4.10.1	Finne sifrene i et tall	35
4.10.2	GCD og LCM	36
4.11	Oppgaver	38
5	Datastrukturer	40
5.1	Union-Find	40
5.2	Binary search tree (BST)	42
5.3	Segment tree	46
5.4	Fenwick tree	48
5.4.1	Fenwick tree i 2-dimensjoner	52
5.5	Oppgaver	54
6	Grafer	55
6.1	Notasjon	55
6.2	Representasjon av grafer	56
6.2.1	Nabomatrise	56
6.2.2	Naboliste	57
6.3	Dybde først søk (DFS)	58
6.3.1	Bicoloring	59
6.4	Bredde først søk (BFS)	62
6.5	Minimum spanning tree	65
6.5.1	Kruskal algoritme	66
6.5.2	Prims algoritme	68
6.6	Shortest path algoritmer	69
6.6.1	Dijkstra	70
6.6.2	Bellman Ford	72
6.6.3	Floyd Warshall (all-pair shortest path)	73
6.7	Topologisk sortering	74
6.7.1	Kahns algoritme	75
6.7.2	Rekursiv algoritme	77
6.8	Oppgaver	79
7	Dynamisk programmering	81
7.1	Memoisering (top-bottom)	81
7.2	Bottom-up	83
7.3	Longest increasing sequence (LIS)	83
7.3.1	LIS (n^2)	84
7.3.2	LIS ($n \log n$)	85
7.4	Bitmask	87
7.5	Oppgaver	89

1 Introduksjon

Dette heftet er ment for å utvide nivået i NIO-finalen og at deltagere skal være bedre forberedt på hva som kan komme / kreves i en eventuell finale. Heftet kan også brukes som et oppslagsverk når man skal løse oppgaver. Noen deler av heftet som matematikk-kapittelet kan også være nyttig til første inntaksrunde til NIO-finalen. Kapitler markert med stjerne (*) er ment for et høyere nivå enn det som kreves i NIO-finalen som BOI og IOI. Dette heftet kan inneholde feil,

i så fall kan du gi meg beskjed på fredrik.anfinsen@gmail.com så skal jeg rette det opp så fort jeg kan.

1.1 Om NIO-oppgavene

Gjennom heftet er det flere NIO-oppgaver fra tidligere finaler og kvalifikasjonsrunder. Oppgavene går ut på å skrive et dataprogram som skal løse et gitt problem. Her er et eksempel på en slik oppgave,

Aftenposten kommer hver dag ut med en avis som inneholder en Sudoku-oppgave. De er bekymret for at et av deres Sudokubrett ikke har en løsning. Du får oppgitt hvordan Sudokubrettet ser ut og skal finne hvor mange løsninger den har og finne én av løsningene. Aftenposten er ute etter antall løsninger siden de vil ha en pekepinn på hvor lett brettet er.

Input

Input består av et rutenett på 9×9 felt, som skal representere Sudokubrettet. Brettet består av tallene $0 \dots 9$, der 0 betyr et tomt felt.

Output

Output består av 10 linjer. Linje nr. 1 skal det stå ett heltall som er antall løsninger brettet har (dette kan være 0). Hvis brettet har en løsning skal du bruke de neste 9 linjene på å skrive ut ett av løsningene til Sudokubrettet.

Eksempelinput

Input

```
5 3 0 0 7 0 0 0 0
6 0 0 1 9 5 0 0 0
0 9 8 0 0 0 0 6 0
8 0 0 0 6 0 0 0 3
4 0 0 8 0 3 0 0 1
7 0 0 0 2 0 0 0 6
0 6 0 0 0 0 2 8 0
0 0 0 4 1 9 0 0 5
0 0 0 0 8 0 0 7 9
```

Output

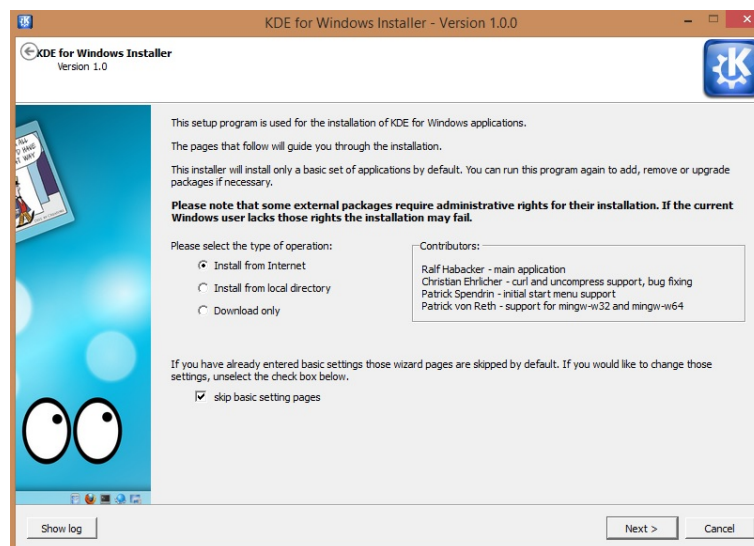
```
1
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9
```

Så kan man starte med å lage et program som leser inn input. Så må man programmere en metode for å løse problemet, etter å ha tenkt en stund på hvordan dette skal gjøres. En metode for å løse dette problemet er å lage en rekursiv algoritme (vi skal se på dette senere). Så kan du kjøre koden på input-eksempelet og se om du fikk det samme svaret som i output-eksempelet. Hvis

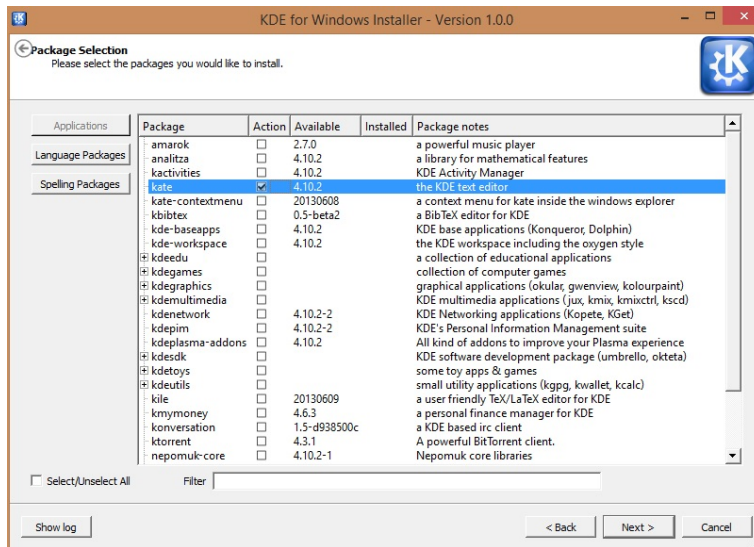
dette stemmer kan du laste opp løsningen på serveren og du vil etter noen få sekunder få en tilbakemelding på hvordan løsningen din gjorde det mot de andre testdataene som ligger på serveren. Denne tilbakemeldingen er i form av en poengscore som blir regnet ut ifra hvor mange forskjellige sudokubrett algoritmen din klarte å løse. For flere detaljer om hvordan dette skjer, se <http://nio.no/bootcamp/?p=236>.

1.2 Installasjon av C++ editor og kompilator

I NIO-finalen kan du velge å programmere i C++, C eller Pascal. Siden C++ er, etter min mening, best tilpasset algoritmeprogrammering, er det det vi skal bruke i dette heftet. Se neste seksjon “Forkunnskaper” for en pensumliste som er nødvendig for å forstå kodeeksemplene i dette heftet. På denne siden <http://nio.no/bootcamp/?p=41> er det beskrevet hvordan du enkelt kan sette opp et C++ environment for å komme i gang. Jeg vil her legge ved min favoritt oppstilling, noe som kan gjøre det enda enklere for deg å programmere til NIO. Installasjonen som jeg viser her krever Windows, men hvis du har Linux er det likedan. Først så må du laste ned tekst-editoren Kate (<http://windows.kde.org/>) og kompilatoren MinGW (<http://www.mingw.org/>, Download Installer til høyre). Installer så Kate gjennom å kjøre KDEWin.

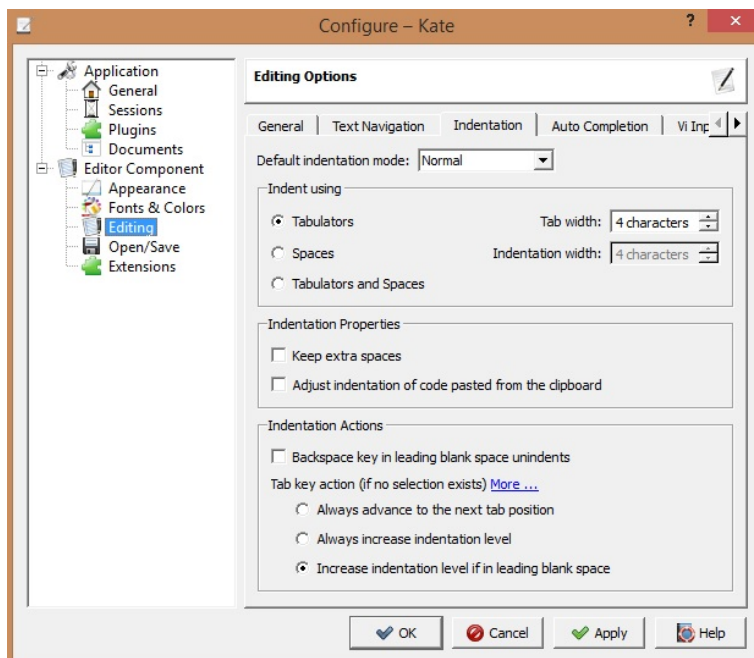


Figur 1: Velg Install From Internet

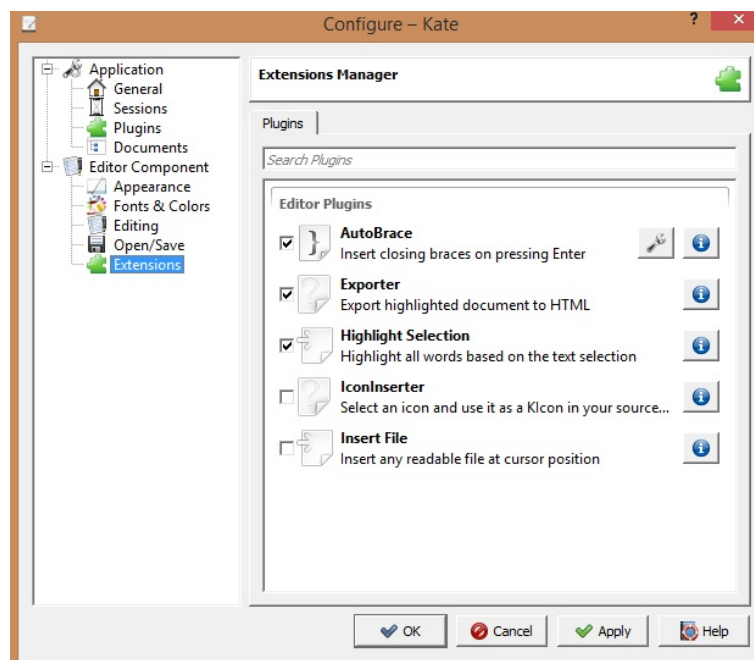


Figur 2: Kryss av for å få Kate

Når du er ferdig med å installere Kate kan du åpne Kate og gå til **Settings -> Configure - Kate**. Deretter stille inn som vist nedenfor:

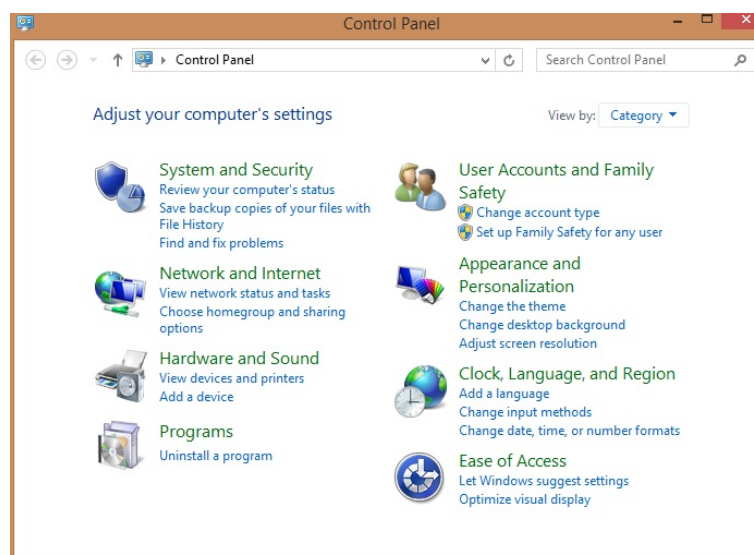


Figur 3: Tabulatorer

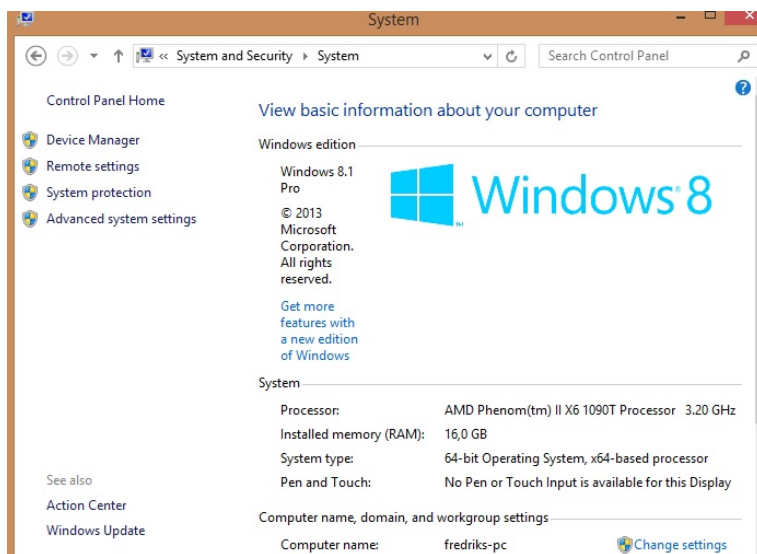


Figur 4: AutoBrace

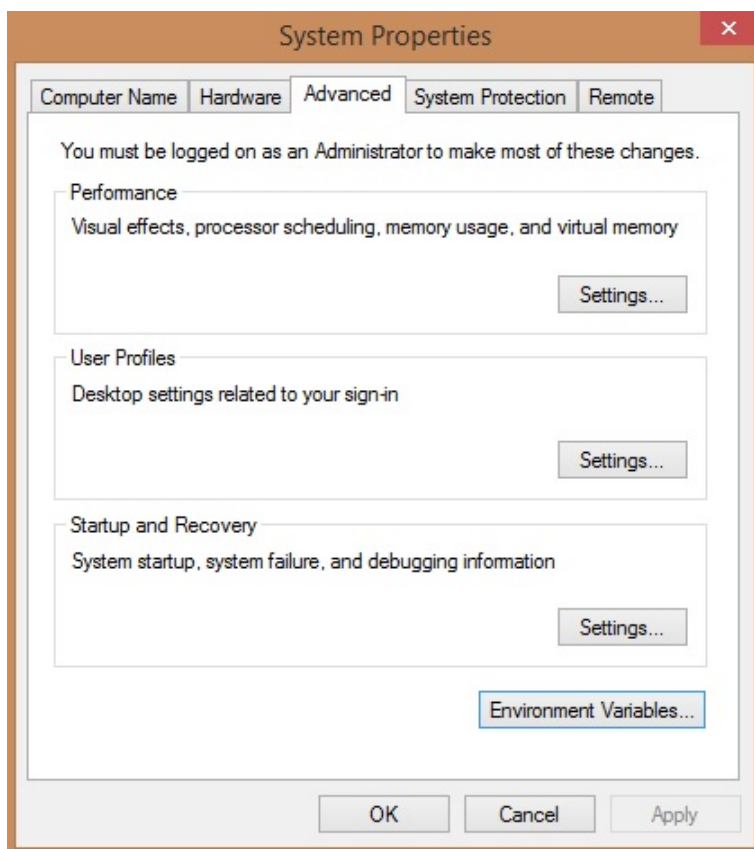
Etter du er ferdig med å installere Kate installerer du MinGW. Husk bare hvor på datamaskinen du installerer MinGW! Etter du har installert den må du nemlig taste inn installasjons-plasseringen i et register i Windows slik at andre programmer kan finne MinGW. Gå til Kontrollpanel,



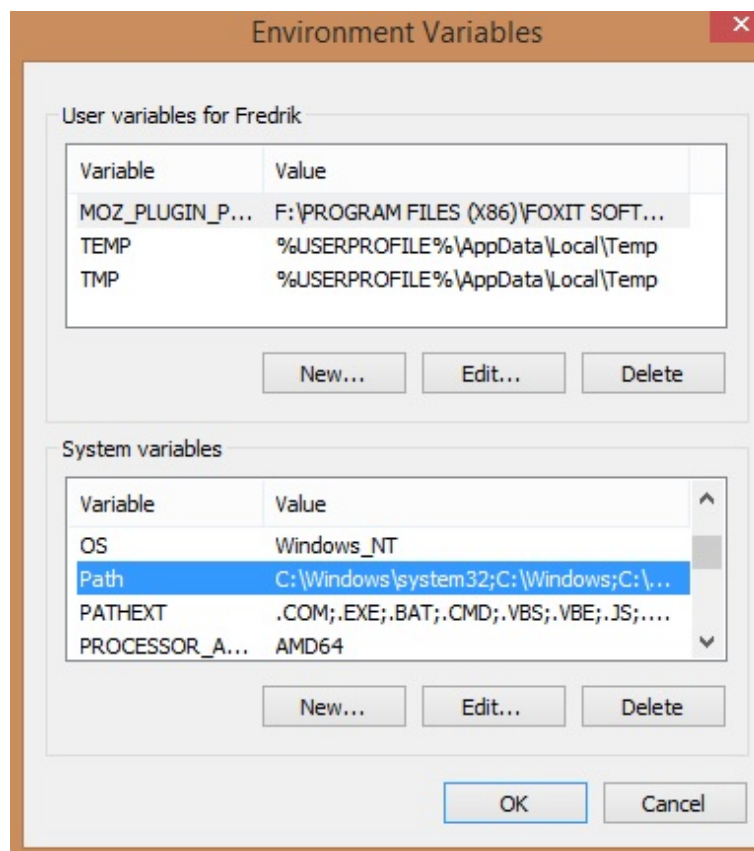
Klikk deg til System,



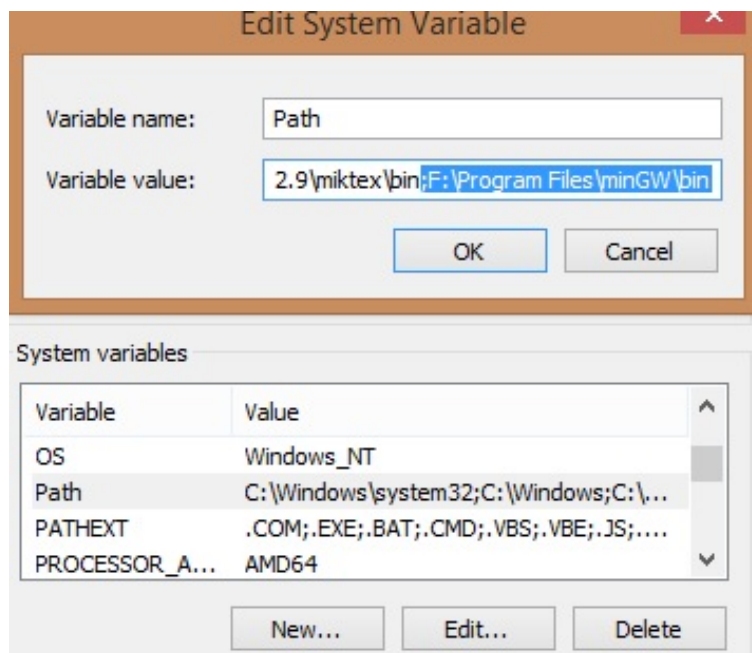
Trykk på Advanced System Settings i vinduet til venstre



Klikk så på Environment Variables...

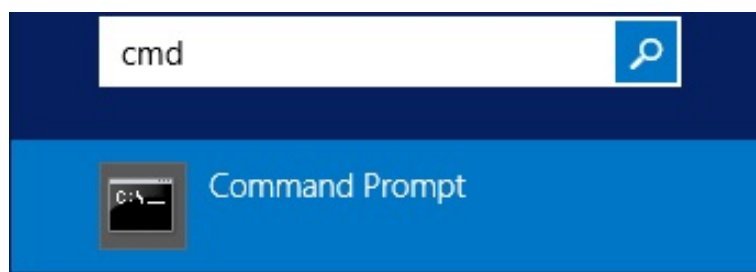


Bla deg ned til Path under System variables. Trykk så Edit... for å få opp dette vinduet

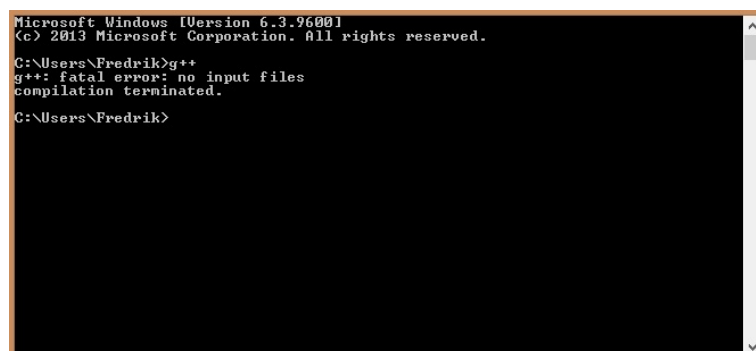


Skriv inn tilsvarende som jeg har markert, bare bytt ut til din egen plassering. Husk å separere det forrige som sto der med et semikolon (;) som er det første tegnet som er markert ovenfor. Etter du har gjort dette restarter datamaskinen din.

Åpne så programmet cmd som ser slikt ut,



Skriv inn g++ og trykk Enter og da skal du se dette:



(hvis det står at den ikke finner g++ så se ovenfor om du har glemt noe på veien).

Du kan navigere opp i mapper med å skrive “cd..”, du kan gå inn i mapper med å skrive “cd [Navn på mappe]”, og du kan få opp en liste over alle filer og mapper i den mappen du er i med å skrive “dir”. Åpne Kate og skriv inn denne koden,

```
#include <iostream>

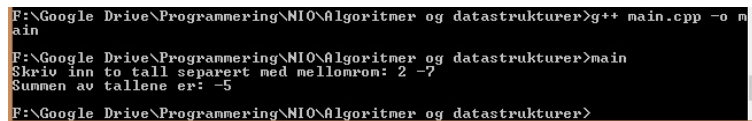
using namespace std;

int main () {
    int a, b;
    cout << "Skriv inn to tall separert med mellomrom: ";
    cin >> a >> b;

    cout << "Summen av tallene er: " << a + b << endl;
}
```

Lagre filen som **main.cpp** et sted på datamaskinen som du husker. Gå tilbake til cmd og naviger deg ved hjelp av kommandoene “cd [navn på mappe]” og “cd ..” til mappen du nettopp lagret filen i. Hvis du f.eks. trenger å bytte harddisk (fra C: til F:) kan du skrive “F:”.

Når du er framme kan du skrive inn “g++ main.cpp -o main”. Kommandoen gjør om koden i filen main.cpp til en kjørbart kode og lagrer den som “main”. Hvis alt er gjort riktig skal du ikke få noen feilmeldinger. Du kan nå kjøre programmet med å skrive “main”. Du kan selvfølgelig velge et annet filnavn enn main.cpp.



```
F:\Google Drive\Programmering\NIO\Algoritmer og datastrukturer>g++ main.cpp -o main
F:\Google Drive\Programmering\NIO\Algoritmer og datastrukturer>main
Skriv inn to tall separert med mellomrom: 2 -7
Summen av tallene er: -5
F:\Google Drive\Programmering\NIO\Algoritmer og datastrukturer>
```

Etterhvert som du senere skal teste programmet med store input kan det være hurt å skrive inputten i en egen fil. Åpne Kate og lag en ny fil. Skriv inn “4 12”. Lagre filen som input.txt i samme mappe som main.cpp. Du kan også endre koden i main.cpp til å bli (men ikke nødvendig),

```
#include <iostream>

using namespace std;

int main () {
    int a, b;
    cin >> a >> b;

    cout << a + b << endl;
}
```

Siden vi har endret på koden må vi igjen kompilere den med å skrive “g++ main.cpp -o main” (som et lite triks kan du bruke opp/ned-pilene på tastaturet for å bla igjennom gamle kommandoer, for å slippe å skrive dem om igjen). Skriv så inn “main < input.txt”. Da vil du mest sannsynlig se noe slikt,

```
F:\Google Drive\Programmering\NIO\Algoritmer og datastrukturer>g++ main.cpp -o m
ain
F:\Google Drive\Programmering\NIO\Algoritmer og datastrukturer>main < input.txt
16
F:\Google Drive\Programmering\NIO\Algoritmer og datastrukturer>
```

Og du er klar for å programmere!

Som et annet lite triks kan du skrive “main < input.txt > output.txt”. Da kjøres programmet main med inputten som står i input.txt (som tidligere), men istedet for å skrive ut resultatet i cmd-vinduet skriver den ut til filen output.txt. Skulle noen gang programmet ditt krasje og henge seg opp i cmd-vinduet kan du trykke **Ctrl+C** som stopper programmet ditt (det hender en del...).

1.3 Forkunnskaper

Før du leser videre burde du lære deg det grunnleggende i C++ programmering. Du burde først lese på denne nettsiden <http://www.cplusplus.com/doc/tutorial/>. Du kan lese til og med **Arrays** også burde du lese på **Data structures** og **Classes (I)**. Det er viktig at du også kan programmere med **String**, se <http://www.cprogramming.com/tutorial/string.html>. Når du er ferdig med det kan du begynne å lære deg STL, <http://community.topcoder.com/tc?module=Static&d1=features&d2=082803> som en kort innføring (du kan selvfølgelig bruke andre nettsider). I tillegg burde du lære deg å bruke datastrukturene **queue**, **stack**, **priority_queue**. Du finner oversikt over hva STL har å tilby på denne nettsiden: <http://www.cplusplus.com/reference/stl/> (den er sterkt anbefalt å bruke). Selv om du mest sannsynligvis ikke har bruk for dem i starten trenger du dem senere. Du velger helt selv om du vil lære deg dem nå eller senere. Etterhvert som du føler at du behersker C++ og STL kan du prøve å bruke **printf** og **scanf** istedet for **cout** og **cin** siden disse er raskere og har flere muligheter. I dette heftet bruker vi kun **cout** og **cin**.

1.4 Oppgaver i heftet

I hvert avsnitt i heftet er det tilhørende oppgaver. Hvis ikke noe annet ved oppgavene er spesifisert er oppgaven hentet fra **UVa Online Judge** (<http://uva.onlinejudge.org/>) der oppgaven er referert ved navn og oppgave-ID som f.eks. **11172 - Relational Operator** du kan så søke etter problemet på **UVa**s nettsider eller direkte på Google. Der står oppgaveteksten, og for å levere svar på oppgaven må du ha en registrert bruker. Du laster opp problemet enten direkte på oppgavesiden eller ved å gå til denne linken http://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=25. Du kan så se hvordan innleveringen din gjorde det med å trykke på **My Submissions**.

I oppgave-tabellene er oppgavene med stjerne (*) anbefalte oppgaver, siden oppgaven er ekstra relevant eller er av høyere kvalitet enn de andre oppgavene.

1.5 Oppgaver

Komme igang oppgaver:

ID	Navn
11172	Relational Operator (*)
11498	Division of Nlogonia (*)
11727	Cost Cutting (*)
11559	Event Planning (*)
11799	Horror Dash (*)
573	The Snail (*)
11507	Bender B. Rodríguez Problem
272	TEX Quotes
1124	Celebrity jeopardy
10550	Combination Lock
11044	Searching for Nessy
11364	Parking
11547	Automatic Answer
12250	Language Detection
12289	One-Two-Three
12372	Packing for Holiday
12403	Save Setu
12577	Hajj-e-Akbar
621	Secret Research
10300	Ecological Premium
11332	Summing Digits
11679	Sub-prime
11764	Jumping Mario
11942	Lumberjack Sequencing
12015	Google is Feeling Lucky
12157	Tariff Plan
12468	Zapping
12503	Robot Instructions
12554	A Special "Happy Birthday" Song!!!
119	Greedy Gift Givers
661	Blowing Fuses
10324	Zeros and Ones
10424	Love Calculator
10919	Prerequisites?
11586	Train Tracks
11661	Burger Time?
11683	Laser Sculpture
11687	Digits
673	Parentheses Balance (*)

Oppgaver om tid

ID	Navn
579	Clock Hands (*)
10683	The decadary watch

Spilloppgaver:

ID	Navn
10646	What is the Card? (*)
489	Hangman Judge (*)
10189	Minesweeper (*)
340	Master-Mind Hints
10530	Guessing Game
10363	Tic Tac Toe

Oppgaver om palindromtall:

ID	Navn
401	Palindromes (*)
10945	Mother bear (*)
11221	Magic square palindromes. (*)
353	Pesky Palindromes
10018	Reverse and Add

Oppgaver om anagrammer:

ID	Navn
156	Ananagrams (*)
195	Anagram (*)
642	Word Amalgamation
10098	Generating Fast

Oppgaver fra virkeligheten:

ID	Navn
161	Traffic Lights (*)
10812	Beat the Spread! (*)
10082	WERTYU
11223	dah dah dah!

2 Binærsøk

Binærsøk er en av de viktigste teknikkene innenfor algoritmisk programmering. Du har kanskje gjort et binærsøk før uten å vite om det. Si du skal slå opp ett navn i telefonkatalogen. Anta at du slår opp navnet Stoltenberg. Du vil neppe begynne å lese telefonkatalogen side etter side fra starten. Du vil mest sannsynlig slå opp på midten av telefonkatalogen og sjekke om det er før navnet Stoltenberg eller etter, og fortsette med å slå opp i midten helt til du finner Stoltenberg. (Hvis du ikke allerede bruker denne teknikken anbefaler jeg sterkt at du bruker den!).

Et annet og mer klassisk eksempel er et spill der du ber en venn om å tenke på et tall mellom 1 og 100. Du skal prøve å gjette deg fram til tallet vennen tenker på med å gjette noen tall og få tilbakemelding på om det tallet du gjettet på var for stort, for lite eller riktig tall.

Anta at vennen din tenker på tallet 18. Du tipper på tallet 50, og får tilbakemelding på at det var alt for stort. Du vil nå kanskje tippe tallet 25, og får fortsatt beskjed om at tallet er for stort. Så tipper du kanskje tallet 12 og får nå beskjed om at det er for lite. Neste tall du tipper på ligger midt mellom 12 og 25 og du spør da om tallet er 18. Vennen må dessverre si seg enig i det. Faktisk trenger du kun 7 spørringer med denne teknikken til å finne hvilket som helst tall mellom 1 og 100. Hvis du utvider spillet der man kan tenke på et tall mellom 1 og 1,000,000,000 trenger du 30 spørringer for å finne fram til tallet. Det er svært lite!

Så hvordan finner vi ut hvor mange spørringer vi trenger for å finne hvilket som helst tall? Tenk deg situasjonen fra i stad der vennen din tenker på ett tall fra og med 1 til og med 100. Før første gjetting er alle de 100 tallene mulige. Hvis du tipper på 50 kan du være så heldig at det var riktig, men i verste fall vil du klare å halvere antall mulige tall. Tipper du igjen på midten vil du igjen halvere antall mulige tall. Med denne teknikken vil du nemlig etter n gjetninger ha $\frac{100}{2^n}$ mulige tall igjen. Vi er ferdig når antall mulige tall som er igjen er ≤ 1 . Det er nemlig når $n = 7$. Vi får at $\frac{100}{2^n} \leq 1$ når $2^n \geq 100$. Siden $2^6 = 64 < 100$ og $2^7 = 128 \geq 100$ er 7 det minste antall gjetninger vi trenger. Hvis du har hatt om logaritmer på videregående vet du kanskje at vi kan finne ut den minste n slikt at $2^n \geq a$. Bruker vi logaritme med grunntall 2 (\log_2) er n gitt ved $n = \lceil \log_2(a) \rceil$ der $\lceil x \rceil$ betyr å runde opp x til nærmeste hele tall. Har du ikke en \log_2 knapp på kalkulatoren kan du regne ut $\log_2(a)$ ved å regne $\log(a)/\log(2)$ uansett hva slags logaritme du bruker (10, e). Vi kan da se at med tallene fra og med 1 til og med 1,000,000,000 trenger vi 30 spørringer. Siden $\log_2(1,000,000,000) \approx 29.89735$, vi runder opp og får 30.

Så vi kan programmere dette spørrespillet i C++, der datamaskinen skal spille din rolle med å gjette seg fram. Vi må hele tiden holde styr på hva det minste og største mulige tallet kan være. La oss kalle disse tallene for a og b . I starten vil $a = 1$ og $b = 100$. Hvis vennen tenker på tallet 18 får vi etter å ha gjettet på tallet 50 $a = 1$ og $b = 49$ (siden vi vet det er under 50).

```
cout << "Tenk paa et tall mellom 1 og 100, og trykk enter" <<
endl;

int a = 1, b = 100;

while (a < b) {
    int midten = (a + b) / 2;
```

```

        cout << "Er tallet MINDRE, STOERRE eller LIK " << midten
            << endl;
        string str;
        cin >> str;

        if (str == "LIK") {
            a = b = midten; //er kun 1 mulighet igjen
        } else if (str == "MINDRE") {
            b = midten - 1;
        } else {
            a = midten + 1;
        }
    }
    cout << "Tallet du tenkte paa var: " << a << endl;

```

Vi kan også gjøre binærsøk på andre ting. For eksempel hvis vi har gitt en liste med tall, så skal vi finne det minste tallet som er større eller lik 100. Uten binærsøk måtte vi mest sannsynlig søke gjennom hele listen med tall helt til vi fant et tall som var større eller lik 100, og så stoppet (hvis listen er sortert).

```

int v[10] = {19, 45, 56, 98, 103, 107, 476, 765, 1009, 2014};
int a = 0, b = 9; //min og maks indeksnummer

while (a < b) {
    int mid = (a + b) / 2;

    if (v[mid] < 100)
        a = mid + 1;
    else if (v[mid] > 100) {
        b = mid;
    } else {
        a = b = mid;
    }
}

cout << "Tallet er: " << v[a] << endl;

```

Prøv å forstå hvorfor det er viktig at listen `v` er sortert, prøv f.eks. å bytte om plasseringene til 98 og 103 i `v`.

Vi kan bytte litt om på koden sånn at i stedet for å finne det minste tallet som er større eller lik 100 kan vi finne det største tallet som er mindre enn 1000.

```

int v[10] = {19, 45, 56, 98, 103, 107, 476, 765, 1009, 2014};
int a = 0, b = 9; //min og maks indeksnummer

while (a < b) {
    int mid = (a + b) / 2;

    if (v[mid] < 1000)
        a = mid;
    else if (v[mid] >= 1000)
        b = mid;
}

cout << "Tallet er: " << v[a] << endl;

```

2.1 Binærsøk i STL

I C++ har vi allerede noen innebygde funksjoner som gjør mye av dette for oss (men LURT å kunne implementere selv for vanskelige oppgaver som krever

binærsøk med spesielle kriterier).

2.1.1 lower_bound

I C++ har vi en innebygd funksjon `lower_bound`, for dokumentasjon se http://www.cplusplus.com/reference/algorithm/lower_bound/. Denne funksjonen tar inn en sortert liste (**vector**) og prøver å finne det minste elementet som er større eller lik ett gitt tall ved hjelp av binærsøk.

Si vi er ute etter å finne det samme som i stad (minste tallet som er større eller lik 100). Vi kan da skrive,

```
int v[10] = {19, 45, 56, 98, 103, 107, 476, 765, 1009, 2014};
vector<int> tab(v, v + 10);

vector<int>::iterator it = lower_bound(tab.begin(), tab.end(),
                                     100);
//it er naa en peker til det tallet vi er ute etter
int idx = (it - tab.begin());
//idx er naa indeksnummeret til tallet
cout << "Tallet er: " << tab[idx] << endl;
```

Hvis du er usikker på noe av koden, se nettsiden ovenfor for flere eksempler og forklaring.

2.1.2 upper_bound

I C++ har vi en tilsvarende funksjon `upper_bound` (http://www.cplusplus.com/reference/algorithm/upper_bound/). Denne funksjonen tar inn en sortert liste (**vector**) og prøver å finne største elementet som er strengt mindre enn et gitt tall ved hjelp av binærsøk.

I motsetning til `lower_bound` gir ikke `upper_bound` helt det vi ville ha i forrige eksempel (største tall som er mindre eller lik 1000). Den gir oss heller første tall som er større enn 1000. Vi kan da skrive,

```
int v[10] = {19, 45, 56, 98, 103, 107, 476, 765, 1009, 2014};
vector<int> tab(v, v + 10);

vector<int>::iterator it = upper_bound(tab.begin(), tab.end(),
                                     1000);
int idx = (it - tab.begin());
cout << "Tallet er: " << tab[idx] << endl;
```

Var vi ute etter å finne største tallet som er mindre eller lik 1000 (hvis det finnes) vet vi at den befinner seg på plass `idx - 1`.

2.1.3 binary_search

Binærsøk trenger ikke bare å brukes til å finne største eller minste verdier, den kan også brukes til å finne ut om et spesielt element eksisterer blant en mengde. Hvis du har en telefonkatalog og skal finne ut om navnet Rosén finnes i katalogen ville du heller ikke begynne med å bla fra side til side fra starten. Du ville gjort et slags binærsøk. C++ har en funksjon for dette. Den heter `binary_search` (http://www.cplusplus.com/reference/algorithm/binary_search/). Den krever som `lower_bound` og `upper_bound` en sortert liste. Se koden nedenfor,

```
int v[10] = {19, 45, 56, 98, 103, 107, 476, 765, 1009, 2014};
vector<int> tab(v, v + 10);
```



```

if (binary_search(tab.begin(), tab.end(), 476))
    cout << "Tallet 476 finnes i listen" << endl;
else
    cout << "Tallet 476 finnes ikke i listen" << endl;

if (binary_search(tab.begin(), tab.end(), 524))
    cout << "Tallet 524 finnes i listen" << endl;
else
    cout << "Tallet 524 finnes ikke i listen" << endl;

```

2.2 NIO oppgaver med binærsøk

Byggmester, 2. runde i NIO 2013/2014.

Bob er byggmester og planlegger å bygge et nytt hus som han kan selge. Det skal bygges langs riksvei 512 et sted. For å få solgt huset for mest mulig ønsker Bob å velge en best mulig tomt for huset. Det er selvfølgelig mange ting som er viktig da. En av faktorene som er viktig for tomten er hvor langt det er til nærmeste nabo. Bob har skaffet informasjon om hvor alle husene langs riksvei 512 befinner seg, og han har plukket ut en del aktuelle tomter. Nå trenger han din hjelp til å finne ut hvor langt det er til nærmeste hus fra de aktuelle tomtene. Alle husene ligger inntil riksveien, og avstanden fra et hus til et annet er avstanden du må kjøre på riksveien for å nå det andre huset. Posisjonen til et hus oppgis som hvor mange meter fra starten av riksveien huset ligger. F.eks. kan et hus ligge 1024 meter fra starten av veien, mens et annet ligger 1324 meter fra start. Avstanden mellom dem er da 300 meter.

Input

Første linje inneholder heltallene N og T , antall hus langs veien og antall aktuelle tomter Bob har plukket ut. Så følger N linjer, hver med et heltall p_i - posisjonen til et hus. Og til slutt følger T linjer, hver med et heltall a_j - posisjonen til et aktuelt hus.

$$1 \leq N \leq 200000$$

$$1 \leq T \leq 10000$$

$$1 \leq p_i \leq 1000000000$$

$$1 \leq a_j \leq 1000000000$$

Output

For hver aktuelle tomt, a_j , skriv ut ett heltall på én linje: Hvor mange meter det er til nærmeste hus. Skriv ut i samme rekkefølge som de kommer i input.

En mulig måte å løse problemet på er at vi for hver tomt går igjennom alle husene for å se hvilket hus som er nærmest tomten. Dessverre blir det $T \cdot N$ sammenligninger, og i verste fall blir det hele $10000 \cdot 200000 = 2,000,000,000$ sammenligninger. Noe som er alt for stort for at en datamaskin skal kunne klare dette innen tidsfristen på ett sekund! I kapitlet **Kompleksitet** kommer vi mer innpå hvordan denne typen utregningen hjelper oss med å bestemme hvilken algoritme vi skal bruke for å løse et problem. En ting er sikkert, vi kan

ikke sjekke alle husene for hver tomt hvis vi skal lage en full løsning på dette problemet. Det er her binærsøket kommer inn! Hvis vi lagrer posisjonene til husene i en sortert liste, og for hver tomt kjører vi `lower_bound` med tomtens posisjon får vi tilbake det første huset som ligger til høyre for tomten (hvis vi tenker på posisjonene som stigende fra venstre til høyre). Vi vet at det huset som ligger nærmest tomten må enten være det første huset til venstre for tomten, eller første huset til høyre for tomten. Når vi kjører `lower_bound` får vi indeksen til det sistnevnte huset (hvis det eksisterer). Men vi vet også at det huset som har ett lavere indeksnummer må være det første huset til venstre for tomten hvis det eksisterer et slikt hus. Vi sjekker begge disse potensielle husene og tar minimum av avstandene. Løsningen i C++,

```
int N, T;
cin >> N >> T;

vector<int> p(N);
for (int i = 0; i < N; i++)
    cin >> p[i];
sort(p.begin(), p.end()); //for at binaersoek skal fungere

int a;
while (T--) {
    cin >> a;
    vector<int>::iterator it = lower_bound(p.begin(), p.end(), a
    );
    int idx = it - p.begin(); //faa indeksnummeret

    if (it == p.end()) //fantas ikke et hus til hoeyre
        cout << a - p[idx-1] << endl; //ser paa huset til
        venstre
    else {
        if (idx == 0) //finnes ingen hus til venstre
            cout << p[idx] - a << endl;
        else //finnes baade hus til hoeyre og venstre
            cout << min(p[idx] - a, a - p[idx-1]) << endl;
    }
}
```

Prøv med inputten,

```
3 2
170
40
195
100
270
```

og se om du får dette til svar,

```
60
75
```

2.3 Oppgaver

Standard binærsøk:

ID	Navn
11057	Exact Sum (*)
10567	Helping Fill Bates (*)
12192	Grapevine (*)
957	Popes
10077	The Stern-Brocot Number System
10474	Where is the Marble?
10706	Number Sequence
11876	$N + \text{NOD}(N)$

Binærsøk svaret:

ID	Navn
11413	Fill the Containers (*)
12032	The Monkey and the Oiled Bamboo (*)
10341	Solve It (*)

3 Kompleksitet

I denne seksjonen skal vi se på hvordan vi kan finne ut hvor effektiv en algoritme er på generelle testdata og avgjøre om algoritmen er rask nok for å løse oppgaven i tide. Det er svært nyttig å vite hvor effektiv en algoritme er før man skal begynne å kode den eller finne ut om man trenger en bedre metode for å løse oppgaven.

3.1 Big-Oh notasjon

Big-Oh notasjon \mathcal{O} er en matematisk måte å beskrive effektiviteten til en algoritme.

La oss ta denne koden,

```
int N;
cin >> N;
for (int i = 0; i < N; i++) {
    cout << i << endl;
}
```

Denne koden leser inn et heltall, for så skrive ut N tall $(0, 1, 2, \dots, N-1)$. Vi sier at koden har kompleksitet $\mathcal{O}(N)$. Dette sier oss at programmet har en kjøretid lineært lik N . Dobler vi N blir også kjøretiden doblet. Hadde vi istedet skrevet ut to tall i for-løkken ville fortsatt kompleksiteten bli $\mathcal{O}(N)$ siden dobler vi N blir antall tall som blir skrevet ut doblet.

Vi ser på en ny kode,

```

int N;
cin >> N;
int sum = 0;
for (int i = 1; i <= 100; i++) {
    sum += N*i;
}

```

Dette programmet leser inn heltallet N . Deretter summerer den $N + 2N + 3N + \dots + 100N$ inn i **sum**. Hvis vi ser på kompleksiteten til programmet vil kjøretiden være konstant og uavhengig av verdien til N . Uansett hvor stor N er vil programmet kun foreta 100 multiplikasjoner og addisjoner. Siden kjøretiden er uavhengig av input sier vi at programmet har en konstant kjøretid, som med big-Oh notasjon skrives som $\mathcal{O}(1)$.

Vi ser på et nytt eksempel,

```

int N;
cin >> N;
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {
        cout << i * j << ", ";
    }
    cout << endl;
}

```

Dette programmet skriver opp gangetabellen. Programmet regner ut alle gange-stykkene $1 * N, 2 * N, \dots, (N - 1) * N, N * N$. Vi ser at programmet har en kompleksitet lik $\mathcal{O}(N^2)$. Siden kompleksiteten er N^2 vil det si at hvis vi erstatter N med et dobbelt så stort tall $2N$. Vil gjøretiden øke med $\frac{(2N)^2}{N^2} = 4$. Vi får altså fire ganger så lang kjøretid ved doubling av N .

Vi tar noen raske eksempler,

```

int N;
cin >> N;
int sum = 0;
for (int i = 1; i <= N; i++)
    for (int j = 1; j <= N; j++)
        for (int k = 1; k <= N-10; k++)
            sum += (i + j + k);

```

Har kompleksitet $\mathcal{O}(N^3)$. Legg merke til at $k \leq N - 10$, men kjøretiden blir fortsatt $\mathcal{O}(N^3)$. Siden for store N har $k \leq N - 10$ liten betydning.

```

int N;
cin >> N;
int sum = 0;
for (int i = 1; i <= N+1000; i++) {
    cout << i << endl;
}

```

Har kompleksitet lik $\mathcal{O}(N)$. Konstantleddet 1000 forsvinner, og er ikke avhengig av N . Matematisk vil \mathcal{O} være asymptotisk lik kjøretiden. Vi tenker ikke på konstantledd.

```

int N, M;
cin >> N >> M;

```

```

for (int i = 0; i < N * N; i++) {
    for (int j = 0; j < M; j++) {
        cout << i + j << endl;
    }
}

```

Kompleksitet lik $\mathcal{O}(N^2M)$.

Kompleksitet på algoritmer til noen tidligere NIO-oppgaver

- **Brodering (2. runde, 2013/2014)** Siden man trenger kun å fylle ut en todimensjonal array med størrelse $R \times K$ får vi kompleksiteten $\mathcal{O}(RK)$.
- **NIO-ball (finale, 2012/2013)** For å løse oppgaven trenger man å regne ut

$$\binom{N-1}{4}$$

som gjøres på konstant tid (uavhengig av N). Vi får kompleksiteten $\mathcal{O}(1)$.

- **Byggmester (2. runde, 2013/2014)** Siden vi for hver tomt foretar ett binærsøk blir kompleksiteten lik $\mathcal{O}(T \log(N))$.

4 Matematikk

4.1 Binære tall

I en datamaskin representeres tall i totallsystemet (binære tall) med kun to sifre 0 og 1. Vi skal se hvordan dette gjøres og hva vi kan bruke dem til. Først viser vi et eksempel fra titallsystemet. Vi kan skrive tallet 86029 i en tabell,

$$\begin{array}{c|c|c|c|c} 10^4 & 10^3 & 10^2 & 10^1 & 10^0 \\ \hline 8 & 6 & 0 & 2 & 9 \end{array}$$

det tabellen sier er at tallet 86029 kan skrives som

$$8 \cdot 10^4 + 6 \cdot 10^3 + 0 \cdot 10^2 + 2 \cdot 10^1 + 9 \cdot 10^0 = 80000 + 6000 + 20 + 9 = 86029$$

Vi skal se på en tilsvarende tabell i totallsystemet,

$$\begin{array}{c|c|c|c|c|c|c|c} 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ \hline 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{array}$$

vi kan regne ut hvilket tall denne tabellen skal forestille på tilsvarende måte som i titallsystemet

$$1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 64 + 32 + 8 + 1 = 105$$

Vi ser at vi trenger hele 7 sifre for å representere tallet 105 i totallsystemet (3 i titallsystemet). Grunnen til dette er at i totallsystemet har vi kun to mulige

sifre. Hvis du ser i i tabellen har vi satt av plass til åtte sifre, grunnen til dette er at i en datamaskin blir alt lagret i blokker på 8 bits (0 og 1'ere) en slik pakke kalles 1 byte. En integer (**int** i C++) bruker 4 slike blokker (4 bytes). Det betyr at en **int** har hele $4 \cdot 8 = 32$ bits til rådighet. Siden vi da kan representere en **int** ved hjelp av en tabell med 32 kolonner, og for hver kolonne har vi to valg enten 0 eller 1, har vi hele 2^{32} mulige verdier for en **int**. Siden en **int** kan lagre både positive og negative verdier er ett av bittene satt av for å bestemme fortegnet til tallet (pluss / minus). Vi står da kun igjen med 31 bits, med 2^{31} kombinasjoner. Siden 0 blir regnet som et positivt tall vil det største tallet en **int** kan lagre være $2^{31} - 1 = 2,147,483,647$. Dette er naturlig å tenke seg hvis vi tenker på titallsystemet. Hva er det største 4 sifrede tallet vi har i titallsystemet? Nemlig $10^4 - 1 = 9,999$. Vi har jo i tillegg 31 bits til å lagre negative tall, faktisk er det minste tallet én **int** kan lagre lik $-2^{31} = -2,147,483,648$. Her trengte vi ikke å trekke fra -1 siden 0 tilhørte de positive tallene.

Det er noen ting som er et MUST å kunne innenfor matematikk/informatikk, og det er alle toerpotenser fra og med 2^0 til og med 2^{10} . Dette er nesten like viktig som å vite at en Grandiosa skal stekes på 220°C .

Potenser	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
Verdi	1	2	4	8	16	32	64	128	256	512	1024

Vi kan skrive et tall i titallsystemet i totallsystemet veldig lett. La oss gjøre det med tallet 863. Vi ser at $512 \leq 863 < 1024$. Vi ser at vi kan ta med tallet 512 i vår sum for å lage tallet 863. Vi ser at vi da har igjen $863 - 512 = 351$. Vi ser at $256 \leq 351 < 512$. Vi tar med 256 i vår sum, og vi står igjen med $351 - 256 = 95$. Nå har vi at $64 \leq 95 < 128$, så vi tar 64 med i vår sum. Vi står igjen med $95 - 64 = 31$. Vi har at $16 \leq 31 < 32$. Vi tar da med 16 i vår sum. Vi står igjen med $31 - 16 = 15$. Vi ser at $8 \leq 15 < 16$. Vi tar med 8 i vår sum, og står igjen med $15 - 8 = 7$. Fortsetter vi sånn her ser vi at både 4, 2 og 1 må være med i summen (siden $1 + 2 + 4 = 7$). Vi kan da skrive tallet 863 i totallsystemet i tabellen,

2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	0	1	1	1	1	1

I en datamaskin vil da tallet 863 se ut som 1101011111. Denne metoden tilsvarer måten vi bruker penger på her i Norge. Si vi skal gi 57 kroner til noen i form av mynter og vi vil gi færrest mulig mynter fra oss. Vi ser da at $20 \leq 57$. Vi gir da én 20-kroning. Vi står igjen med $57 - 20 = 37$. Vi ser at $20 \leq 37$ og vi legger på én 20-kroning til. Vi står igjen med $37 - 20 = 17$. Nå er $20 \geq 17$ og vi kan ikke gi flere 20-kroninger. Vi ser dermed at $10 \leq 17$ og legger på én 10-kroning. Nå mangler vi $17 - 10 = 7$ kroner. Nå er 10 kroner for mye og vi ser at $5 \leq 7$. Vi gir også bort én femkroning. Vi mangler $7 - 5 = 2$ kroner. Vi ser at eneste muligheten vi har igjen er å gi 2 én-kroninger. Da er vi ferdig.

4.2 Binære operasjoner

Vi kan utføre noen boolske operasjoner på binære tall. Si vi har de binære tallene $a = 1101011011$ og $b = 0101110110$.

4.2.1 AND

Vi kan da bruke operasjonen AND (&) på dem,

	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
a	1	1	0	1	0	1	1	1	1	1
b	0	1	0	1	1	1	0	1	1	0
a & b	0	1	0	1	0	1	0	0	1	0

Vi ser at vi får et nytt tall $c=a&b$ som har alle bittene sine lik 0, bortsett fra de plassene der både a og b har bit lik 1. Vi krever at en bit i c er 1 hvis og bare hvis den tilsvarende bittene er 1 i a og (AND) 1 i b.

&	0	1
0	0	0
1	0	1

4.2.2 OR

Tilsvarende har vi en OR (|) operasjon,

	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
a	1	1	0	1	0	1	1	1	1	1
b	0	1	0	1	1	1	0	1	1	0
a b	1	1	0	1	1	1	1	1	1	1

Som du ser har bittene i $c = a | b$ lik 1 hvis den tilsvarende bittene i a eller (OR) tilsvarende bittene i b er lik 1. Dette gjelder også hvis både bittene i a og bittene i b er begge lik 1. Sagt på en annen måte, én bit i c er kun 0 hvis og bare hvis den tilsvarende bittene i a og b er 0.

	0	1
0	0	1
1	1	1

4.2.3 XOR

Vi har en XOR (^) operasjon,

	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
a	1	1	0	1	0	1	1	1	1	1
b	0	1	0	1	1	1	0	1	1	0
a ^ b	1	0	0	0	1	0	1	1	0	1

Hvis vi har $c = a ^ b$ vil bittene i c være lik 1 hvis og bare hvis den tilsvarende bittene i a og den tilsvarende bittene i b er forskjellige.

\wedge	0	1
0	0	1
1	1	0

4.2.4 NOT

Til slutt har vi en NOT (\sim) operasjon som inverterer bittene i et tall (0 \rightarrow 1 og 1 \rightarrow 0).

	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
a	1	1	0	1	0	1	1	1	1	1
\sim a	0	0	1	0	1	0	0	0	0	0

4.2.5 Bit-manipulasjon

Vi skal se nærmere på hva vi kan bruke disse operasjonene til i praksis senere i kapitlet, men først kan vi se på hvordan man kan bit-manipulere en **int**.

For det første så kan man veldig lett gange en **int** med 2 veldig raskt og dele med 2 veldig raskt. Dette er en av fordelene man har med å jobbe i totallsystemet. Blant annet i titallsystemet har vi fordelene av at det er enkelt å gange/dele med 10 med å flytte alle sifrene ett hakk til venstre eller ett hakk til høyre (flytte komma). Det samme gjelder for tall i totallsystemet, og vi kan flytte sifrene i C++ på denne måten,

```
int n = 27;
n = (n << 1); //ganger med 2
n = (n >> 1); //deler med 2
n = (n >> 2); //deler med 4, flytter 2 plasser
```

Vi starter med $n = 27$, så ganger vi med 2 og ender opp med $n = 54$. Etterpå deler vi på 2 igjen og ender opp med $n = 27$. Etterhvert deler vi med 4 og ender opp med $n = 6$. Grunnen til at vi ender opp med $n = 6$ og ikke $n = 6.75$ er at **n** er en **int** og divisjonen runder ned. Legg også merke til at vi i siste kodelinje skiftet sifrene 2 plasser til høyre, og vi da deler på $2^2 = 4$. Dette er tilsvarende for titallsystemet med at hvis vi flytter kommaet to plasser til venstre deler vi på $10^2 = 100$.

Hvis vi er interessert i å lage en toerpotens kan vi starte med tallet 1 og gange dette tallet med 2 helt til vi når den toerpotensen vi vil ha. For eksempel hvis vi vil ha 2^7 kan vi skrive,

```
int n = 1;
n = (n << 7);

//eller bare
int n = (1 << 7);
```

Vi ender da opp med det binære tallet $n = 10000000$ (128 i titallsystemet). Vi kommer senere inn på hvorfor det er viktig å generere slike toerpotenser.

Hvis vi er ute etter å lage ett binærtall bestående av bare 1'er bits kan vi skrive koden,

```
int n = (1 << 6) - 1;
```


Vi får da det binæretallet $n = 111111$ (6 siffer). Tilsvarende ville vi fått i titallsystemet hvis vi tar $10^3 - 1$ får vi 999.

Nå skal vi se på hvordan vi kan legge til, fjerne eller bytte en bit i en **int**.

Vi jobber med tallet $a = 105$ som i totallsystemet er 1101001, og vi vil at vi skal bytte ut bit nr. 2 (0 indeksert fra høyre) til å bli 1.

```
int a = 105; //1101001
a = a | (1 << 2);
//eller en forkortet versjon
a |= (1 << 2);
```

a	1	1	0	1	0	0	1
$1 \ll 2$	0	0	0	0	1	0	0
$a (1 \ll 2)$	1	1	0	1	1	0	1

Vi ser at **a** har byttet en 0 bit til en 1 bit. Legg merke til at hvis **a** hadde allerede bit nr. 2 lik 1 ville ingenting skjedd.

Vi kan også sette en bit lik 0 med denne koden,

```
int a = 105; //1101001
a = a & ~(1 << 5);
//eller en forkortet versjon
a &= ~(1 << 5);
```

a	1	1	0	1	0	0	1
$\sim(1 \ll 5)$	1	0	1	1	1	1	1
$a \& \sim(1 \ll 5)$	1	0	0	1	0	0	1

Vi ser at **a** har byttet bit nr. 5 til å bli 0 istedet for 1. Legg merke til at det ikke ville skjedd noen endring hvis bit nr. 5 opprinnelig var 0.

Hvis vi heller er interessert i å bytte om en bit i en **int**, altså gjøre om en 1 bit til en 0 bit eller motsatt, kan vi kjøre denne koden,

```
int a = 105; //1101001
a = a ^ (1 << 4);
//eller en forkortet versjon
a ^= (1 << 4);
```

a	1	1	0	1	0	0	1
$1 \ll 4$	0	0	1	0	0	0	0
$a \wedge (1 \ll 4)$	1	1	1	1	0	0	1

Vi ser at bit nr. 4 er byttet om fra å være 0 til å bli 1. Hvis bit nr. 4 opprinnelig hadde vært 1 ville vi endt opp med at bit nr. 4 hadde blitt 0. Så i motsetning til de to andre manipulasjonene over vil det alltid skje en endring med denne manipulasjonen.

Vi kan også teste om en bit er på (1) eller av (0) med denne koden,

```

int a = 105; //1101001
if ((a & (1 << 0)) != 0) { //husk godt med parenteser
    //bit nr. 0 er 1
} else {
    //bit nr. 0 er 0
}

```

Legg merke til at $(1 \ll 0)$ er lik 1, så kunne skrevet 1 i stedet for $1 \ll 0$.

a	1	1	0	1	0	0	1
$1 \ll 0$	0	0	0	0	0	0	1
$a \& (1 \ll 0)$	0	0	0	0	0	0	1

Legg merke til at hvis bit nr. 0 i **a** hadde vært 0, ville vi endt opp med at $a \& (1 \ll 0) = 0000000 = 0$. Vi skal senere se på bruken av disse manipulasjonsteknikkene, men først litt pause fra bits.

4.3 Fakultet

For de av dere som har hatt matematikk R1 er dere nok godt kjent med fakultet. For å oppsummere er $n!$ (uttales n-fakultet) det samme som,

$$n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 2 \cdot 1$$

Skal vi regne ut $4!$ (fire fakultet) er det det samme som $4 \cdot 3 \cdot 2 \cdot 1 = 24$. Fakultet har mange anvendelser innenfor matematikk. For eksempel tenk deg at du har 7 personer som skal stå i en kø langs én linje. Hvis vi kun tenker på rekkefølgen, hvor mange forskjellige køer finnes det? Vi kan tenke oss at det er 7 valg for hvem som skal stå fremst i køen. Etter at denne personen har stelt seg i køen har vi 6 valg for hvem som skal stå nest fremst i køen. Tilsvarende er det 5 muligheter for neste osv. Tilsammen blir dette $7! = 5040$ mulige køer. Vi kan for treningens skyld regne ut fakultet ved hjelp av denne rekursjonen (funksjon som kaller seg selv),

```

int fakultet(int n) {
    if (n <= 1)
        return 1;
    return n*fakultet(n-1);
}

```

Det rekursjonen gjør når vi skal regne ut $4!$ er å gjøre følgende utregning: $4! = 4 \cdot 3! = 4 \cdot 3 \cdot 2! = 4 \cdot 3 \cdot 2 \cdot 1! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$. Vi ser at hvis vi kjører fakultet(0) får vi 1 som svar. Dette er faktisk matematisk riktig, siden $0!$ er definert til å være 1, dette skal vi se nærmere på senere.

Hvis du vil regne ut fakultet uten en rekursjon kan du f.eks. kode,

```

int fakultet = 1;
for (int i = 2; i <= n; i++)
    fakultet *= i;

```

4.4 Binomialkoeffisient

Binomialkoeffisienter er viktige innenfor kombinatorikk. Er det vanskelig å forstå denne delen kan det være lurt å ta en titt i en matematikk R1 bok i sannsynlighetskapitlet. Tenk deg at under en NIO-finale med 20 deltagere skal det

velges 4 stykker som skal delta i IOI (International Olympiad in Informatics). Hvor mange måter kan dette gjøres på? Først har vi 20 muligheter for første person som kommer til IOI, la oss kalle denne personen A. Etter dette har vi 19 muligheter for neste person, la oss kalle denne personen B. Så har vi 18 muligheter for person C, og 17 muligheter for person D. Vi står da igjen med $20 \cdot 19 \cdot 18 \cdot 17 = 116,280$ muligheter. Dette stemmer ikke helt. Fordi med utregningen vi nå har gjort antar vi at rekkefølgen har noe å si, men om de kvalifiserte blir plukket ut i denne rekkefølgen ADCB eller BDAC har ingenting å si. Så vi kan si at ADCB og BDAC er "like". Siden antall måter vi kan stokke om bokstavene i ABCD er $4!$ må vi dele svaret vårt på $4!$. Vi har da at antall muligheter for de 4 personene som skal til IOI er,

$$\frac{20 \cdot 19 \cdot 18 \cdot 17}{4 \cdot 3 \cdot 2 \cdot 1} = 4845$$

Dette kan mer generelt skrives som (hvis vi har n deltagere og r personer som skal til IOI),

$$\frac{n!}{(n-r)! \cdot r!}$$

denne matematiske formelen forkortes gjerne med nCr . Prøv å sette $n = 20$ og $r = 4$ og da ser du at vi ender opp med det forrige svaret på 4845. Legg også merke til at hvis vi setter $r = n$ (alle deltagerne skal til IOI) får vi $0!$ i nevneren som vi husker er definert til å være lik 1. Svaret blir også 1 fordi det finnes bare 1 mulighet, nemlig at alle kommer til IOI.

Det kan være lurt, hvis du ikke har regnet med binomialkoeffisienter før, at du regner noen slike oppgaver fra en mattebok. Problemet er bare å regne slike i en datamaskin. Grunnen til det er for eksempel hvis vi skal regne ut $80C10$, altså antall måter å trekke ut en gruppe på 10 personer ut av en større gruppe på 80 personer. Vi ender opp med formelen,

$$\frac{80!}{(80-10)! \cdot 10!} = \frac{80!}{70! \cdot 10!}$$

det er håpløst i regne ut $80!$ direkte ved hjelp av standardbiblioteket i C++. Derimot kan vi bruke et triks med at,

$$\frac{80!}{70!} = 80 \cdot 79 \cdot \dots \cdot 72 \cdot 71$$

Men fortsatt er det et veldig stort tall $\approx 4.6 \cdot 10^{20}$. For stort for standardbiblioteket. Vi skal jo også dele dette på $10!$, og vi ender da opp med en lite nok tall for C++. Men hvordan deler vi dette store tallet hvis vi ikke kan regne det ut? Vi må dele mens vi regner ut $80 \cdot 79 \cdot \dots \cdot 72 \cdot 71$ med $10!$, slik at svaret ikke blir for stort. Vi ganger altså tallene $80 \cdot 79 \cdot \dots \cdot 71$ mens vi deler på $1 \cdot 2 \cdot \dots \cdot 10$. Vi gjør følgende $\frac{80}{1} = 80$, $\frac{80 \cdot 79}{2} = 3160$, $\frac{3160 \cdot 78}{3} = 82160$ osv. Så hvordan vet vi at dette vil fungere, vil ingen av tallene bli et desimaltall? Nei, siden vi starter med å dele 80 på 1, dette vil alltid gå opp. Neste vi gjør er å ta dette svaret (80) og gange det med neste tall 79 og dele dette på 2. Grunnen til at $80 \cdot 79$ er delelig på 2 er at produktet av 2 etterfølgende tall må være delelig med 2 siden ett av dem er partall. Neste steg er å ta det svaret vi fikk (3160) og gange dette med neste tall 78 og dele dette på 3. Grunnen til at $3160 \cdot 78$ er delelig med 3 er at ett av de opprinnelige tallene 80, 79, 78 må være i 3 gangeren siden det er tre

etterfølgende tall. Dermed må også $\frac{80 \cdot 79}{2} \cdot 78$ være delelig med 3. Vi fortsetter slikt til vi har ganget med 71 og delt med 10.

Her er en eksempelkode som bruker denne teknikken (vi bruker long long siden svaret kan ofte bli ganske stort),

```
long long binom(long long n, long long r) {
    if (r > n/2)
        r = n-r;

    long long ans = 1;
    for (long long i = n; i > n-r; i--) {
        ans *= i;
        ans /= n-i;
    }
    return ans;
}
```

Legg merke til i starten sjekker vi om r er større enn $n/2$. Siden binominalkoeffisienter er symmetriske om $n/2$ med hensyn på r kan vi like godt speile r om $n/2$ som er å sette $r = n - r$. Dette sparer oss for mange unødvendige multiplikasjoner og divisjoner. Matematisk vil det si at,

$$\binom{n}{r} = \binom{n}{n-r}$$

Vi erstatter dermed r med den minste verdien an r og $n - r$.

Som et eksempel ser vi på **NIO-ball fra NIO-finalen 2012/2013**.

Ballspillet NIO-ball ligner litt på fotball, med unntak av at det spilles med 60000 spillere på hvert lag, og en litt sær regel for når en scoring gir poeng: En scoring gir kun poeng dersom de fem siste spillerene fra laget som scorer som har vært borti ballen har trøye med nummer i strengt stigende rekkefølge. Alle spillerene på hvert lag har trøyer nummerert 1 til 60000. Hver spillers tall er unikt. Gitt nummeret på spilleren som scoret, skriv ut hvor mange kombinasjoner av de siste fem spillerene på ballen som gir gyldig mål.

Input: Input består av ett heltall $1 \leq n \leq 60,000$ som er draktnummeret på den siste spilleren som var nær ballen.

Output: Output består av ett heltall, antall kombinasjoner av de siste fem spillerene på ballen som gir gyldig mål.

Ved hjelp av kombinatorikk ser vi at vi er ute etter å se hvor mange måter vi kan plukke ut 4 personer av de $n - 1$ personene som har et draktnummer som er mindre enn n . Siden rekkefølgen ikke har noe å si kombinatorisk, blir svaret $(n-1)C4$. Grunnen til at rekkefølgen ikke har noe å si er fordi når vi har plukket ut de 4 personene kan vi ordne dem i stigende rekkefølge etterpå. Effektiviteten i utregningen er konstant, siden antall multiplikasjoner er konstant lik 4 og uavhengig av hvor stor n er har vi en kompleksitet lik $\mathcal{O}(1)$.

```
int n;
cin >> n;
cout << binom(n-1, 4) << endl;
```

Der **binom** er funksjonen vi skrev ovenfor.

4.5 Utvalg (2^n)

I noen oppgaver trenger man å kunne generere alle utvalg av en gitt mengde. Hvis vi har 3 personer Per, Pål og Espen er alle utvalgene lik,

{}, {Per}, {Pål}, {Espen}, {Per, Pål}, {Per, Espen}, {Pål, Espen}, {Per, Pål, Espen}

Så hvordan genererer vi alle slike utvalg? En måte er å generere alle utvalgene rekursivt med denne koden,

```
string navn[3] = {"Per", "Paal", "Espen"};

void alleUtvalg(int pos, vector<string> v) {
    if (pos == 3) { //vi har vaert gjennom alle navnene
        //skriver dem ut
        for (int i = 0; i < v.size(); i++)
            cout << v[i] << endl;
        cout << "---" << endl;
    } else {
        //vi har 2 valg, gaa til neste navn eller
        //ta med personen i listen
        alleUtvalg(pos + 1, v);

        v.push_back(navn[pos]);
        alleUtvalg(pos + 1, v);
    }
}
```

Prøv å kjøre koden med å starte med en tom vector,

```
alleUtvalg(0, vector<string> (0));
```

Og se at vi får alle utvalgene, inkludert den tomme mengden. Analyser hvorfor de kommer ut i den rekkefølgen de gjør.

En annen måte å kode det samme på, som kan være litt ryddigere, er å bruke egenskapene til totallsystemet. Hvis vi ser på bittene til tallene fra og med 0 opp til en toerpotens får vi alle utvalgene vi er ute etter. La oss se på alle tallene fra 0 opp til $7 (2^3 - 1)$.

	2^2	2^1	2^0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Hvis du studerer tabellen ser du at vi får alle utvalg av bittene, tenk deg at hver bitposisjon symboliserer hvert navn (Per, Pål og Espen). Der 1 betyr ta med navnet, og 0 betyr ikke ta med. Vi kan da kode dette slikt,

```
string navn[3] = {"Per", "Paal", "Espen"};
for (int i = 0; i < (1 << 3); i++) {
    for (int j = 0; j < 3; j++)
```

```

        if ((i & (1 << j)) != 0) //sjekker om bitten er "paa"
            cout << navn[j] << " ";

        cout << endl << "----" << endl;
    }

```

Bortsett fra rekkefølgen får vi det samme som den rekursive funksjonen. Vi skal se hvordan vi kan bruke disse utvalgene for å få poeng på en NIO-oppgave. **Lekeplass, NIO-finalen 2013/2014:**

Forskning har vist at i alle barnehager så grupperer barna seg automatisk inn i to vennegjenger. Når barna er ute på lekeplassen så vil de typisk unngå barn fra den andre gjengen og bare leke med andre fra sin egen gjeng. For å passe på at barna ikke blir alt for adskilte er det viktig å vite hva avstanden mellom disse to vennegjengene er. Denne avstanden er definert som den minste avstanden mellom to barn som er i forskjellige vennegjenger. Dersom denne avstanden er stor så er det fare for at samholdet i barnehagen blir veldig dårlig etterhvert. Det er veldig vanskelig å finne ut av hvilke barn som er med i hvilken vennegjeng. Du har derfor blitt bedt om å lage et program som skal kunne finne ut av dette. Gitt posisjonen til barna på lekeplassen skal programmet ditt finne den inndelingen av barna i to vennegjenger som gjør at avstanden mellom de to gjengene blir størst mulig.

Input

Første linje inneholder et heltall N ($2 \leq N \leq 1,000$) - antall barn i barnehagen. Deretter kommer N linjer med to heltall X_i, Y_i hver ($0 \leq X_i \leq 1,000,000$, $0 \leq Y_i \leq 1,000,000$), som er posisjonen til det i 'te barnet. I testsett verdt 40 poeng vil $N \leq 20$.

Output

Ett flyttall som angir den største avstanden mellom de to vennegjengene.

Vi skal nå se på hvordan vi kan kode en løsning for å få de 40 poengene. En fullstendig løsning skal vi se på senere når vi skal snakke om Grafer. En slik løsning vi nå skal kode kalles en Brute Force løsning, vi bruker nemlig det at datamaskinen er veldig raskt til å kunne finne en løsning på en måte som er "dum". Vi skal nemlig lage et dataprogram som lager alle mulige inndelinger av de to vennegruppene. Vi finner så korteste avstand mellom to barn som er i forskjellige vennegrupper, og sjekker om dette er større enn de andre inndelingen vi har gjort før. Siden vi skal ha to vennegrupper trenger vi ikke å sjekke bittene i tallet 0 og $(2^N - 1)$ siden dette symboliserer at alle barna er i samme vennegruppe (alle bittene er like). Under er koden,

```

typedef long long ll;
#define INF 1000000000000000LL

ll X[1000], Y[1000];

ll sqrDist(int p1, int p2) {
    return (X[p2] - X[p1]) * (X[p2] - X[p1]) +
           (Y[p2] - Y[p1]) * (Y[p2] - Y[p1]);
}

int main() {

```

```

int N;
cin >> N;

for (int i = 0; i < N; i++)
    cin >> X[i] >> Y[i];

ll maxSqrDist = 0;
for (int bitmask = 1; bitmask < ((1 << N) - 1); bitmask++) {
    ll minSqrDist = INF;

    //ser paa avstanden mellom alle par av barn
    //fra de to forskjellige gruppene
    for (int barn1 = 0; barn1 < N; barn1++) {
        bool enerBit = ((bitmask & (1 << barn1)) != 0);
        for (int barn2 = barn1 + 1; barn2 < N; barn2++) {
            if (enerBit == ((bitmask & (1 << barn2)) != 0))
                continue; //er i samme vennegruppe
            minSqrDist = min(minSqrDist,
                             sqrDist(barn1, barn2));
        }
        maxSqrDist = max(maxSqrDist, minSqrDist);
    }
    cout << sqrt(maxSqrDist) << endl;
}

```

Siden **bitmask** går fra 1 til $2^N - 2 \approx 2^N$ og **barn1** fra $0 - N$ og **barn2** fra **barn1** - N . Blir kompleksiteten til algoritmen lik $\mathcal{O}(2^N * N^2)$ (konstantledd forsvinner). Siden den virkelige kompleksiteten er litt mindre enn dette er algoritmen akkurat rask nok for $N \leq 20$.

4.6 Permutasjoner

Noen ganger er vi ute etter å finne alle permutasjonene av en mengde elementer. Si vi har de tre bokstavene (A, B, C). Alle permutasjonene av disse tre bokstavene er,

$$(A, B, C) (A, C, B) (B, A, C) (B, C, A) (C, A, B) (C, B, A)$$

Vi får 6 permutasjoner av de 3 elementene siden på den første plassen har vi 3 muligheter for hvilken bokstav som skal stå der. Neste plassen har vi kun 2 muligheter, og den siste er det kun én mulighet igjen. I alt $3! = 6$ permutasjoner. Vi kan lage et C++-program som genererer alle disse permutasjonene. Det finnes faktisk en egen funksjon i C++ som generer de for oss! Se på koden under,

```

vector<int> v;
for (int i = 1; i <= 3; i++)
    v.push_back(i);
do {
    for (int i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    cout << endl;
} while (next_permutation(v.begin(), v.end()));

```

For å generere alle permutasjonene kreves det at listen er sortert før du begynner. `next_permutation` returnerer **false** når den har gått gjennom alle permutasjonene, altså når **v** er en synkende liste (3, 2, 1). I tillegg til å returnere **false** gjør den om listen til å bli den første permutasjonen (1, 2, 3). Det som

gjør `next_permutation` så smart er at hvis vi legger til en liste som inneholder duplikater, f.eks. (1, 2, 2). Så genererer den kun forskjellige permutasjoner,

$$(1, 2, 2) (2, 1, 2) (2, 2, 1)$$

Vi får kun 3 forskjellige permutasjoner, mens i stad fikk vi hele 6 permutasjoner. Det er fordi den skjønner at vi har to like 2-tall, og dermed vil alle permutasjoner der de to 2-tallene har byttet plass være like. Vi får da at antall permutasjoner blir,

$$\frac{3!}{2!} = \frac{6}{2} = 3$$

Vi kan også bruke `next_permutation` på en array, la oss prøve å kjøre den på en array med bokstaver som i det øverste eksempelet.

```
char v[] = {'A', 'B', 'C'};
do {
    for (int i = 0; i < 3; i++)
        cout << v[i] << " ";
    cout << endl;
} while (next_permutation(v, v + 3));
```

For en dokumentasjon på `next_permutation` se http://www.cplusplus.com/reference/algorithm/next_permutation/.

4.7 Modulo

I noen oppgaver blir du bedt om å regne ut et stort tall, men siden slike store tall er knotete å arbeide med blir du ofte ikke bedt om å regne ut selve tallet. Ofte blir du heller bedt om å finne ut hva er resten dersom vi dividerer det store tallet på et annet tall. Si vi skal regne ut $40!$, dette er et alt for stort tall til å lagres i en **long long**. Derimot er vi heller interessert i hva er $40!$ modulo 131071 (et primtall). Dette er det samme som å spørre: hva blir resten dersom vi deler $40!$ på 131071. Vi har en operator i C++ som ser ut som et prosentymbol (%). Dette er operatoren for å regne med rester/modulo. Vi vet at hvis vi deler 11 på 3 vil vi få 2 til rest siden $11 = 3 \cdot 3 + 2$. I C++ kan vi regne ut dette slikt,

```
int rest = 11 % 3;
//rest = 2...
```

Vi kan også sjekke om et tall er odde eller et partall med å se på resten når vi deler på 2,

```
int a;
cin >> a;
if ((a % 2) == 0)
    cout << a << " er et partall\n";
else
    cout << a << " er et oddetall\n";
```

Men hvis vi er interessert i hva $40! \% 131071$ er kan vi ikke først regne ut $40!$ siden det er for stort. Heldigvis er det slikt at med moduloregning kan vi da modulo underveis mens vi regner ut (dette lærer du i tallteori). Si vi er interessert i hva $(11 + 14) \% 3$ er. Vi kan enten regne ut at $11 + 14 = 25$ og vi vet da at $25 \% 3$ er 1. Vi kan også finne ut at $11 \% 3 = 2$, og at $14 \% 3 = 2$. Summerer vi restene får vi at $2 + 2 = 4$. Vi regner så dette svaret modulo 3 og får at $4 \% 3 = 1$. Som er akkurat det samme svaret som vi fikk i stad! Vi kan også gjøre tilsvarende for

multiplikasjon. Vi ser at $(10 \cdot 9) \% 3 = 0$. Vi har på en annen side at $10 \% 3 = 1$ og at $9 \% 3 = 0$. Ganger vi restene sammen får vi $1 \cdot 0 = 0$, som er det vi fikk istad. Vi kan da kode 40!%131071 slikt,

```
int ans = 1;
for (int i = 2; i <= 40; i++) {
    ans *= i;
    ans = ans % 131071;
}
```

Vi får svaret 72026.

Du lurer kanskje på om mellomregningene blir for store. Husk bare at når vi regner ut resten når vi deler på N vil alltid svaret ligge mellom $0 \dots N - 1$. Vi vet da at i eksemplet ovenfor at den største verdien til **ans** er $(131071 - 1) \cdot 40$. Noe som er godt innenfor en **int**.

4.8 Potenser

Noen ganger har vi behov for å regne ut potenser. Si vi skal regne ut 4^8 , det kan vi gjøre på flere måter. Vi kan enten skrive,

```
int potens = 4 * 4 * 4 * 4 * 4 * 4 * 4 * 4 * 4;
```

eller vi kan skrive

```
int potens = 1;
for (int i = 0; i < 8; i++)
    potens *= 4;
```

Men uansett hvilke av disse to måtene vi velger får vi hele 7 multiplikasjoner! C++ har også en innebygd potensfunksjon (engelsk: power) som også tar desimaltall,

```
int potens = pow(4.0, 8);
```

Den er også treg siden den krever 7 multiplikasjoner, men vi kan gjøre noe smartere!

Hvis vi i stedet regner ut $4^8 = 65536$ slikt, $((4 \cdot 4)(4 \cdot 4))((4 \cdot 4)(4 \cdot 4))$ som på en annen måte betyr at vi kan regne slikt,

$$4^2 = (4 \cdot 4) = 16, \quad 4^4 = 4^2 \cdot 4^2 = 16 \cdot 16 = 256, \quad 4^8 = 4^4 \cdot 4^4 = 256 \cdot 256 = 65536$$

Så vi bruker 1 multiplikasjon for å få 16, vi bruker den 16 vi nå har og 1 multiplikasjon og får 256. Så bruker vi en multiplikasjon for å få $65536 = 4^8$. Totalt brukte vi 3 multiplikasjoner i stedet for 7! La oss se hvor mange multiplikasjoner vi trenger for å regne ut 3^5 . $3^2 = 3 \cdot 3 = 9$ derifra kan vi regne ut $3^4 = 3^2 \cdot 3^2 = 9 \cdot 9 = 81$. For å regne ut sluttsvaret 3^5 kan vi nå regne ut $3^5 = 3 \cdot 3^4 = 3 \cdot 81 = 243$. Vi trengte da kun 3 multiplikasjoner. Det virker kanskje ikke så imponerende at vi kun gjorde 3 multiplikasjoner i stedet for normalt 4 for å regne ut 3^5 . Men med store eksponenter sparer vi mye med denne teknikken. Vi skal se på hvordan en slik funksjon kan kodes, jeg har valgt å bruke et triks med binære tall (funksjonen regner ut a^n der n er ett heltall),

```
double eksponent(double a, int n) {
    double ans = 1.0;
    double faktor = a;

    while (n > 0) {
```

```

        if (n & 1)
            ans *= faktor;

        faktor *= faktor;
        n /= 2;
    }
    return ans;
}

```

Som oppgave kan du jo skrive 4 og 8 som binære tall for å regne ut 4^8 , og så forstå hvorfor denne algoritmen fungerer. Legg også merke til at $(n \& 1)$ er ekvivalent med $(n \& (1 \ll 0))$.

4.9 Primtall

Som du kanskje vet er primtall viktig innenfor matematikk, men også veldig viktig innenfor informatikk! Man bruker blant annet primtall til diverse krypteringer, som f.eks. i nettbanker. Vi skal se hvordan vi kan generere primtall, sjekke om ett tall er et primtall og primtallsfaktoriserer ett heltall. Så la oss starte med den første, finne primtall og vi skal bruke teknikken som er beskrevet her http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes. Si vi vil finne alle primtall opp til 1,000 (husk at 1,000 ikke er et primtall, så slipper å sjekke den). Da kan vi kode slikt,

```

bool erPrimtall[1000];
vector<int> primtall;

for (int i = 0; i < 1000; i++)
    erPrimtall[i] = true; //alle tall er potensielle primtall
erPrimtall[0] = erPrimtall[1] = false;

for (int i = 0; i < 1000; i++) {
    if (erPrimtall[i]) {
        primtall.push_back(i);
        for (int j = i * i; j < 1000; j+=i)
            erPrimtall[j] = false;
    }
}

```

Nå vil vi ha en vector med navn **primtall** som inneholder alle primtallene opp til 1,000 i stigende rekkefølge, og i tillegg har vi **erPrimtall** som enten er **True** eller **False** avhengig om tallet er et primtall eller ikke. Vi kan da slå opp på om 59 er et primtall eller ikke med å se på **erPrimtall[59]**. Dette skjer da i konstanttid. Legg også merke til at vi starter med $j = i*i$. Grunnen til det er at hvis vi f.eks. finner ut at $i = 3$ er et primtall, da er det ikke noe vits å starte med $j = i*2 = 6$ siden 6 allerede er i 2 gangeren og er da allerede “krysset” ut. Vi kan heller starte på $j = i*i = 3*3 = 9$ siden det er det første tallet i 3 gangeren som IKKE er krysset ut av noen av de andre primtallene (kun en optimalisering). I tillegg kan du brukes **bitset** (<http://www.cplusplus.com/reference/bitset/bitset/>) i stedet for **bool [1000]** for å spare litt minne og for raskere avkrysning.

Vi skal nå primtallsfaktoriserer tall. Vi skal faktoriserer tallet 1092. Vi ser at 1092 er delelig med 2 (partall), vi får da at $1092 = 2 \cdot 546$. Men 546 er også et partall, og dermed delelig med 2! Vi har at $1092 = 2 \cdot 2 \cdot 273$. Vi ser så at tallet er delelig med 3 (tversummen er delelig med 3). Vi har nå $1092 = 2 \cdot 2 \cdot 3 \cdot 91$. Vi ser at 91 ikke lenger er delelig med 3. Vi går da til neste primtall, 5. Men 91

er ikke delelig med 5. Så vi går til neste primtall 7. 91 er delelig med 7. Vi har nå at $1092 = 2 \cdot 2 \cdot 3 \cdot 7 \cdot 13$. Siden 13 også er et primtall har vi faktorisert 1092 fullstendig. Du kan lure på hvorfor vi hoppet rett fra 5 til 7 eller fra 3 til 5? Det er fordi hvis tallet hadde vært delelig med $6 = 2 \cdot 3$ ville også tallet vært delelig med 2 og 3. Siden vi viste at tallet ikke lenger var delelig med 2, 3 kunne den heller ikke vært delelig med 6. Det samme med $4 = 2 \cdot 2$. Dermed trenger vi kun å sjekke primtallene.

Her er en kode som faktoreriserer, den antar at du allerede har primtallslisten (**primtall**) som vi genererte i koden ovenfor.

```
vector<int> faktoreriser(int N) {
    vector<int> faktor;
    for (int i = 0; i < primtall.size() && N > 1; i++) {
        while ((N % primtall[i]) == 0) {
            faktor.push_back(primtall[i]);
            N /= primtall[i];
        }
    }

    if (N > 1)
        faktor.push_back(N);

    return faktor;
}
```

Legg merke til på slutten at vi sjekker om $N > 1$. Det vil betyr at det som er igjen av N er større enn tusen og har ingen faktorer under tusen. Vi antar da med mindre enn 100% sikkerhet at det som er igjen av N er et primtall. Grunnen til at vi vet at N er større enn tusen er fordi N ikke lenger har noen faktorer som er mindre enn 1,000. Siden de aller fleste tallene har faktorer som er mindre enn tusen, antar vi at N er et primtall.

4.10 Diverse

4.10.1 Finne sifrene i et tall

Vi skal se på en kort metode for å finne sifrene i et heltall på en datamaskin. Legg merke til at hvis vi har tallet 1234 vil vi få 4 til rest dersom vi deler tallet på 10. Generelt vil vi for heltall alltid få sifferet på enerplassen som rest når vi deler tallet på 10. Etter vi har fått vite at sifferet på enerplassen er 4 kan vi dele tallet på 10, og da står vi igjen med tallet 123 (siden i C++ vil divisjonen runde ned fra 123.4 til 123). Vi fortsetter så med å dele tallet på 10 og ser på resten. Slikt holder vi på med helt til det vi står igjen med er 0. Under er algoritmen som gir ut en vector med sifrene med enerplassen først, så tierplassen osv.

```
vector<int> siffre(int N) {
    vector<int> ans;
    if (N == 0)
        ans.push_back(0);
    while (N != 0) {
        ans.push_back(abs(N % 10));
        N /= 10;
    }
    return ans;
}
```

Grunnen til at vi sjekker om $N = 0$, og har lagt ved **abs** er fordi funksjonen også skal fungere for $N \leq 0$.

4.10.2 GCD og LCM

Vi starter med GCD (greatest common divisor, http://no.wikipedia.org/wiki/St%C3%B8rste_felles_divisor).

```
int GCD(int a, int b) {
    if (b == 0)
        return a;
    return GCD(b, a % b);
}
```

Vi har også en enkel funksjon for LCM (least common multiple, http://no.wikipedia.org/wiki/Minste_felles_multiplum).

```
int LCM(int a, int b) {
    return (a / GCD(a, b)) * b;
}
```

Legg merke til at vi først deler **a** med **GCD(a, b)** siden vi vet at delestykke går opp per definisjon. Vi slipper da å regne med det potensielt store tallet $a * b$.

Så hva skal vi med GCD og LCM? En grunn til å programmere GCD er for å programmere LCM. Men hva skal vi med LCM? Faktisk er det én av de viktigste funksjonene innenfor tallteorien. NIO ligger heller ikke bakpå, ettersom man kunne bruke LCM i en finaleoppgave i NIO. Vi tar en titt! **Kortstokkemaskinen, NIO-finalen 2005/2006:**

Pokerfeberen rir landet, og du og vennene dine har forlenget gjort Texas Hold'em til altopplukende fritidsaktivitet. Dere begynner imidlertid å bli grundig lei av å måtte stokke kortene mellom hver eneste runde. Derfor har dere nettopp gått til innkjøp av den fantastiske kortstokkemaskinen Deterministic Card Shuffler. Til deres store skuffelse oppdager dere at maskinen stokker kortene på samme måte hver eneste gang (kanskje dere skulle ha studert navnet på maskinen litt nærmere før dere kjøpte den). Firmaet som solgte maskinen er merkelig nok ikke mulig å komme i kontakt med, så dere får ikke pengene tilbake. Derfor tenker dere å ta maskinen i bruk likevel, men dere lurer på hvor godt den sorterer. For å finne ut dette tenker dere blant annet å la maskinen sortere en kortstokk gjentatte ganger og se hvor mange omganger som trengs for at kortene skal komme tilbake i den opprinnelige rekkefølgen.

Input

Input består av to linjer. Den første linjen inneholder et tall $2 \leq n \leq 500,000$, som beskriver antall kort i kortstokken. Alle kortene er unike, og de er nummerert fra og med 0 til og med $n - 1$. Den andre linjen inneholder n tall $a_0, a_1, a_2, \dots, a_{n-1}$ som beskriver mønsteret maskinen stokker kortene etter: kortet på plass i flyttes til plass a_i . Alle tallene mellom 0 og $n - 1$ vil være representert på denne linjen. Tallene vil aldri være helt sortert (som f.eks. 01234); du vil altså aldri få en maskin som ikke stokker kortene i det hele tatt.

Output

En linje med ett tall på: Antallet ganger en kortstokk må sendes gjennom maskinen for at den opprinnelige rekkefølgen skal komme tilbake. Husk linjeskift etter tallet. Du kan stole på at resultatet vil være mindre enn eller lik 2, 147, 483, 647, slik at det vil få plass i en **int**.

For å løse dette problemet må vi dessverre bruke `scanf` for å lese inn tallene. Dette er fordi vi i verste fall må lese inn hele 500,000 tall som med `cin` tar over ett sekund (vanligvis klarer `cin` å lese hundretusen tall i sekundet). En måte å løse problemet på er for hvert kort å se hvor mange stokkinger vi trenger for at kortet skal komme tilbake til sin opprinnelige posisjon. Disse verdiene kan vi lagre i en tabell `dist`. Tenk deg at vi kun fokuserer på to av kortene og vi skal finne ut hva er det minste antall stokkinger vi trenger for at de to kortene skal begge befinne seg på sin opprinnelige posisjon igjen. Hvis vi tenker oss at de to kortene har opprinnelig posisjon i og j . Da vet vi at etter `dist[i]` stokkinger vil kortet i være på sin opprinnelige posisjon. Stokker vi `dist[i]` ganger til vil fortsatt i befinne seg på sin opprinnelig posisjon, men aldri i mellom stokkingene siden `dist[i]` er minste antall stokkinger for at kortet i skal stokkes tilbake på plass. Så kun etter at vi har stokket kortene $k \cdot \text{dist}[i]$ ganger for et heltall k vil kortet i befinne seg på sin opprinnelige posisjon. Tilsvarende vil kortet j befinne seg på sin opprinnelige posisjon kun når antall stokkinger er på formen $q \cdot \text{dist}[j]$ for et heltall q . Sagt på en annen måte betyr det at hvis kort nr. j skal befinne seg på sin opprinnelige posisjon må antall stokkinger være delelig med `dist[j]`. Tilsvarende gjelder også for i . Så det minste antall stokkinger for at begge kortene i og j skal befinne seg på sin opprinnelige posisjon er da det minste tallet som er delelig med både `dist[i]` og `dist[j]` altså `LCM(dist[i], dist[j])`. Nå fokuserte vi kun på at to av kortene skal befinne seg på sin opprinnelige posisjon, men vi kan utvide dette til tre kort. Vi vet at kortene i og j befinner seg kun samtidig på sine opprinnelige posisjoner hvis antall stokkinger er på formen $p \cdot \text{LCM}(\text{dist}[i], \text{dist}[j])$ for et heltall p . Hvis vi setter $z = \text{LCM}(\text{dist}[i], \text{dist}[j])$ er de tre kortene i , j og m alle sammen på sin opprinnelige posisjon første gang etter `LCM(z, dist[m])` stokkinger. Så kan vi utvide det til fire kort osv. Under er koden,

```

typedef long long ll;
int main () {
    int N;
    scanf("%d", &N);

    vector<int> a(N);
    for (int i = 0; i < N; ++i)
        scanf("%d", &a[i]);

    vector<int> dist(N, 1);
    for (int i = 0; i < N; ++i) {
        int pos = a[i];
        while (pos != i) {
            dist[i]++;
            pos = a[pos];
        }
    }

    ll z = LCM(dist[0], dist[1]);
    for (int i = 2; i < N; ++i)
        z = LCM(z, dist[i]);

    printf("%d\n", z);
    return 0;
}

```

Pga. store tall burde du nok implementere `GCD` og `LCM` til å bruke `long long` i stedet for `int`.

4.11 Oppgaver

Generelle oppgaver:

ID	Navn
10055	Hashmat the Brave Warrior
10071	Back to High School Physics
10281	Average Speed
10469	To Carry or not to Carry
10773	Back to Intermediate Math (*)
11723	Numbering Roads (*)
11875	Brick Game (*)
10137	The Trip (*)
10079	Pizza Cutting

Bitmanipulasjon:

ID	Navn
594	One Little, Two Little, Three Little Endians
10264	The Most Potent Corner (*)
11173	Grey Codes
11926	Multitasking (*)
11933	Splitting Numbers (*)

Matematisk simulering:

ID	Navn
100	The $3n + 1$ problem
371	Ackermann Functions
382	Perfection (*)
1225	Digit Counting (*)
10035	Primary Arithmetic
10346	Peter's Smokes (*)
10370	Above Average
10783	Odd Sum
10879	Code Refactoring
11150	Cola (*)
11877	The Coco-Cola Store
616	Coconuts, Revisited (*)
11254	Consecutive Integers (*)

Matematiske følger og systemer:

ID	Navn
136	Ugly Numbers
443	Humble Numbers (*)
694	The Collatz Sequence
10042	Smith Numbers (*)

Logaritmer og eksponenter:

ID	Navn
113	Power of Cryptography

Polynomer:

ID	Navn
498	Polly the Polynomial
10268	498-bis (*)

Tallsystemer:

ID	Navn
575	Skew Binary (*)
10931	Parity (*)

Binomialkoeffisienter:

ID	Navn
369	Combinations
530	Binomial Showdown
10219	Find the ways ! (*)

Printall:

ID	Navn
543	Goldbach's Conjecture (*)
686	Goldbach's Conjecture (II)

GCD, LCM:

ID	Navn
10407	Simple division (*)
11827	Maximum GCD (*)

Modulo:

ID	Navn
374	Big Mod (*)
10127	Ones
10176	Ocean Deep ! - Make it shallow !! (*)

Tallteori:

ID	Navn
10110	Light, more light (*)

5 Datastrukturer

Vi skal nå se på noen kjente datastrukturer. Du er kanskje kjent med noen datastrukturer fra før av som queue, stack, priority_queue, map, set osv.? Vi skal nå kode våre egne datastrukturer som er nyttige til diverse algoritmer vi senere skal se på.

5.1 Union-Find

Union-Find er den første datastrukturen vi skal programmere. Si du nettopp har startet på videregående og det er 30 personer i klassen din. Ingen kjenner hverandre fra før av, så alle kjenner kun seg selv (håper vi!). Etterhvert som tiden går blir to og to personer venner, og da blir også deres venner venner med hverandre. Så når vi får vite hvem som blir venn med hvem, hvordan kan vi holde styr på hvem som egentlig er venn med hvem? Si vi har 4 personer: Gates, Jobs, Brin og Zuckerberg. Ingen av dem er venner. Så blir Brin og Gates venner. Da har vi 3 vennegrupper, (Gates og Brin), (Jobs) og (Zuckerberg). Så blir Zuckerberg og Jobs venner. Vi står nå igjen med 2 vennegrupper: (Gates og Brin) og (Jobs og Zuckerberg). Så blir Brin og Zuckerberg venner. Da har vi kun 1 vennegruppe igjen, nemlig (Gates, Jobs, Brin og Zuckerberg). Så hvis vi spør datastrukturen vår om Zuckerberg og Gates er venner vil den si ja. Så hvordan koder man en slik datastruktur? Vi bruker en slags leder i hver vennegruppe. Så først har vi 4 vennegrupper: Gates, Jobs, Brin og Zuckerberg. De er hver sin egen leder. Når Gates og Brin blir venner avtaler de at Gates skal bli lederen for dem begge. Så hvis vi spør Brin om hvem som er lederen hans svarer han Gates, det samme svarer også Gates (seg selv). Når Zuckerberg og Jobs blir venner avtaler de at Jobs blir lederen av deres vennegruppe. Så

hvis vi spør hvem som er lederen til Zuckerberg svarer han Jobs, det samme vil også Jobs svare (seg selv). Men så blir Brin og Zuckerberg venner. De avtaler så at gruppen til Zuckerberg skal joine gruppen til Brin. Vi spør først hvem som er lederen til Brin, vi får at det er Gates. Vi spør så hvem som er lederen til Zuckerberg, og vi får at det er Jobs. Vi forteller så Jobs at Gates skal bli hans nye leder. Vi hvis vi spør Jobs om hvem som er lederen hans svarer han Gates. Derimot hvis vi spør Zuckerberg om hvem som er lederen hans, spør han først sin forrige leder Jobs om hvem som er lederen til Jobs, Jobs svarer Gates, og dermed svarer Zuckerberg til oss at hans nye leder er Gates. Og slik holder det på. Nedenfor er en graf som viser eksempelet ovenfor.



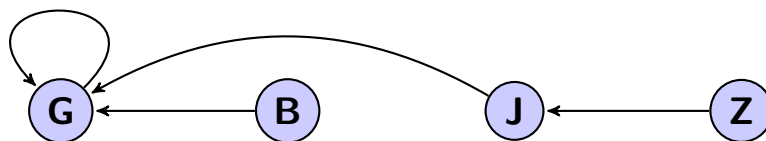
Gates og Brin blir venner.



Zuckerberg og Jobs blir venner.



Brin og Zuckerberg blir venner.



Så hvis vi spør Zuckerberg om hvem som er lederen hans, går han der pilen peker (Jobs), og ser så at Jobs er ikke lederen i gruppen deres, han går så videre til der Jobs peker (Gates) og ser da at Gates er lederen av gruppa. Da vet Zuckerberg at Gates er lederen av gruppa. Dermed kan vi også fort sjekke om f.eks. Zuckerberg og Brin er i samme gruppe, det er bare å sjekke om de oppgir samme leder av siden egen gruppe. Vi kan også holde styr på hvor mange grupper det finnes og hvor mange personer det er i hver gruppe. Se under for kode,

```

class UnionFind {
private:
    vector<int> p, rank, setSize;
    //p er lederen til hver enkelt node
    //rank er kun for optimalisering naar vi slaar sammen
    grupper
    //setSize er antall noder i hver gruppe
    int numSets; //antall grupper
public:
    UnionFind(int N) {
        setSize.assign(N, 1); //er kun 1 i hver gruppe
        numSets = N; //N grupper
        rank.assign(N, 0);
        p.assign(N, 0);
        for (int i = 0; i < N; i++)
            p[i] = i;
    }
    int findSet(int i) {
        return (p[i] == i) ? i : (p[i] = findSet(p[i]));
        //p[i]=findSet(p[i]) er kun for aa optimalisere,
        //slik at den husker hvem som er lederen til neste gang
    }
    bool isSameSet(int i, int j) { return findSet(i) == findSet(
j); }
    void unionSet(int i, int j) {
        if (!isSameSet(i, j)) {
            numSets--;
            int x = findSet(i), y = findSet(j);
            // rank holder avstanden fra lederen til sine
            medlemer kort
            if (rank[x] > rank[y]) {
                p[y] = x;
                setSize[x] += setSize[y];
            } else {
                p[x] = y;
                setSize[y] += setSize[x];
                if (rank[x] == rank[y])
                    rank[y]++;
            }
        }
    }
    int numDisjointSets() { return numSets; }
    int sizeOfSet(int i) { return setSize[findSet(i)]; }
};

```

Vi kan da simulere eksempelet ovenfor slikt,

```

//0=Gates, 1=Brin, 2=Jobs, 3=Zuckerberg
UnionFind UF(4);
UF.unionSet(0, 1);
UF.unionSet(2, 3);
if (UF.inSameSet(0, 3))
    cout << "Gates og Zuckerberg er venner\n";
else
    cout << "Gates og Zuckerberg er ikke venner\n";

cout << "Antall grupper: " << UF.numDisjointSets() << endl;
cout << "Antall personer i gruppen som Brin er med i: " << UF.
sizeOfSet(1) << endl;

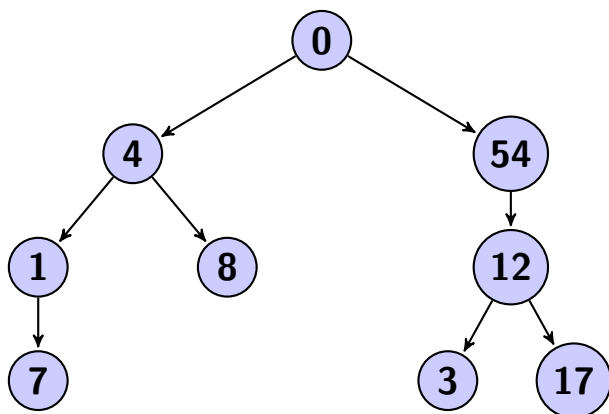
UF.unionSet(1, 3);
cout << "Lederen til Zuckerberg er: " << UF.findSet(3) << endl;

```

Vi skal få bruk for **Union-Find** senere. Pga. optimaliseringen i koden (under **findSet**) får vi en tilnærmet konstant kjøretid $\mathcal{O}(1)$.

5.2 Binary search tree (BST)

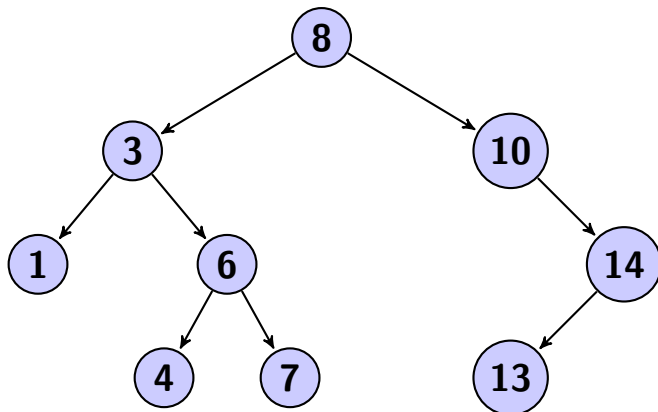
Vi skal nå programmere vårt første trestruktur! Et tre i informatikken er en graf (dette lærer du i neste kapittel) som har en rot, og hver node i grafen kan ha flere undernoder. Det er faktisk ikke nødvendig å ha en rot, men i de trærne vi skal se på er det en rot. Under er et eksempel på et tre,



Vi ser at node **0** er roten i treet, den har to child-noder, node **4** og node **54**. Node **4** har igjen to child-noder, nemlig node **1** og node **8**, osv.

Sitat fra Tom Lindstrøm, professor i matematikk ved UiO: *“I matematikk tegner vi alltid trær med roten nederst i treet og bygger treet oppover, men i informatikk har de roten øverst og bygger seg nedover. Det viser jo tydelig at informatikere ikke er så ofte ute i naturen!”*

Vi skal nå se på hvordan binært søketre (binary search tree) skiller seg ut fra et vanlig tre. I et binært søketre har hver node maksimalt to child-noder. Vi omtaler hver av child-nodene som leftchild og rightchild. I tillegg skal verdien i en node være større enn alle verdiene i venstre subtre, og mindre enn alle verdiene i høyre subtre. Her er et eksempel på et binært søketre,



Hvis vi først starter med å studere roten ser vi at alle nodene til venstre for roten har mindre verdi enn rotens verdi, og alle nodene til høyre har høyere verdi enn rotens verdi. Dette gjelder for alle nodene i hele treet! Det som gjør dette treet så praktisk er at hvis du står ved roten og vil finne ut om verdien 4 er inneholdt i treet vet du nøyaktig hvilken vei du skal ta for å nå den. Hvis i tillegg treet er balansert, altså at vi får et tree som har en minimal dybde (lengden fra roten til noden lengst unna roten), kan vi finne alle verdier i treet på $\mathcal{O}(\log N)$ tid, der N er antall noder. Selvfølgelig kunne vi løst det samme problemet med å lagre alle verdiene i en tabell, sortere den, og binærsøke alle verdiene vi vil finne. Men med et binært søketre har vi muligheten for å fjerne og legge til elementer i $\mathcal{O}(\log N)$ tid, istedet for $\mathcal{O}(N)$ i en vector. Blant annet bruker **set** og **map** i C++ slike binære søketre for å legge til/fjerne og søke etter elementer. Det er verdt å nevne at **set** og **map** er selvbalanserende, det vil si at den automatisk underveis som du legger/fjerner elementer holder dybden av treet minimalt. Dette er dessverre litt for komplisert for å ta med her, men for de interesserte kan lese om et slikt tre her http://en.wikipedia.org/wiki/Red%E2%80%93black_tree. Vi skal nå se på hvordan man lager et slikt tre i en datamaskin. Først ser vi hva som skjer hvis vi skal legge til verdien 5 i treet ovenfor. Vi går først til roten og ser at $5 < 8$, vi vandrer så til rotens leftchild. Vi havner så ved noden **3** og ser at $3 < 5$, altså vandrer vi til node **3** sin rightchild. Vi befinner oss nå i node **6**, og vet at $5 < 6$, altså vandrer vi til node **6** sin leftchild. Da ender vi opp på node **4**, siden $4 < 5$ prøver vi nå å vandre til node **4** sin rightchild. Da finner vi ut at node **4** ikke har noen rightchild, men da lager vi en rightchild for node **4** og setter inn node **5** der. Du kan jo tenke deg hva som skjer hvis du skal lage et binært søketre fra scratch og legger inn nodene i denne rekkefølgen: 1, 3, 4, 6, 7, 8, 10, 14, 13. Da vil vi ende opp med et tre som er svært ubalansert, nemlig en lang lenke/sti. Dermed kan det være lurt å velge medianen av mengden som rot, siden vi kan bruke roten til å dele binære søketreet i 2. Også kan leftchild og rightchild til roten være medianen av de to intervallene du nå står igjen med (øvre og nedre del). Dette er det idealet måten å gjøre det på. Når vi skal kode et binært søketre vil vi at hver node skal inneholde sin egen verdi, peker til left/right-child og (noen ganger) peker til sin parent-node. En node vil da se slikt ut,

```

struct sNode {
    int value;
    int left, right;
    int parent;
};

```

Alle nodene blir lagret i en vector, og **left**, **right** og **parent** er da indeksnummeret i vector-listen. Noen ganger er **left** og **right** udefinert, og vi setter da dem til -1 . Roten vil også ha en udefinert parent-node, og vi setter denne også til -1 . Vi har også at første elementet i vektoren er roten i treet. Under er koden for det binære søketreet,

```

struct sNode {
    int value;
    int left, right;
    int parent;

    sNode(int _value, int _left, int _right, int _parent) {
        value = _value;
        left = _left;
    }
};

```

```

        right = _right;
        parent = _parent;
    }
};

vector<sNode> node; //node[0] = rot-node

int finnNode(int value) { //returnerer indeksen til noden med
    verdi lik value
                                //returnerer -1 dersom noden ikke
                                finnes
    if (node.size() == 0)
        return -1;

    int curNode = 0;
    while (curNode != -1 && node[curNode].value != value) {
        if (value < node[curNode].value)
            curNode = node[curNode].left;
        else
            curNode = node[curNode].right;
    }

    return curNode;
}

int finnParentNode(int value) {
    //returnerer -1 hvis det ikke finnes noen rot-node
    if (node.size() == 0)
        return -1;

    int curNode = 0;

    while (true) {
        int nxtNode;
        if (value < node[curNode].value)
            nxtNode = node[curNode].left;
        else
            nxtNode = node[curNode].right;

        if (nxtNode == -1)
            return curNode;
        curNode = nxtNode;
    }
}

void leggTilNode(int value) { //legger til en node i treet med
    verdi lik value
    int par = finnParentNode(value);

    if (par != -1) { //finnes en parent-node
        if (value < node[par].value)
            node[par].left = node.size();
        else
            node[par].right = node.size();
    }

    node.push_back(sNode(value, -1, -1, par));
}

void leftPrint(int idx) { //skriver ut alle nodene i
    //venstre subtre foer den
    //skriver ut seg selv(idx)

```

```

//for saa aa skrive ut hoeyre subtre
    if (idx == -1)
        return;

    leftPrint(node[idx].left);
    cout << node[idx].value << endl;
    leftPrint(node[idx].right);
}

void rightPrint(int idx) { //skriver ut alle nodene i
                           //hoeyre subtre foer den
                           //skriver ut seg selv(idx)
                           //for saa aa skrive ut venstre
                           subtre

    if (idx == -1)
        return;

    rightPrint(node[idx].right);
    cout << node[idx].value << endl;
    rightPrint(node[idx].left);
}

int main() {
    int N;
    cin >> N;

    int tmp;
    for (int i=0; i<N; i++) {
        cin >> tmp;
        leggTilNode(tmp);
    }

    leftPrint(0);

    cout << "-----\n";

    rightPrint(0);
}

```

Poenget med **leftPrint** og **rightPrint** er hvordan man rekursivt kan gå ned i treet. Som en god øvelse kan du prøve å løse denne oppgaven om binære søketre-er: http://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=3769.

5.3 Segment tree (*)

Hvordan et segment-tre fungerer er beskrevet på NIOs-bootcamp side, <http://nio.no/bootcamp/?p=216>. Jeg har tenkt til å legge ved koden for å programmere et segment-tre. Dette segmenttreet er ikke helt som beskrevet i bootcamp. Den er ikke ment for å regne på summer, men i stedet finner den indeksen til det minste tallet i et gitt intervall. Det er enkelt å skrive den om til å finne største verdi eller å regne på summer som beskrevet i bootcamp. Grunnen til at jeg valgte å ikke skrive et segment-tre for summer er at datastrukturen Fenwick Tree er bedre til det.

```

class SegmentTree {
private: vector<int> st, A;
    int n;
    int left (int p) { return p << 1; } //vi lagrer venstre
                           child paa indeks p*2

```

```

int right(int p) { return (p << 1) + 1; } //og høyre child
paa indeks p*2+1

void build(int p, int L, int R) { // O(n log n)
    if (L == R)
        st[p] = L; //lagrer indeksen
    else { //rekursivt regnet ut verdiene
        build(left(p) , L , (L + R) / 2);
        build(right(p), (L + R) / 2 + 1, R );
        int p1 = st[left(p)], p2 = st[right(p)];
        st[p] = (A[p1] <= A[p2]) ? p1 : p2;
    }
}

int rmq(int p, int L, int R, int i, int j) { // O(log n)
    if (i > R || j < L) return -1; //segmentet er utenfor
    intervallet
    if (L >= i && R <= j) return st[p]; //paa insiden av
    intervallet

    //delvis innenfor intervallet, regner ut venstre og
    høyre del
    int p1 = rmq(left(p) , L , (L+R) / 2, i, j)
    ;
    int p2 = rmq(right(p), (L+R) / 2 + 1, R , i, j)
    ;

    if (p1 == -1) return p2; // hvis vi kommer utenfor
    segment forespoerselen
    if (p2 == -1) return p1;
    return (A[p1] <= A[p2]) ? p1 : p2;
}

int update_point(int p, int L, int R, int idx, int new_value
) {
    int i = idx, j = idx;

    // if the current interval does not intersect
    // the update interval, return this st node value!
    if (i > R || j < L)
        return st[p];

    // hvis intervallet er inkludert i oppdaterings omraadet
    // oppdater st[node]
    if (L == i && R == j) {
        A[i] = new_value; // oppdater under arrayen
        return st[p] = L; //denne indeksen
    }

    // regner ut minste i venstre
    // og høyre del av intervallet
    int p1, p2;
    p1 = update_point(left(p) , L , (L + R) /
    2, idx, new_value);
    p2 = update_point(right(p), (L + R) / 2 + 1, R
    , idx, new_value);

    // returnerer indeksen til det minste tallet
    return st[p] = (A[p1] <= A[p2]) ? p1 : p2;
}

public:

```

```

SegmentTree(const vector<int> &_A) {
    A = _A; //kopierer
    n = (int)A.size();
    st.assign(4 * n, 0); // lagrer stor nok vector med
        nuller
    build(1, 0, n - 1); //rekursivt bygger treet
}

int rmq(int i, int j) { return rmq(1, 0, n - 1, i, j); } //
    bare overloading

int update_point(int idx, int new_value) {
    return update_point(1, 0, n - 1, idx, new_value);
}
};

```

Her er hvordan segmenttreet kan brukes,

```

int arr[] = { 18, 17, 13, 19, 15, 11, 20 }; // orginal-arrayen
vector<int> A(arr, arr + 7); // lager en vector av arrayen
SegmentTree st(A);

printf("          idx    0, 1, 2, 3, 4, 5, 6\n");
printf("          A is {18,17,13,19,15, 11,20}\n");
printf("RMQ(1, 3) = %d\n", st.rmq(1, 3)); // svar = indeks 2
printf("RMQ(4, 6) = %d\n", st.rmq(4, 6)); // svar = indeks 5
printf("RMQ(3, 4) = %d\n", st.rmq(3, 4)); // svar = indeks 4
printf("RMQ(0, 0) = %d\n", st.rmq(0, 0)); // svar = indeks 0
printf("RMQ(0, 1) = %d\n", st.rmq(0, 1)); // svar = indeks 1
printf("RMQ(0, 6) = %d\n", st.rmq(0, 6)); // svar = indeks 5

printf("          idx    0, 1, 2, 3, 4, 5, 6\n");
printf("Naa, modifier A til {18,17,13,19,15,100,20}\n");
st.update_point(5, 100); // oppdaterer A[5] fra 11 til 100
printf("Disse verdiene forblir uforandret\n");
printf("RMQ(1, 3) = %d\n", st.rmq(1, 3)); // 2
printf("RMQ(3, 4) = %d\n", st.rmq(3, 4)); // 4
printf("RMQ(0, 0) = %d\n", st.rmq(0, 0)); // 0
printf("RMQ(0, 1) = %d\n", st.rmq(0, 1)); // 1
printf("Disse forandres\n");
printf("RMQ(0, 6) = %d\n", st.rmq(0, 6)); // 5->2
printf("RMQ(4, 6) = %d\n", st.rmq(4, 6)); // 5->4
printf("RMQ(4, 5) = %d\n", st.rmq(4, 5)); // 5->4

```

5.4 Fenwick tree (*)

Vi skal nå se på et Fenwick tre (også kalt Binary indexed tree), det brukes til å regne prefix-summer av en array. Tenk deg denne eksempel-oppgaven,

Du har gitt tallsekvensen $a_0, a_1, a_2, \dots, a_N$. Underveis i programmet kan du få to mulige forespørsler. Enten får du én forespørsel om å regne ut summen av tallene innenfor et gitt intervall $[i, j]$ altså $a_i + a_{i+1} + a_{i+2} + \dots + a_{j-1} + a_j$. Eller du får en forespørsel om å endre verdien av ett av tallene i arrayen **a**. For eksempel at verdien til a_i skal endres til 827. Lag så skal du lage et program som kan simulere disse operasjonene.

Den enkleste måten å løse denne oppgaven på er å ha en array som du lagrer verdiene i, og for hver forespørsel oppdaterer du enten arrayen eller kjører en

løkke som regner ut summen mellom indeksene i og j . Det å oppdatere arrayen skjer på konstanttid ($\mathcal{O}(1)$). Derimot det å regne ut summen kan i verste fall føre til $\approx N$ addisjoner. Dermed en kjøretid på $\mathcal{O}(N)$ tid! Hvis $N = 10,000,000$ og antall sum-forespørsler er 1,000 vil det å summere alle tallene ta alt for lang tid! Men vi kan gjøre ting smartere! Vi kan lage en ny array \mathbf{b} som ikke inneholder verdien til hvert element i \mathbf{a} , men heller prefix-summer. Så hvis vi slår opp på $b[i]$ får vi $a[0] + a[1] + \dots + a[i-1] + a[i]$. Hvis vi skal finne summen av alle elementene mellom indeksene i og j , der $i < j$, kan vi gjøre følgende regnestykke, $b[j] - b[i-1]$. Vi får først summen av alle elementene opp til og med indeks j , også trekker vi fra summen av alle elementene opp til (men ikke med) indeks i . Vi kan da finne summen av alle elementene fra indeks i til indeks j i konstant-tid (*big* $\mathcal{O}(1)$). Det dumme er at hvis man skal oppdatere et element, la oss si indeks i , må man oppdatere arrayen \mathbf{b} for alle indekser $[i, N-1]$, der N er antall elementer. Dette tilsvarer $\mathcal{O}(N)$ i kjøretid. Det blir nemlig alt for tregt, og da er vår metode ikke rask nok lenger. Legg også merke til at hvis elementene aldri blir oppdatert er det å lagre prefix-summer en god idé! Derimot skal vi ha muligheten for oppdateringer er Fenwick tre et godt valg. Fenwick tre regner ut prefix-summer på $\mathcal{O}(\log N)$ tid, og hvis vi skal regne ut summen over et intervall kan vi bruke 2 prefix-summer (som vist ovenfor $b[j] - b[i-1]$), og dermed blir kompleksiteten lik $\mathcal{O}(\log N)$. I et Fenwick tre kan vi også oppdatere et element på $\mathcal{O}(\log N)$ tid. Med disse kompleksitetene kan vi løse oppgaven ovenfor. Så hvordan er et Fenwick tre bygd opp? Som sagt kan vi løse oppgave ovenfor ved hjelp av et segment-tre, men fordelene med et Fenwick tre er at koden er kortere og dermed minker man sjansen for feil. I et Fenwick tre drar vi utnytte av det binære tallsystemet for å få en logaritmisk kjøretid, og det at Fenwick tre bruker bit-manipulasjon gjør at koden er rask! Før vi setter i gang med en forklaring av treet er det lurt å se hvordan man finner det minst signifikante bittet i et tall. Tenk deg vi har det binære tallet 10011010. Her er det minst signifikante bittet understreket. Det er nemlig det 1'er bittet som har minst verdi, ergo det 1'er bittet som er lengst til høyre. Vi kan finne dette bittet med å sjekke hvert bit fra høyre til venstre, men dette tar unødvendig lang tid. Derimot kan vi gjøre som koden under,

```
int LSONe(int n) { //minste signifikante 1'er bittet
    return n & (-n);
}

int N;
cin >> N;

cout << "minste sign. bit: " << LSONe(N) << endl;
```

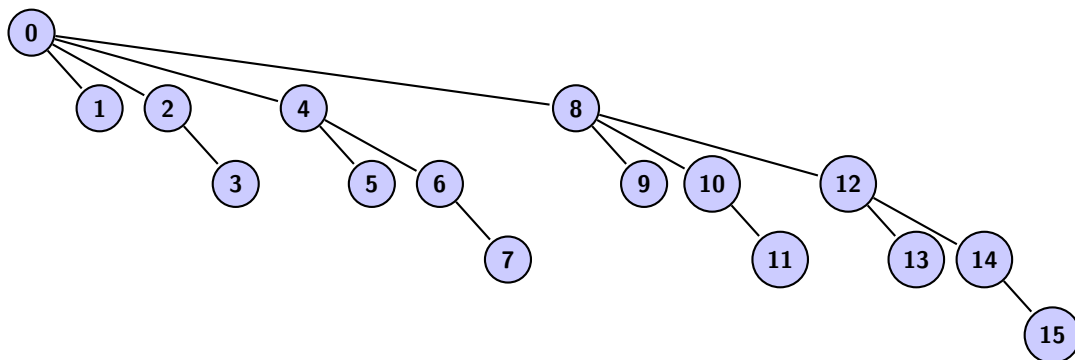
Hvis $N = 10011010$ (154 i titallsystemet) vil output bli 2 (verdi, ikke indeks). Siden etter vi har regnet ut $N \& (-N)$ står vi igjen med det binære tallet 00000010, som tilsvarer 2 i titallsystemet. Hvis vi vil fjerne det minst signifikante bittet fra $N = 10011010$, kan vi gjøre dette $N- = N \& (-N)$ vi står da igjen med $N = 10011000$. Nå er det minst signifikante bittet fjernet, siden vi trakk fra 2. I det Fenwick treet vi skal implementere lagrer vi alle verdiene i en array som heter \mathbf{ft} . Vi bruker ikke element nummer 0 ($\mathbf{ft}[0]$) på grunn av problemer med de binære operasjonene vi skal gjøre. Så vårt første element blir da $\mathbf{ft}[1]$. I den prefix-sum tabellen vi så ovenfor skal verdien av $\mathbf{b}[i]$ være $b[0] + b[1] + \dots + b[i]$ som vi forkorter med å skrive $[0 \dots i]$. I et Fenwick tre skal $\mathbf{ft}[i]$ være lik $[i - \text{LSONe}(i) + 1 \dots i]$, altså summen av alle elementene $a_{i-\text{LSONe}(i)+1} + \dots + a_i$. Det kan være vans-

kelig å se hvordan en slik ordning kan se ut, men under er en tabell som viser fordelingen for noen verdier av i ,

i	Binær repr. av i	Intervall $[i - \text{LSOne}(i) + 1 \dots i]$
1	1	[1...1]
2	10	[1...2]
3	11	[3...3]
4	100	[1...4]
5	101	[5...5]
6	110	[5...6]
7	111	[7...7]
8	1000	[1...8]
9	1001	[9...9]
10	1010	[9...10]

Med en slik fordeling av tallene kan vi regne ut prefix-summen av intervallet $[1 \dots i]$ slikt, $\text{ft}[i] + \text{ft}[i'] + \text{ft}[i''] + \dots$, helt til indeksen blir 0. Der vi har at $i' = i - \text{LSOne}(i)$, og $i'' = i' - \text{LSOne}(i')$ osv. Vi fjerner altså det minst signifikante bittet for hver gang. Siden antall bitt er $\log N$ vil kjøretiden bli $\mathcal{O}(\log N)$. La oss ta et eksempel. Si vi vil finne summen av alle elementene i intervallet $[1 \dots 6]$. Funksjonen som regner ut prefix-summer kaller vi **rsq** som står for “range sum query”. Vi starter da med å finne **rsq(6)**, funksjonen slår da opp i **ft[6]**, siden **ft[6]** har ansvaret for intervallet $[5 \dots 6]$ får vi at **ft[6]** = $a_5 + a_6$. Siden 6 i totallsystemet er 110 får vi ved å fjerne det minst signifikante bittet tallet 100 som er lik 4. Vi slår da opp på **ft[4]** og får at **ft[4]** = $a_1 + a_2 + a_3 + a_4$. Vi ser at hvis vi fjerner det minst signifikante bittet fra 4 som i totallsystemet er 100 får vi 0. Vi er dermed ferdig. Da står vi igjen med **ft[6]** + **ft[4]** som faktisk er summen av intervallet $[1 \dots 6]$, og vi er ferdig. Vi ser også at **ft[6]** og **ft[4]** ikke har noen overlapp av elementer. Siden **ft[6]** har ansvaret for $[6 - \text{LSOne}(6) + 1 \dots 6] = [5 \dots 6]$. Mens når vi skal til 4 fra 6 regner vi ut $6 - \text{LSOne}(6)$. Vi hopper nemlig til én mindre enn det intervallet til **ft[6]** dekker. Dermed får vi ingen overlapp, og vi dekker på denne måten alle elementene i intervallet $[1 \dots 6]$.

Nedenfor er en graf som viser hvilket tall vi kommer til dersom vi fjerner det minst signifikante bittet fra tallet. Vi kan se at hvis vi fjerner det minst signifikante bittet fra 10 får vi 8. Vi ser da at når vi skal regne ut prefix-summer er det dybden av dette treet som har noe å si for kjøretiden, og dybden er maksimal $\log N$.



Hvis vi ser på grafen ser vi at fra 8 kommer vi til 0. Det betyr at $ft[8]$ er summen av intervallet $[0 + 1 \dots 8]$.

Vi tar med ett til eksempel. Si vi skal regne ut $rsq(10)$. Vi ser først på $ft[10]$ som er summen av intervallet $[9 \dots 10]$ og legger dette til vår sum. Etterpå fjerner vi det minst signifikante bittet og får 8. Vi kjører så $rsq(8)$ og legger til $ft[8]$ til vår sum, dette er summen av intervallet $[1 \dots 8]$. Trekker vi fra det minste signifikante bittet i 8 får vi 0. Vi er dermed ferdig, siden vi ikke har et element på indeks 0. Summen av intervallet $[1 \dots 10]$ er da $ft[10] + ft[8]$.

Nå gjenstår det å kunne oppdatere et element på logaritmisk tid. Det vil si at hvis vi oppdaterer en indeks i må vi oppdatere alle $ft[i]$ verdiene der indeks i er inneholdt. Tenk deg at vi skal oppdatere verdien til element på indeks 2 til en verdi som er 10 mer. F.eks fra 14 til 24. Vi ser da at de intervallene som indeks 2 er inneholdt er $ft[2]$, $ft[4]$ og $ft[8]$. Hvis vi legger til 10 til alle disse ft -verdiene er vi ferdig. Så hvordan finner vi ut hvilke intervall vi skal oppdatere, og klare å oppdatere dem på logaritmisk tid? Vi kan faktisk gjøre det samme som vi gjorde med rsq , bare motsatt vei! Hvis vi skal øke verdien til element på indeks 2 med 10 kan dette gjøres slikt: Første intervall som inneholder indeks 2 er opplagt $ft[2]$. Denne øker vi med 10. Etterpå legger vi til det minst signifikante bittet i 2 til tallet 2. Vi får da tallet 4. Vi øker så verdien av $ft[4]$ med 10. Vi legger så det minst signifikante bittet i 4 til tallet 4 og får 8. Vi øker så verdien av $ft[8]$ med 10. Vi legger så til det minst signifikante bittet i 8 til tallet 8 og får 16. Siden $16 > N$ er vi ferdig med å oppdatere intervallene.

La oss ta ett til eksempel. La oss trekke fra 5 til element på indeks 3. Vi går så først til $ft[3]$ og trekker fra 5. Siden det minst signifikante bittet i 3 er 1 vil vi etter å ha lagt til det minst signifikante bittet i 3 med tallet 3 få 4. Vi oppdaterer så $ft[4]$ med å trekke fra 5. Vi legger så det minst signifikante bittet i 4 til tallet 4 og får 8. Vi minker så verdien av $ft[8]$ med 5. Vi legger så til det minst signifikante bittet i 8 til tallet 8 og får 16. Siden $16 > N$ er vi ferdige med å oppdatere intervallene. Siden vi maks kan legge til $\log N$ bitt til vår startindeks før tallet er større enn N har oppdateringen en kompleksitet lik $\mathcal{O}(\log N)$. Under er koden for Fenwick tre implementasjonen,

```
#define LSOne(a) (a & (-a))

class FenwickTree {
private:
    vector<int> ft;
```

```

public:
    FenwickTree (int n) {    ft.assign(n + 1, 0);    }

    //[1 ... i]
    int rsq(int i) {
        int sum = 0;
        for (; i > 0; i -= LSOne(i))
            sum += ft[i];
        return sum;
    }

    //[i ... j]
    int rsq(int i, int j) {
        return rsq(j) - (i == 1 ? 0 : rsq(i - 1));
    }

    //endrer verdien av element nr. k med aa oেকে den med v
    //v kan vaere negativ for aa minke verdien
    void update(int k, int v) {
        for (; k < ft.size(); k += LSOne(k))
            ft[k] += v;
    }
};

int main() {
    // idx   0 1 2 3 4 5 6 7  8 9 10,
    // ingen indeks 0!
    FenwickTree ft(10); // ft = {-,0,0,0,0,0,0,0,0, 0,0,0}
    ft.update(2, 1); // ft = {-,0,1,0,1,0,0,0,0, 1,0,0}, idx
    // 2,4,8 => +1
    ft.update(4, 1); // ft = {-,0,1,0,2,0,0,0,0, 2,0,0}, idx
    // 4,8 => +1
    ft.update(5, 2); // ft = {-,0,1,0,2,2,2,0,0, 4,0,0}, idx
    // 5,6,8 => +2
    ft.update(6, 3); // ft = {-,0,1,0,2,2,5,0,0, 7,0,0}, idx
    // 6,8 => +3
    ft.update(7, 2); // ft = {-,0,1,0,2,2,5,2,0, 9,0,0}, idx
    // 7,8 => +2
    ft.update(8, 1); // ft = {-,0,1,0,2,2,5,2,10,0,0}, idx
    // 8 => +1
    ft.update(9, 1); // ft = {-,0,1,0,2,2,5,2,10,1,1}, idx
    // 9,10 => +1
    printf("%d\n", ft.rsq(1, 1)); // 0 => ft[1] = 0
    printf("%d\n", ft.rsq(1, 2)); // 1 => ft[2] = 1
    printf("%d\n", ft.rsq(1, 6)); // 7 => ft[6] + ft[4] = 5 + 2 =
    // 7
    printf("%d\n", ft.rsq(1, 10)); // 11 => ft[10] + ft[8] = 1 +
    // 10 = 11
    printf("%d\n", ft.rsq(3, 6)); // 6 => rsq(1, 6) - rsq(1, 2) =
    // 7 - 1

    ft.update(5, 2); // oppdater demo
    printf("%d\n", ft.rsq(1, 10)); // naa 13
}

```

5.4.1 Fenwick tree i 2-dimensjoner

I noen oppgaver er det nyttig å kunne regne ut summen av en tabell i 2-dimensjoner. En måte er å bruke en lignende prefix-sum tabell, der $v[i][j]$ betyr summen av alle elementene som er innenfor rektangelet som har øvre venstre hjørne lik $(0,0)$ og med nedre høyre hjørne lik (j,i) . På denne måten kan man

regne ut summen av slike rektangler på konstant-tid. Koden under viser hvordan vi kan regne ut slike summer av en gitt tabell **a**.

```
int main() {
    //N er antall rader, M er antall kolonner
    int N, M;
    cin >> N >> M;

    vector< vector<int> > a(N, vector<int> (M));
    for (int i = 0; i < N; i++)
        for (int j = 0; j < M; j++)
            cin >> a[i][j];

    vector< vector<int> > v(N, vector<int> (M));
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            v[i][j] = a[i][j];
            if (i > 0) v[i][j] += v[i - 1][j];
            if (j > 0) v[i][j] += v[i][j-1];
            if (i>0 && j>0) v[i][j] -= v[i-1][j-1];
        }
    }

    int x1, y1, x2, y2;
    cout << "Fra linje, til linje, fra kolonne, til kolonne: ";
    cin >> y1 >> y2 >> x1 >> x2;

    int sum = v[y2][x2];
    if (x1 > 0) sum -= v[y2][x1-1];
    if (y1 > 0) sum -= v[y1-1][x2];
    if (x1>0 && y1>0) sum += v[y1-1][x1-1];

    cout << "Sum: " << sum;
}

```

Med denne teknikken kan man f.eks. løse denne øvingsoppgaven fra **IOI, Taiwan, 2014** <http://www.ioi2014.org/images/competition/1-square.pdf>, siden her skjer det ingen endringer, og man kan finne størrelsen på kvadratet ved hjelp av binærsøk. Problemet med en slik prefix-sum tabell er som i 1-dimensjoner, nemlig at det å oppdatere tabellen tar lang tid. Her tar det nemlig $\mathcal{O}(N * M)$ tid for oppdatering, men $\mathcal{O}(1)$ for sum. Med Fenwick tre i 2 dimensjoner kan vi få sum og oppdatering til å kjøre i $\mathcal{O}(\log N * \log M)$ tid. Tenk deg denne oppgaven **Mobiles, IOI 2001** (<http://www.ioinformatics.org/locations/ioi01/contest/day1/mobiles/mobiles.pdf>) her har vi en todimensjonal array der alle verdiene er satt til 0. Etterhvert kan du få beskjed om at en bestemt element i arrayen kan øke eller minke med en gitt verdi. Du kan også få forespørsler om å regne ut summen av alle verdiene innenfor et gitt rektangel av arrayen. Siden vi jobber med summer kan vi bruke et Fenwick tre, og siden vi har en todimensjonal array kan vi bruke Fenwick i 2D. Koden er ikke veldig forskjellig fra den for én dimensjon, bare at vi nå må bruke en todimensjonal array for **ft**. Under er koden og har samme idé som Fenwick i 1-dimensjon,

```
class FenwickTree {
private:
    int N, M; //N=ant rader, M=ant kolonner
    vector< vector<int> > ft; //rad, kolonne

public:

```

```

FenwickTree (int n, int m) {
    N = n+1;
    M = m+1;
    ft.assign(N, vector<int> (M, 0));
}

//[1, 1] til [y, x]
int rsq(int x, int y) {
    int sum = 0;
    for (; y > 0; y -= LSOne(y))
        for (int j = x; j > 0; j -= LSOne(j))
            sum += ft[y][j];
    return sum;
}

//[y1, x1] til [y2, x2]
int rsq(int x1, int y1, int x2, int y2) {
    return rsq(x2, y2) - rsq(x2, y1-1) - rsq(x1-1, y2) + rsq
        (x1-1, y1-1);
}

//endrer verdien av element [y][x] med aa oeke den med v
//v kan vaere negativ for aa minke verdien
void update(int x, int y, int v) {
    for (; y < N; y += LSOne(y))
        for (int j = x; j < M; j += LSOne(j))
            ft[y][j] += v;
}
};

```

Nå kan vi teste ut datastrukturen på **Mobiles** oppgaven fra IOI 2001. Dette kan du teste på denne nettsiden: <http://wcipeg.com/problem/ioi0111>. Hvis du er interessert i å kode løsningen selv trenger du ikke å se på løsningen nedenfor,

```

int main() {
    int cmd, N;
    cin >> cmd >> N;

    int X, Y, A, L, B, R, T;
    FenwickTree FT(N, N);
    while (true) {
        cin >> cmd;
        switch (cmd) {
            case 1:
                cin >> X >> Y >> A;
                FT.update(X+1, Y+1, A); //1-indeksert
                break;
            case 2:
                cin >> L >> B >> R >> T;
                cout << FT.rsq(L+1, B+1, R+1, T+1) << endl;
                break;
            case 3:
                return 0;
                break;
        }
    }
}

```

5.5 Oppgaver

Union-Find:

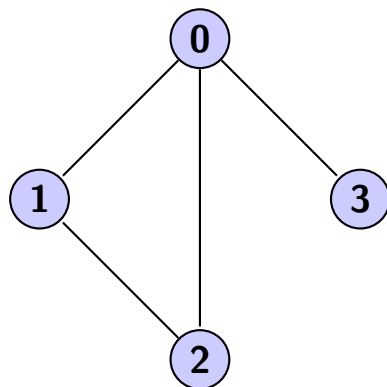
ID	Navn
793	Network Connections (*)
1197	The Suspects
10227	Forests
10507	Waking up brain (*)
10583	Ubiquitous Religions
10608	Friends
10685	Nature
11503	Virtual Friends (*)
11690	Money Matters

Trær:

ID	Navn
536	Tree Recovery (*)
10459	The Tree Root (*)
12347	Binary Search Tree

6 Grafer

Vi skal nå se på grafer. Vi skal ikke snakke om funksjonsgrafer fra matematikk, men heller om relasjonsgrafer. Si du har et vennenettverk, vi kan da bruke en graf for å vise hvem som er venn med hvem.

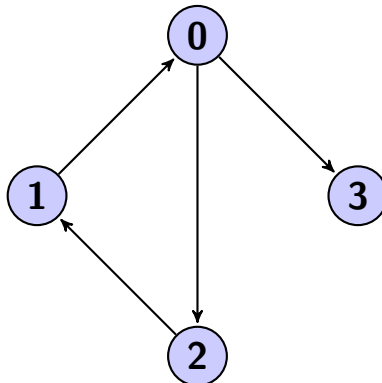


Vi kan for eksempel tolke grafen ovenfor som at person **0** er venn med alle (og alle er venn med 0). I tillegg er person **1** og person **2** også venner. Det er mange problemer som kan omskrives til å bli et grafproblem, men da trenger vi noen verktøy for å angripe dem.

Først litt notasjon.

6.1 Notasjon

Av figuren ovenfor sier vi at sirklene er **noder**. Nodene kan være forbundet/(ha en relasjon) med hverandre, og uttrykkes som oftest med en linje mellom nodene som er forbundet. Disse linjene kalles **kanter**. To **noder** kalles **nabonoder** hvis det går en kant mellom dem. I figuren ovenfor er **2** og **1** nabonoder, men det er ikke **3** og **1**. Ut i fra figuren ovenfor sier vi at node **0** har **3 kanter**. Node **3** har kun én kant. Antall noder i grafen skrives gjerne som V , og antall kanter i grafen skrives som E . Grafen ovenfor har $V = 4$ og $E = 4$. En graf kalles fullstendig / komplett hvis det finnes en kant mellom hvert par av noder. Dette impliserer at $E = V * (V - 1) / 2$ (husk at kantene går begge veier). En graf sies å være **connected** hvis for hver par av noder finnes det en **sti** av kanter som kan føre oss mellom de to nodene (nødvendigvis ikke direkte). Grafen ovenfor er connected siden vi kan vandre ved hjelp av kantene mellom hvert par av noder. Hvis vi summerer opp antall kanter hver node har, som i grafen ovenfor blir $3 + 2 + 2 + 1 = 8$, får vi $2E$. Grunnen til dette er at hver kant går mellom nøyaktig 2 noder. Hvis vi har en kant mellom nodene **a** og **b** vil denne kanten bli telt dobbelt når vi summerer opp antall kanter hver node har. Dette gjelder kun for urettet graf som ovenfor. Vi har også en annen type graf som vi kaller rettet graf. Det er grafer der hver kant har en retning som figuren nedenfor.



Når vi snakker om et venne-nettverk er det unaturlig å bruke en rettet graf. Derimot skal vi forklare et T-bane-nettverk kan det være greit å bruke en rettet graf, der nodene er stasjoner, kantene er tog-skinner og retningen viser kjøretretning. I stedet for å si at node **0** har 3 kanter vil vi heller si at node **0** har 2 utgående kanter, og 1 inngående kant. I figuren er node **1** en nabonode for node **2**, men ikke motsatt pga. retningen.

6.2 Representasjon av grafer

Først må vi ha en metode for hvordan en slik graf skal representeres i datamaskin. Vi har noen standardmåter,

6.2.1 Nabomatrise

Vi kan lagre en 2-dimensjonal array som vist under,

	0	1	2	3
0	F	T	T	T
1	T	F	T	F
2	T	T	F	F
3	T	F	F	F

Som skal representere grafen vi så på ovenfor. T står for “true” og F står for “false”. T betyr at de er venner, F betyr ikke venner. Så hvis vi slår opp på rad 0 og kolonne 1 står det T fordi person 0 og 1 er venner. Selvfølgelig er det slikt at hvis vi slår opp på rad 1 og kolonne 0 står det også T (de er venner begge veier, urettet graf). Dermed er altså tabellen symmetrisk om diagonalen. Vi kan kode dette slikt,

```
int N;
cout << "Antall personer: ";
cin >> N;

vector< vector<bool> > mat(N, vector<bool> (N, false));

int M;
cout << "Antall vennepar: ";
cin >> M;

int a, b;
while (M--) { //tilsvarer for (int i=0; i<M; i++), M false hvis
  og bare hvis M=0
  cout << "Skriv indeksene til venneparet: ";
  cin >> a >> b;
  mat[a][b] = mat[b][a] = true;
}
```

Vi har da bygget array slik som beskrevet ovenfor. Legg merke til at siden det å være venner går begge veier har vi satt **mat[a][b]** og **mat[b][a]** like true. Vi kunne også bare satt **mat[a][b]** lik true. Da kunne vi f.eks. vise "hvilke personer som vil bli venn med hvem", der **mat[a][b]=true** betyr at **a** vil bli venn med **b**, men kanskje ikke **b** vil bli venn med **a**.

Vi har også en annen og mer vanlig måte å representere en graf på,

6.2.2 Naboliste

I stedet for å lage en array med alle par-kombinasjonene (med true/false), kan vi lagre kun de som er venner. Vi demonstrerer teknikken på grafen ovenfor,

```
0 : 1, 2, 3
1 : 0, 2
2 : 0, 1
3 : 0
```

Der hver person har en liste hver om hvilke personer de er venn med. Dette kan kodes slikt,

```
int N;
cout << "Antall personer: ";
```

```

cin >> N;

vector< vector<int> > edge(N);

int M;
cout << "Antall vennepar: ";
cin >> M;

int a, b;
while (M--) { //tilsvarer for (int i=0; i<M; i++), M false hvis
    og bare hvis M=0
    cout << "Skriv indeksene til venneparet: ";
    cin >> a >> b;
    edge[a].push_back(b);
    edge[b].push_back(a);
}

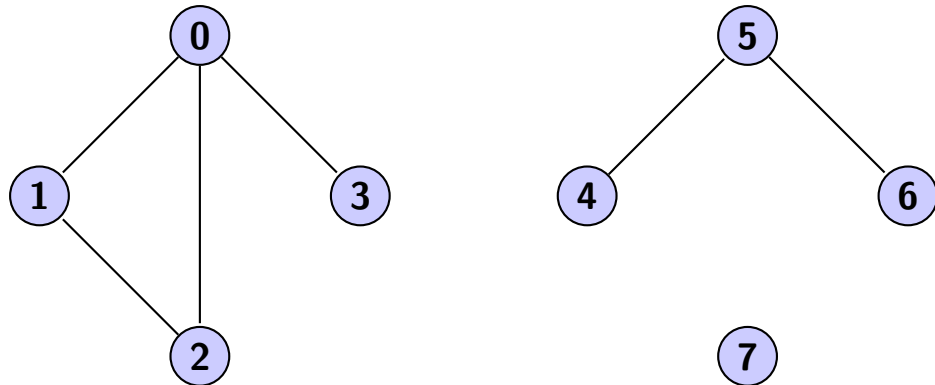
```

Vi sier at **a** er venn med **b**, og at **b** er venn med **a**.

Skal vi velge nabomatrix eller naboliste til å representere grafer? Nabomatrix bruker du dersom du må hele tiden slå opp om **a** har en kant til **b**. Siden dette kan skje på konstanttid (slå opp i $\text{mat}[\mathbf{a}][\mathbf{b}]$). Derimot hvis du skal hele tiden finne alle kantene til noden **a** er det lure å bruke naboliste. Tenk deg at det er titusen noder, og **a** har kun 100 kanter. Vi kan da finne alle kantene til **a** med å bla igjennom en liste på 100 elementer istedet for å bla igjen $\text{mat}[\mathbf{a}][\mathbf{i}]$ for $i \in [0, 9999]$ noe som er betydelig tregere.

6.3 Dybde først søk (DFS)

Vi skal nå se på en teknikk som heter dybde først søk. Det er en teknikk for å kunne gjennomføre et grafsøk. Si vi har grafen nedenfor som representere et vennenettverk,



Vi er så ute etter å lage en algoritme der vi oppgir en person og vil finne alle vennene til den personen, og alle venners venner til den personen, og alle venners venners venner til den personen osv. (algoritmen skal også inkludere første personen). Så algoritmen lager en personliste over hele vennenettverket som den gitte personen er med i. Så hvis vi kjører den algoritmen på person nummer **3** er vi ute etter å få en liste (0, 1, 2, 3). Siden **3** er venn med **0** som igjen er venn med **1** og **2**, og dette utgjør hele vennenettverket til person **3**. Kjører vi algoritmen på person **7** vil vi kun få listen (7). Person **7** er den eneste

personen i sitt vennenettverk. For å lage en slik algoritme bruker vi dybde først søk, se på denne video for en forklaring av algoritmen <https://www.youtube.com/watch?v=zLZhSSXAwXI> (det er kun første halvdel som omhandler dybde først søk). Vi kan programmere denne algoritmen ved å bruke en **stack**. Her er koden (vi bruker naboliste),

```

int N;
cout << "Antall personer: ";
cin >> N;

vector< vector<int> > edge(N);

int M;
cout << "Antall vennepar: ";
cin >> M;

int a, b;
while (M-->) { //tilsvareer for (int i=0; i<M; i++), M false hvis
    og bare hvis M=0
    cout << "Skriv indeksene til venneparet: ";
    cin >> a >> b;
    edge[a].push_back(b);
    edge[b].push_back(a);
}

int p;
cout << "Skriv nummeret til personen: ";
cin >> p;

vector<int> nettverk;
vector<bool> visited(N, false);
stack<int> stc;
stc.push(p);

while (!stc.empty()) {
    int person=stc.top();
    stc.pop();

    if (visited[person])
        continue;

    visited[person]=true;

    nettverk.push_back(person);
    for (int i = 0; i < edge[person].size(); i++)
        stc.push(edge[person][i]);
}

for (int i = 0; i < nettverk.size(); i++)
    cout << nettverk[i] << ", ";

```

Prøv å kjøre programmet med grafen ovenfor og se at den fungerer.

Det som gjør dybde først søk (DFS) enklere å kode enn bredde først søk (BFS) (som vi skal se på senere) er at DFS kan enkelt programmeres rekursivt! Vi kan lage en funksjon som gjør akkurat det samme som ovenfor (men vi slipper å bruke datastrukturen **stack**),

```

vector< vector<int> > edge(N);
vector<int> nettverk;
vector<bool> visited(N, false);

void DFS(int person) {

```

```

    if (visited[person])
        return;

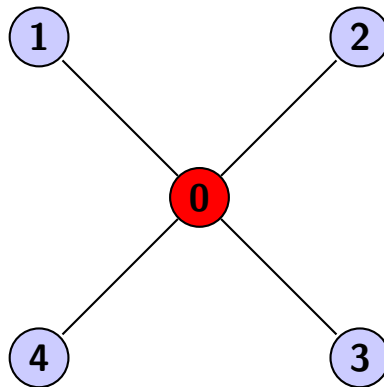
    visited[person] = true;
    nettverk.push_back(person);
    for (int i = 0; i < edge[person].size(); i++)
        DFS(edge[person][i]);
}

```

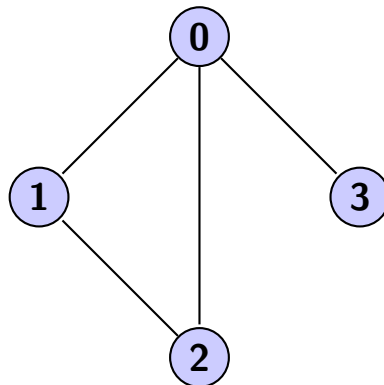
Kjøretiden for begge tilfellene er $\mathcal{O}(E)$ der E er antall kanter i grafen. Dette er fordi vi kun “går” over en kant kun én gang.

6.3.1 Bicoloring

Bicoloring er et problem som ofte oppstår blant programmeringsoppgaver. Det er at man skal fargelegge alle nodene i en graf, med 2 forskjellige farger, slik at ingen naboener har samme farge. Grafen nedenfor er en slik fargelagt graf (med fargene blå og rød),



Derimot kan ikke grafen nedenfor fargelegges slikt,



Siden hvis vi fargelegger node 0 rød, må både node 1, 2 og 3 fargelegges blå. Siden node 1 og 2 er naboener og har samme farge kan vi ikke fargelegge grafen med to farger slik at ingen naboener har samme farge.

Vi skal nå se på oppgaven Julegaver fra 2. runde 2013/2014 som kan løses ved hjelp av bicoloring.

Nils synes han brukte altfor mye tid på å handle julegaver i fjor, så i år vil han gjøre det enklere. Han har allerede søkt litt på nettet, og funnet to flotte gaver han kan gi bort (et rødt skjerf og en bok om programmering). Fordi disse gavene er så fine og fordi han vil spare tid ønsker han kun å gi bort slike skjerf og bøker. Problemet er selvfølgelig at folk blir sure hvis de oppdager at noen de kjenner har fått samme gave fra han. Så derfor kan han ikke gi samme gave til to personer som kjenner hverandre. Samtidig ønsker Nils at så mange som mulig skal lære seg programmering og derfor ønsker han at flest mulig skal få programmeringsboka. Hjelp Nils finne ut hvor mange programmeringsbøker han kan gi bort, uten at to bekjente får samme gave, hvis han deler ut gavene smartest mulig til alle han kjenner. Han må gi gave til absolutt alle han kjenner.

Input

Første linje inneholder heltallet N , antall personer Nils kjenner. Deretter følger N linjer nummerert fra 0 til $N - 1$. Linje nummer i forteller hvem person nummer i kjenner. Hver linje starter med heltallet K , antall bekjente til person nummer i . Deretter kommer K heltall som er nummeret til personer som person nummer i kjenner. Merk: Hvis person x kjenner person y , kjenner nødvendigvis person y person x .

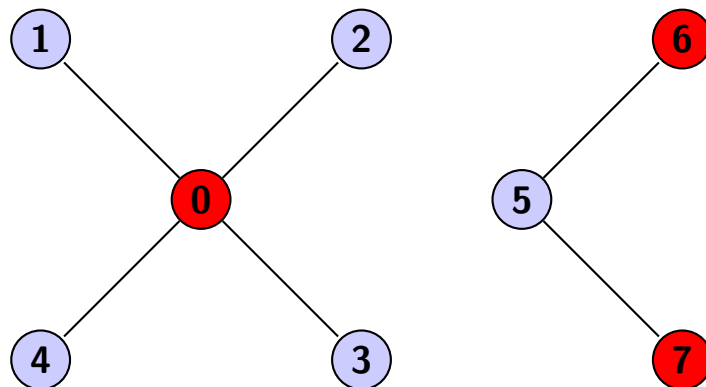
$$1 \leq N \leq 100000$$

$$1 \leq M \leq 100000 \quad \text{der } M \text{ er antall bekjentsskaper}$$

Output

Du skal kun skrive ut ett heltall: Det maksimale antallet programmeringsbøker Nils kan gi bort som gave til jul uten at to personer som kjenner hverandre får samme gave. Skriv ut 0 hvis det ikke lar seg gjøre å dele ut gaver til alle han kjenner.

Hvis vi lager et vennettverk ut fra inputten, ser vi at problemet spør oss om å fargelegge denne grafen som beskrevet ovenfor. Vi må også finne ut om det i det hele tatt er mulig å fargelegge den slikt. Vi må også passe på at inputten kan gi oss flere uavhengige vennettverk! Så vi må fargelegge alle disse nettverkene hver for seg. Siden vi vil maksimalisere antall programmeringsbøker må vi se over hvert eneste vennettverk og finne hvilken farge (rød / blå) som det er flest av i nettverket. Det er nemlig nodene med denne fargen som skal ha programmeringsbøker!



Av grafen ovenfor ser vi at vennetnettverket til venstre burde vi gi de blå nodene programmeringsbøker, mens i nettverket til høyre burde vi gi de røde nodene programmeringsbøker.

Så hvordan lager vi en algoritme som fargelegger en graf med kun 2 farger? Først så finner vi en node, la oss kalle den **a**, som ikke er fargelagt. Vi fargelegger denne noden rød. Vi sjekker så om noen nabonoder til **a** også er rød, i så fall kan vi ikke fargelegge grafen med to farger (kollisjon). Hvis ingen av nabonodene til **a** er røde fargelegger vi dem blå. Vi går så til disse nabonodene og fargelegger deres nabonoder røde, finner vi en kollisjon vet vi at det ikke går med kun 2 farger. Vi fortsetter så videre med deres nabonoder osv. Her er en løsning til Julegaver,

```

int N, K, tmp, num[3]={0,0,0}, sum=0; //num holder styr paa
    antall noder som er ufarget, roed, blaa
vector<int> edge[100000]; //en array av vector<int>
char col[100000]; //0=ufarget, 1=roed, 2=blaa

bool dfs(int p, int c) { //p er noden,
    //c er fargen vi skal fargelegge med
    //dfs gir tilbake true hvis nettverket
    kunne fargelegges med 2 farger
    //false hvis det ikke gikk

    if (col[p]!=0 && col[p]!=c) return false;
    if (col[p]==c) return true; //den er allerede
    farget, hopp videre
    col[p]=c;
    num[c]++; //antall noder med fargen c oekes
    int rev=(c==1)?2:1; //rev=motsatt farge, roed->blaa, blaa->
    roed
    for (int i = 0; i < edge[p].size(); i++)
        if (!dfs(edge[p][i], rev))
            return false;
    return true;
}

int main() {
    memset(col, 0, sizeof col); //setter alle col[i] lik 0.
    cin >> N;
    for (int i = 0; i < N; i++) {
        cin >> K;
        for (int j = 0; j < K; j++)
            cin >> tmp, edge[i].push_back(tmp); //2 linjer i
            1!!!
    }
}

```

```

for (int i = 0; i < N; i++)
    if (col[i] == 0) //ufarget
        if (!dfs(i, 1)) { //kunne ikke fargelegge nettverket
            printf("0\n");
            return 0;
        } else
            sum+=max(num[1], num[2]), num[1]=num[2]=0; //2
            linjer i 1! vi legger til den fargen det var
            mest av i nettverket vi akkurat har
            fargelagt til i summen, og saa setter vi
            antall noder som er roed og blaa til 0.

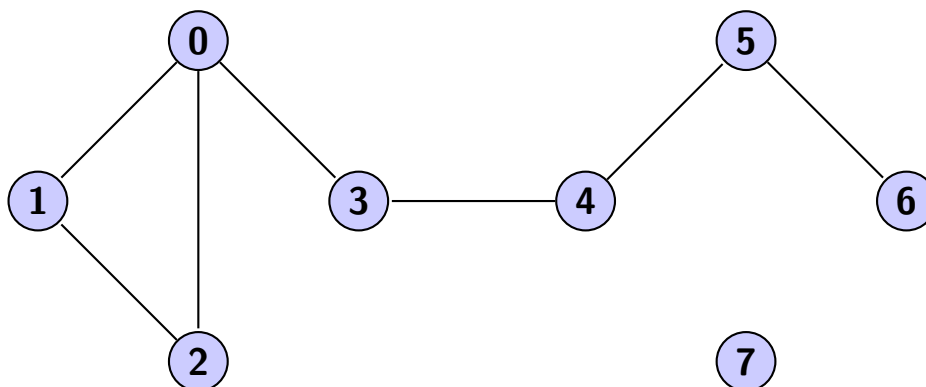
printf("%d\n", sum);
return 0;
}

```

6.4 Bredde først søk (BFS)

Bredde først søk (BFS) er en annen måte å grafsøke på som ligner veldig på DFS. Grunnen til navnene dybde og bredde er at hvis vi kjører DFS på en node vil den begynne å søke helt ned i dybden på nabonodene før den går videre til neste nabonode. Derimot vil BFS gå i bredden først, altså undersøke alle nabonodene før den undersøker nabonodenes nabonoder. Se siste halvdel av denne videoen for en forklaring av BFS algoritmen <https://www.youtube.com/watch?v=zLZhSSXAwXI>.

Det finnes en teori som heter "Six degrees of separation" (http://en.wikipedia.org/wiki/Six_degrees_of_separation) om at alle kjenner alle gjennom maks 6 venneledd. Altså at du gjennom maks 6 ledd vil f.eks. kjenne Obama. Tenk deg at du kjenner en venn som er med i AUF, denne personen kjenner kanskje en annen person i AUF som igjen kjenner Jens Stoltenberg. Jens Stoltenberg derimot kan man si kjenner Obama. Da er du kun 4 ledd unna å kjenne Obama, og innenfor teorien. Med BFS kan vi virkelig sjekke om denne teorien stemmer hvis vi hadde informasjon om alle som er venner i verden. Derimot skal vi heller fokusere på å finne en algoritme for å se hvor mange venneledd to personer er unna hverandre. Se på grafen nedenfor,



Vi kan lage en algoritme der vi kan finne ut hvor mange ledd to personer er fra hverandre. F.eks. vil person 2 og 0 være kun 1 ledd unna hverandre. Derimot

vil person **1** og person **6** være hele 5 ledd unna hverandre. Og mellom person **7** og **5** finnes det ingen slike ledd vi kan følge. Implementasjonen ligner veldig på DFS, bortsett fra at vi bytter ut **stack** med **queue**,

```

int N;
cout << "Antall personer: ";
cin >> N;

vector< vector<int> > edge(N);

int M;
cout << "Antall vennepar: ";
cin >> M;

int a, b;
while (M--> { //tilsvarer for (int i=0; i<M; i++), M false hvis
    og bare hvis M=0
    cout << "Skriv indeksene til venneparet: ";
    cin >> a >> b;
    edge[a].push_back(b);
    edge[b].push_back(a);
}

int A, B;
cout << "Skriv inn personene du vil vite om: ";
cin >> A >> B;

vector<bool> visited(N, false);
queue< pair<int, int> > que; //foerste parameter er person,
    andre er antall ledd fra person A
que.push( make_pair(A, 0) );
visited[A] = true;

while (!que.empty()) {
    pair<int, int> par=que.front();
    que.pop();

    int person = par.first;
    int dist = par.second;

    if (person == B) {
        cout << "Antall ledd mellom " << A << " og " << B << "
            er: " << dist << endl;
        return 0;
    }

    for (int i = 0; i < edge[person].size(); i++)
        if (!visited[ edge[person][i] ]) {
            que.push(make_pair(edge[person][i], dist+1));
            visited[ edge[person][i] ] = true;
        }
}

cout << "Det finnes ingen vennesti mellom " << A << " og " << B
    << endl;

```

Prøv å kjøre programmet på grafen ovenfor, og mellom person **2** og **3**. Det burde bli 2 ledd (gjennom 0) og ikke 3 ledd (gjennom 1). Tenk på hvorfor vi alltid får den korteste ruten mellom personene!

Siden BFS går kun gjennom en kant kun én gang (som DFS) har den også en kompleksitet lik $\mathcal{O}(E)$, der E er antall kanter i grafen.

Hvis du har spilt en del sjakk har du kanskje noen ganger hatt en springer på sjakkbrettet og lurt på hva er det minste antall flytt du trenger for å flytte springeren fra det feltet det nå står på til et annet gitt felt. Vi kan også bruke bredde først søk for å løse denne oppgaven. Tenk på sjakkbrettet som en graf med 64 noder, en node per felt. Men hvordan skal kantene være? Hvem skal være nabonoder? I dette problemet er det naturlig at to noder er nabonoder hvis og bare hvis en springer kan flytte fra det ene feltet til det andre feltet på ett trekk. Så en springer kan kun flytte til nabonodene til den noden den står på. Vi kan da kjøre et BFS fra den noden den står på til den noden vi vil nå. Man skulle tro man trenger å først lage hele grafen også lage kantene i grafen før man gjør et BFS, men vi kan faktisk lage grafen etterhvert som vi gjør BFS. Når vi står på et felt er det ubrukelig å vite om alle nabonodene til en node vi ikke står på. Hver gang vi er på en node kan vi regne/finne ut alle nabonodene til noden vi står på, vi trenger ikke å lagre dem. Hvis vi nummererer radene i sjakkbrettet fra 0 til 7 og kolonnene fra 0 til 7 ser koden slikt ut,

```

struct sState {
    pair<int, int> pos; //rad, kol
    int dist;
    sState (pair<int, int> _pos, int _dist) {
        pos = _pos;
        dist = _dist;
    }
};

int dx[] = {1, 2, 2, 1, -1, -2, -2, -1};
int dy[] = {-2, -1, 1, 2, 2, 1, -1, -2};
//hvordan en springer flytter, dx[i] og dy[i]
//er differanse i x og y retning naar springeren flytter,
//og vi har 8 mulige flytt

int main () {
    pair<int, int> start, slutt;

    cout << "Fra felt (rad, kolonne): ";
    cin >> start.first >> start.second;

    cout << "Til felt (rad, kolonne): ";
    cin >> slutt.first >> slutt.second;

    bool visited[8][8];
    memset(visited, 0, sizeof visited); //0 = false

    queue<sState> que;
    que.push(sState(start, 0));
    visited[start.first][start.second] = true;

    while (!que.empty()) {
        sState par = que.front();
        que.pop();

        if (par.pos == slutt) {
            cout << "Antall steg: " << par.dist;
            break;
        }
    }

    int nxtRad, nxtKol;
    for (int i = 0; i < 8; i++) {
        nxtRad = par.pos.first + dy[i];
    }
}

```

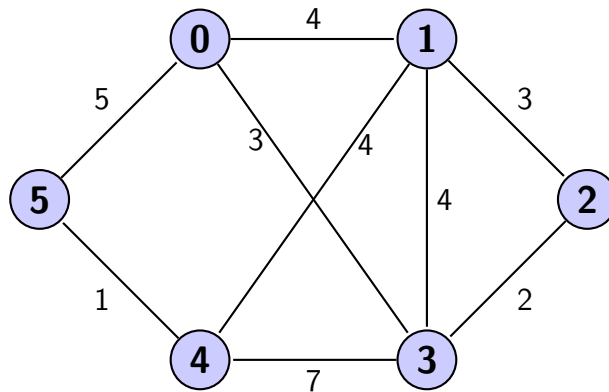
```

nxtKol = par.pos.second + dx[i];
if (nxtRad >= 0 && nxtRad < 8 &&
    nxtKol >= 0 && nxtKol < 8 &&
    !visited[nxtRad][nxtKol])
{
    que.push(sState(make_pair(nxtRad, nxtKol), par.
        dist+1));
    visited[nxtRad][nxtKol] = true;
}
}
}
}

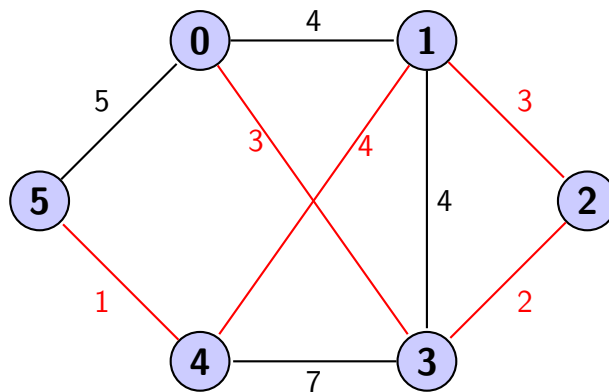
```

6.5 Minimum spanning tree

Før vi begynner på minimum spanning tree eller på norsk “minste utspent tre” skal vi se på en såkalt vektet graf som vi skal jobbe mye med, se grafen nedenfor,



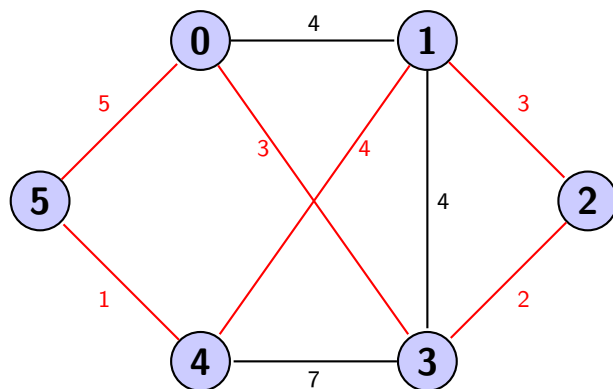
Tenk deg at nodene i grafen er byer, kantene er potensielle veier vi kan bygge der tallene er hvor mye veien koster. Vi har lyst til å bygge veier slikt at vi kan reise mellom alle par av byer for billigst mulig penge. Vi kan selvfølgelig bygge alle veiene, men det blir unødvendig dyrt. Faktisk er den optimale måten å bygge ut veinettet på tegnet inn i grafen under, der rød betyr at vi skal bygge veien,



Under har jeg skrevet ned to forskjellige algoritmer som løser dette problemet.

6.5.1 Kruskal algoritme

Kruskal er kanskje den rensligste måten å implementere minimum spanning tree på. For en forklaring av algoritmen sjekk wikipedia og denne videoen <https://www.youtube.com/watch?v=T43K2qt0IiY>. Kruskals algoritme har en kompleksitet på $\mathcal{O}(E \log V)$ der E er antall kanter i grafen og V er antall noder. Vi lagrer først alle potensielle kanter i en liste. Så sorterer vi listen med hensyn på vektallet (lengden). Vi går starten med kanten med det minste vektallet og legger den til å listen over kanter vi vil bruke. Deretter legger vi til nest minste kanten til listen osv. Men vi må passe på å ikke legge til unødvendige kanter, nemlig de kantene som vil forme en sykel. Hvis vi lar være å legge til kanter som former en sykel vil vi til slutt ende opp med et tre, derfor navnet minimum spanning tree. Det at vi danner en sykel er at vi får en graf som nedenfor,



Der vi nå har lagt til en rød strek mellom node 0 og 5. Denne kanten er unødvendig siden vi nå har laget en sykel (vi kan følge den røde streken rundt og rundt). Den siste kanten mellom 0 og 5 gjør det mulig å reise fra node 5 til 0 og motsatt, men dette kunne vi allerede fra før av ved hjelp av stien (5 – 4 – 1 – 2 – 3 – 0). Det at det ikke finnes en sykel i grafen er ekvivalent med at det er en entydig sti mellom alle par av noder. For å unngå å lage en sykel bruker man Union-Find datastrukturen vi skrev i “Datastrukturer” kapittelet. Vi starter med at alle nodene er sin egen gruppe. Etterhvert som vi legger til en kant mellom node **a** og **b** slår vi gruppene til **a** og **b** sammen. Vi danner en sykel når vi legger til en kant mellom to noder som allerede er i samme gruppe. Så lenge vi passer på å ikke legge til kanter som former en sykel er vi i mål! Under er en kode som tar inn en graf og som skriver ut kantene som former det minste utspente treet i grafen (har du aldri sett **operator** før så se <http://stackoverflow.com/questions/4892680/sorting-a-vector-of-structs>),

```
struct sEdge {
    int u, v; //kant mellom u og v
    int weight; //vektallet
    sEdge () {}
    bool operator< (const sEdge &other) const { //sortere med
        hensyn paa vektallet
        return weight < other.weight;
    }
};
```

```

    }
};

int main () {
    int N, M; //N=ant. noder, M=ant. kanter
    cin >> N >> M;

    vector<sEdge> edge(M);

    for (int i = 0; i < M; i++)
        cin >> edge[i].u >> edge[i].v >> edge[i].weight;

    sort(edge.begin(), edge.end());

    UnionFind UF(N);
    vector<sEdge> minSpanTree;
    int sum = 0;

    for (int i = 0; i < edge.size(); i++) {
        if (!UF.isSameSet(edge[i].u, edge[i].v)) { //vil ikke
            danne en sykel
            UF.unionSet(edge[i].u, edge[i].v);
            minSpanTree.push_back(edge[i]);
            sum += edge[i].weight;
        }
    }

    cout << "Minimum spanning tree:\n";
    for (int i = 0; i < minSpanTree.size(); i++)
        cout << "(" << minSpanTree[i].u << ", " << minSpanTree[i]
            .v << "): " << minSpanTree[i].weight << endl;
    cout << "Sum: " << sum << endl;
}

```

Prøv så med grafen ovenfor, som da er,

```

6 9
0 1 4
1 2 3
2 3 2
3 4 7
4 5 1
5 0 5
0 3 3
1 4 4
1 3 4

```

6.5.2 Prims algoritme

Prims algoritme er en annen måte å kode minimum spanning tree problemt på. Nå bruker vi ikke UnionFind datastrukturen, istedet bruker vi priority_queue. For en forklaring av Prims algoritme se <https://www.youtube.com/watch?v=YyLaRffCdk4>. Vi bygger oss et tre ut fra en node. Prims algoritme har en kompleksitet lik $\mathcal{O}(E + V \log V)$, der E er antall kanter og V er antall noder. Dette er hvis vi bruker en naboliste og ikke en nabomatrise. Siden priority_queue er en såkalt max heap som gir tilbake det største og ikke minste elementet er vi

nødt til å trikse litt. Vi må lure den til å gi oss det minste elementet istedet. Dette gjør vi med å skrive en “feil” overloader for **operator** i vår **struct**. Se koden nedenfor,

```

struct sEdge {
    int fra, til;
    int weight;
    sEdge (int _fra, int _til, int _weight) {
        fra      = _fra;
        til      = _til;
        weight   = _weight;
    }
    bool operator< (const sEdge &other) const { //sortere med
        hensyn paa vektallet
        return weight > other.weight; //byttet om...
    }
};

vector< vector<sEdge> > edge;
priority_queue<sEdge> que;
vector<bool> visited;

void leggTilNaboNoder(int node) {
    for (int i = 0; i < edge[node].size(); i++) {
        int nabo = edge[node][i].til;
        if (!visited[nabo])
            que.push(sEdge(node, nabo, edge[node][i].weight));
    }
}

int main () {
    int N, M; //N=ant. noder, M=ant. kanter
    cin >> N >> M;
    edge.resize(N);

    int a, b, w;
    for (int i = 0; i < M; i++) {
        cin >> a >> b >> w;
        edge[a].push_back(sEdge(a, b, w));
        edge[b].push_back(sEdge(b, a, w));
    }

    visited.assign(N, false);
    visited[0] = true;
    leggTilNaboNoder(0);

    vector<sEdge> minSpanTree;
    int sum = 0;

    while (minSpanTree.size() < N-1) {
        sEdge par = que.top();
        que.pop();

        if (visited[par.til])
            continue;

        visited[par.til] = true;
        minSpanTree.push_back(par);
        sum += par.weight;
        leggTilNaboNoder(par.til);
    }
}

```

```

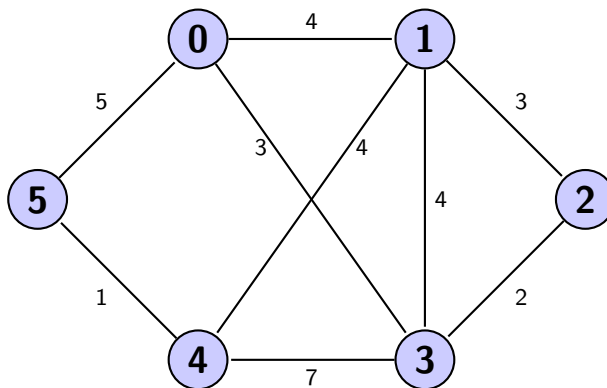
cout << "Minimum spanning tree:\n";
for (int i = 0; i < minSpanTree.size(); i++)
    cout << "(" << minSpanTree[i].fra << ", " << minSpanTree
        [i].til << "): " << minSpanTree[i].weight << endl;
cout << "Sum: " << sum << endl;
}

```

Legg merke til at vi sjekker om `minSpanTree.size()` er mindre enn $N - 1$. Siden et minimum spanning tree inneholder nøyaktig $N - 1$ kanter, der N er antall noder. Faktisk er det sånn at alle trær som har N noder har nøyaktig $N - 1$ kanter. Skal du forbinde 3 øyer ved hjelp av bruer trenger du kun 2 ($3 - 1 = 2$) bruer for å forbinde øyene.

6.6 Shortest path algoritmer

Vi skal nå se på noe vi kaller for shortest path algoritmer for grafer. Det er nemlig gitt en vektet graf, finn korteste vei mellom to gitte noder. Gitt grafen nedenfor,

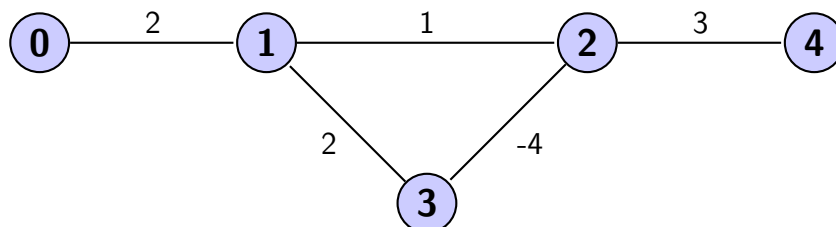


Hvis vi tenker på vektallene i grafen som distanser ser vi at korteste vei fra node **1** til node **5** er $4 + 1 = 5$ der vi går denne ruten (1-4-5). Det var ganske tydelig hvor den korteste veien går, men ved store grafer kan det bli vanskelig å finne korteste veien mellom to noder. Vi trenger da noen algoritmer som kan gjøre dette for oss. Vi har allerede lært om BFS, den kan brukes til å finne korteste vei mellom to noder som vi gjorde for å finne minste antall trekk for en springer å flytte fra et felt til et annet. Med BFS kreves det nemlig at alle vektallene er like (vanligvis 1), siden vi går ett og ett lag utover i grafen helt til vi når fram til noden vi vil nå. Vi vet dermed at vi har gått korteste vei siden alle vektallene er like. Derimot hadde vi kjørt BFS på grafen ovenfor der vi starter i node **1** og vil finne korteste vei til node **5** kan det hende vi hadde gått (1-0-5) som er lenger enn (1-4-5). BFS finner nemlig korteste vei med hensyn på antall noder vi besøker på veien, men vi vil ha korteste vei med hensyn på summen av vektallene på veien.

6.6.1 Dijkstra

Dijkstra er en standardalgoritme for å finne korteste avstand mellom to gitte noder. Den originale implementasjonen av Dijkstra krever at alle vektallene er

positive, men den jeg har implementert nedenfor kan også ha negative verdier. Derimot tåler den ikke at grafen har såkalte negative sykler. Tenk deg at vi skal finne korteste vei fra node 0 til node 4 i grafen under,



Den minste avstanden er nemlig $-\infty$ (minus uendelig). Siden vi kan gå i den negative sykkelen (1-2-3-1) uendelig mange ganger, og for hver gang vi går den minker avstanden med -1 . Grafen har en negativ sykel, dermed udefinert minste avstand, og da også ubrukelig å bruke Dijkstra. Nedenfor er Dijkstra implementasjonen for å finne korteste vei fra én node til alle andre noder, for hjelp, se <https://www.youtube.com/watch?v=8Ls1RqHCOPw>,

```

#define INF 1000000000
typedef pair<int, int> ii;

int main() {
    int V, E;

    cout << "Antall noder og kanter: ";
    cin >> V >> E;

    vector< vector<ii> > edge(V); //der first = vekttall, second
        = nabonode

    int u, v, w;
    for (int i = 0; i < E; i++) {
        cout << "Fra node, til node, vekttall: ";
        cin >> u >> v >> w;
        edge[u].push_back(ii(w, v));
        edge[v].push_back(ii(w, u)); //siden grafen ikke er
            rettet
            //(men den kunne vaert)
    }

    int S;
    cout << "Startnode: ";
    cin >> S;

    vector<int> dist(V, INF);
    dist[S] = 0;

    priority_queue<ii> pq; //distanse til noden, node
    pq.push(ii(0, S));
    //husk at priority_queue gir ut det stoerste elementet
    //vi maa dermed manipulere den til aa gi ut det minste
    //hvis vi har at A<B og legger dem inn i pq,
    //vil pq gi oss B, (men vi vil ha A) derimot hvis vi
    //legger inn (-A) og (-B) inn i pq
    //vil pq gi oss den stoeste, altsaa (-A), vi kan da regne
    //--(-A)=A og faa tilbake den opprinnelige verdien
  
```

```

while (!pq.empty()) {
    ii par = pq.top();
    pq.pop();
    int d = -par.first, p = par.second;
    if (d > dist[p]) continue; //viktig sjekk, tilfelle
        duplikater i pq

    for (int j = 0; j < edge[p].size(); j++) {
        ii tmp = edge[p][j];
        if (d + tmp.first < dist[tmp.second]) {
            dist[tmp.second] = d + tmp.first;
            pq.push(ii(-dist[tmp.second], tmp.second));
        }
    }
}
cout << '\n';
for (int i = 0; i < V; i++)
    cout << "Korteste avstand fra node " << S << " til "
        << i << " er: " << dist[i] << endl;
}

```

Dijkstra algoritmen har en kompleksitet på $\mathcal{O}(E + V \log V)$. Der V og E er henholdsvis antall noder og kanter. Grunnen til at vi får en logaritmisk faktor er pga. priority_queue.

6.6.2 Bellman Ford

Vi så at vi hadde et problem med Dijkstra algoritmen, nemlig at den ikke klarte å håndtere negative sykler. Det som skjer med Dijkstra når den finner negative sykler er at den kommer inn i en uendelig lang loop (vil alltid finne forbedringer). Vi skal nå se på Bellman-Fords algoritme for å finne korteste vei fra en node til alle andre noder (samme som Dijkstra). For en forklaring av Bellman-Fords algoritme se, http://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm eller <https://www.youtube.com/watch?v=05WQNgR4URk> (hvis du foretrekker indisk-engelsk). Her er koden for Bellman-Ford algoritmen,

```

#define INF 1000000000
typedef pair<int, int> ii;

int main() {
    int V, E;

    cout << "Antall noder og kanter: ";
    cin >> V >> E;

    vector< vector<ii> > edge(V); //der first = vekttall, second
        = nabonode

    int u, v, w;
    for (int i = 0; i < E; i++) {
        cout << "Fra node, til node, vekttall: ";
        cin >> u >> v >> w;
        edge[u].push_back(ii(w, v));
        edge[v].push_back(ii(w, u)); //siden grafen ikke er
            rettet
        //((men den kunne vaert)
    }

    int S;
    cout << "Startnode: ";
}

```



```

cin >> S;

vector<int> dist(V, INF);
dist[S] = 0;

for (int i = 0; i < V-1; i++)
    for (int k = 0; k < V; k++) {
        for (int j = 0; j < edge[k].size(); j++) {
            int tmp = edge[k][j];
            dist[tmp.second] = min(dist[tmp.second], dist[k]
                + tmp.first);
        }
    }

bool hasNegativeCycle = false;
for (int k = 0; k < V; k++)
    for (int j = 0; j < edge[k].size(); j++) {
        int tmp = edge[k][j];
        if (dist[tmp.second] > dist[k] + tmp.first)
            hasNegativeCycle = true;
    }

if (hasNegativeCycle)
    cout << "Har negativ sykel\n";
else {
    cout << "Har ikke negativ sykel\n";

    for (int i = 0; i < V; i++)
        cout << "Korteste avstand fra node " << S << " til "
            << i << " er: " << dist[i] << endl;
}
}

```

Kompleksiteten til Bellman-Ford er $O(VE)$, der V og E er henholdsvis antall noder og antall kanter.

6.6.3 Floyd Warshall (all-pair shortest path)

Som nevnt tidligere kan vi brukes Dijkstra for å finne avstanden fra én node til alle andre noder. Derimot hvis vi vil ha korteste avstand mellom alle par av noder, og ikke kun én kilde, kan vi bruke Floyd Warshalls algoritme. Vi kunne selvfølgelig kjørt et Dijkstra søk fra alle nodene og dermed finne ut det samme. Det som gjør Floyd Warshalls algoritme så bra er at den er kort å kode! Algoritmen bruker dynamisk programmering som er neste kapittel, men du kan allikevel forstå algoritmen uten å lese om dynamisk programmering. For en forklaring kan du se http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm og <https://www.youtube.com/watch?v=EMaOMMsA5Jg>. Når vi skal regne ut all-pair shortest path bruker vi en todimensjonal tabell $\mathbf{dist}[i][j]$ som vil da gi oss korteste vei fra node i til node j . I starten setter vi $\mathbf{dist}[i][i]$ lik 0 for alle i (avstanden til seg selv er 0). For hver kant fra node a til node b med vekt w setter vi $\mathbf{dist}[a][b]=w$. Alle andre verdier i tabellen blir satt til 1000000000 (uendelig). Poenget med algoritmen er å gå igjennom alle par av noder, i og j , vi skal så prøve å finne korteste vei mellom disse nodene. Vi prøver da å finne en tredje node k , og ser om $dist[i][k] + dist[k][j] < dist[i][j]$. Altså se om avstanden fra node i til node j der vi på veien går igjennom node k er kortere enn det vi tidligere har funnet ut er korteste vei fra i til j . Se koden nedenfor,

```
#define INF 1000000000
```

```

int main() {
    int V, E;
    cout << "Antall noder og kanter: ";
    cin >> V >> E;

    vector< vector<int> > dist(V, vector<int> (V, INF));
    for (int i = 0; i < V; i++)
        dist[i][i] = 0;

    int u, v, w;
    for (int i = 0; i < E; i++) {
        cout << "Fra node, til node, vekttall: ";
        cin >> u >> v >> w;
        dist[u][v] = dist[v][u] = w;
        //vi tenker oss at man kan gaa begge veier
        //kunne ogsaa bare satt
        //dist[u][v] = w; hvis vi oensket det
    }

    for (int k = 0; k < V; k++)
        for (int i = 0; i < V; i++)
            for (int j = 0; j < V; j++)
                dist[i][j] = min(dist[i][j],
                                dist[i][k] + dist[k][j]);

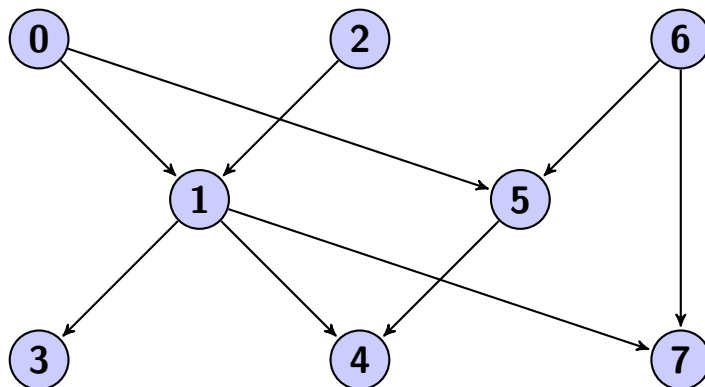
    for (int i = 0; i < v; i++) {
        for (int j = 0; j < V; j++) {
            cout << "Minste avstand fra " << i << " til " << j
                << " er: ";
            if (dist[i][j] == INF)
                cout << "uendelig\n";
            else
                cout << dist[i][j] << endl;
        }
    }
}

```

Som du ser er selve algoritmen kun på 3 linjer! Og på grunn av de tre løkkene har Floyd Warshalls algoritme en kompleksitet lik $\mathcal{O}(V^3)$, der V er antall noder. Altså burde du ikke ha så mye mer enn 300 noder før algoritmen bruker mer enn ett sekund.

6.7 Topologisk sortering

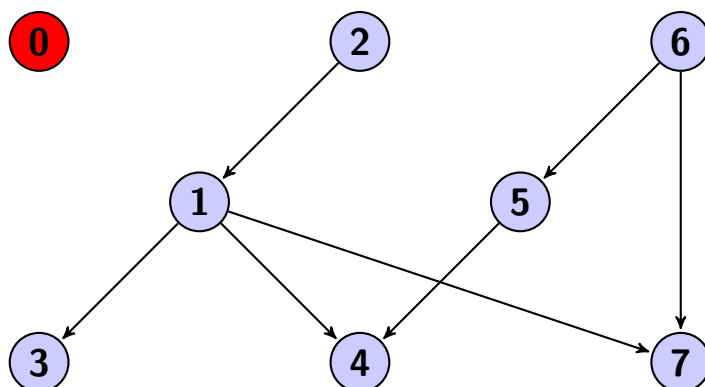
Vi skal nå se på topologisk sortering, vi starter med å se på eksempel. Nå som jeg startet på UiO trengte jeg å velge noen fag jeg skulle ha. Dessverre var det sånn at noen fag krevde at du hadde fullført noen andre fag (fag bygget på hverandre). Så jeg måtte velge fag slik at jeg hadde dekket alle forkunnskapene mine før jeg tok dem. Grafen nedenfor gjenspeiler noen av fagene, pilene betyr at du må ta fagene i pilens retning, altså må du ta fag nr. 0 før fag nr. 1, men du trenger ikke å ha tatt fag nr. 6 før fag nr. 1.



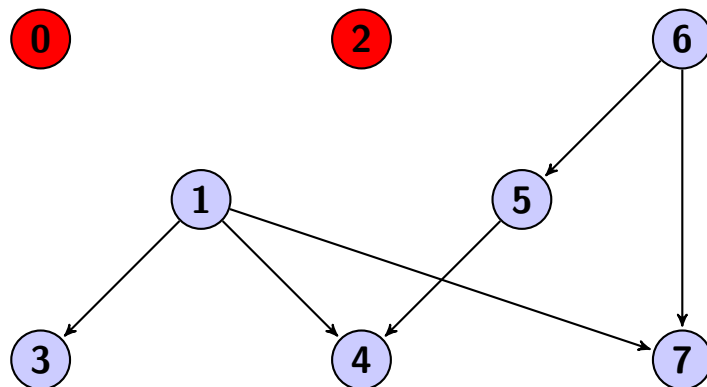
Det vi er ute etter er å lage et program som kan skrive ut en mulig rekkefølge å ta fagene i slik at vi ikke får problemer med at et fag kreves før et annet fag. Under er det beskrevet to måter å gjøre det på. Siden problemet er et grafproblem representerer vi fagene og kantene som en graf (rettet). Legg også merke til at problemet er umulig å løse dersom grafen inneholder sykler, siden da finnes det ikke et fag i den sykkelen som du kan ta, siden forkunnskapene aldri kan bli dekket.

6.7.1 Kahns algoritme

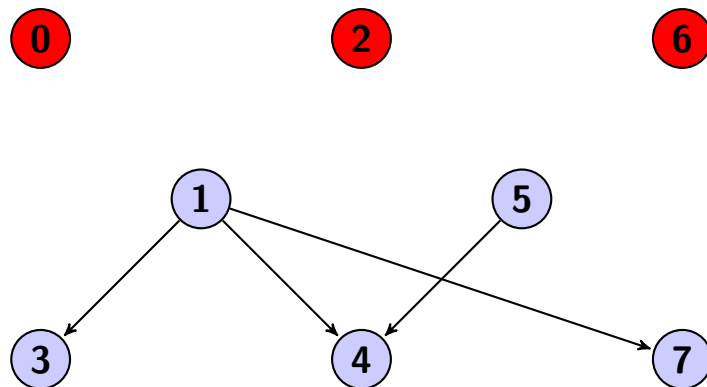
Kahns algoritme er den mest åpenbare og dermed starter vi med den. Vi tenker oss at vi arbeider med grafen ovenfor. Først så går vi gjennom alle nodene i grafen og putter alle nodene som ikke har noen inngående kanter inn i en queue. Siden de ikke har noen kanter inn til seg trenger vi ikke å bekymre oss for at vi trenger å ta noen fag før dem. Vi går så gjennom queue'en og vi er da klare å skrive ut én og én node. Etterhvert som vi skriver ut noder må vi fjerne alle kantene som går ut fra den noden. F.eks. i starten vil queue'en bestå av: 0, 2 og 6. Etter vi har skrevet ut node nr. 0 står vi igjen med grafen,



Der rød betyr at vi har skrevet ut noden. Vi går så videre i queue'en og skriver ut node nr. 2. Vi står da igjen med grafen,



Men nå ser vi at node nr. 1 ikke har noen innkommende kanter, vi kan da legge den til i queue'en. Da vil queue'en inneholde 6 og 1. Vi skriver så ut node nr. 6,



Vi ser så at vi kan legge til node nr. 5 i queue'en. Fortsetter vi slikt vil vi skrive ut alle fagene i en lovlig rekkefølge å ta dem i. Når vi skal implementere algoritmen kan vi istedet for å fjerne kanter, heller ha et tall som sier hvor mange innkommende kanter en node har. Etterhvert som vi skriver ut en node, minker vi denne verdien på alle nabonodene. Her er koden,

```

int main() {
    int V, E;
    cout << "Antall noder og kanter: ";
    cin >> V >> E;

    vector< vector<int> > edge(V);
    vector<int> antInnKanter(V, 0);

    int u, v;
    for (int i = 0; i < E; i++) {
        cout << "Fra node, til node: ";
        cin >> u >> v;
        edge[u].push_back(v);
        antInnKanter[v]++;
    }

    queue<int> que;
    for (int i = 0; i < V; i++)
  
```

```

        if (antInnKanter[i] == 0)
            que.push(i);

vector<int> sortertListe;
while (!que.empty()) {
    int node = que.front();
    que.pop();
    sortertListe.push_back(node);

    for (int j = 0; j < edge[node].size(); j++) {
        int nabo = edge[node][j];
        if (--antInnKanter[nabo] == 0)
            que.push(nabo);
    }
}

cout << "Fagene kan tas i denne rekkefølgen:\n";
for (int i = 0; i < sortertListe.size(); i++)
    cout << sortertListe[i] << ", ";
}

```

Kjører vi koden med inputten nedenfor (som er ekvivalent med grafen ovenfor),

```

8 9
0 1
0 5
1 3
1 4
1 7
2 1
5 4
6 5
6 7

```

Får vi outputten,

```

Fagene kan tas i denne rekkefølgen:
0, 2, 6, 1, 5, 3, 7, 4

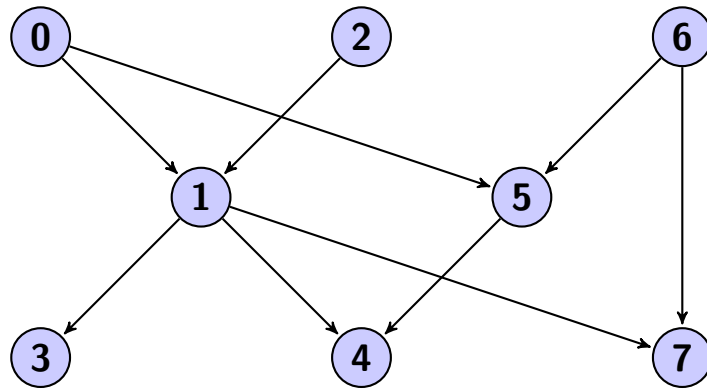
```

Som vi ser stemmer, men dette er én av flere løsninger. Du kan f.eks. bytte om 0 og 2 og fortsatt få en lovlig løsning. Hvis du istedet vil sette et kriterie på rekkefølgen løsningen skal skrives ut i, f.eks. minste tall først, kan du bruke en `priority_queue`. Uten `priority_queue` har algoritmen en kompleksitet lik $\mathcal{O}(V)$, der V er antall noder.

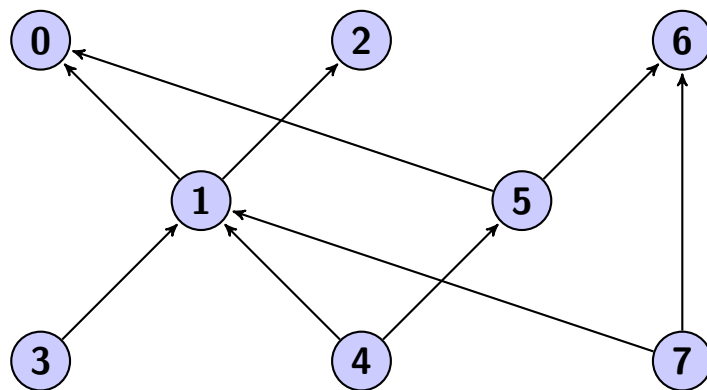
6.7.2 Rekursiv algoritme

Vi skal nå presentere en rekursiv algoritme for å løse det samme problemet. I denne algoritmen må vi holde styr på om en node er skrevet ut eller ikke, vi bruker variabelen **visited** som er en array. Først er ingen noder skrevet ut og derfor er **visited[i]** lik **false** for alle i . Vi går så til en tilfeldig node og prøver å få skrevet den ut, men vi kan ikke skrive ut noden hvis den har noen innkommende kanter til seg. For å løse dette problemet ser vi først om den har

noen innkommende kanter til seg, og hvis ja, så prøver vi rekursivt å skrive ut “parent” nodene (nodene som har en utgående kant til noden vi holder på med). Se på grafen nedenfor,



Hvis vi skal prøve å skrive ut node 3, sjekker vi først om node 3 har noen innkommende kanter, og det har den. Vi prøver da å skrive ut node nr. 1, men den har også innkommende kanter. Vi går så til node nr. 0 og ser at den ikke har innkommende kanter, og vi skriver den derfor ut. Vi går så til den andre parent noden til node nr. 1, node nr. 2 og ser at den heller ikke har innkommende kanter og vi skriver den ut. Nå har heller ikke node nr. 1 innkommende kanter, siden alle parent'ene er skrevet ut. Vi kan da skrive ut node nr. 1. Og dermed kan vi også til slutt skrive ut node nr. 3. Vi går så til neste node som ikke er skrevet ut, kanskje vi havner på node nr. 4. Vi ser at node nr. 4 har innkommende kanter. Vi går så til første parent, node nr. 1. Siden node nr. 1 allerede er skrevet ut hopper vi bare videre. Fortsetter vi algoritmen slikt kommer vi i mål. En annen ting vi må gjøre før vi implementerer koden er å reversere alle kantene. Siden det da blir enklere å finne alle parent-nodene. Vi får da grafen under,



Under er koden,

```

vector<bool> visited;
vector< vector<int> > edge;

void skrivUt(int node) {
    if (visited[node])
  
```

```

        return;

    for (int i = 0; i < edge[node].size(); i++)
        skrivUt(edge[node][i]);

    cout << node << ", ";
    visited[node] = true;
}

int main() {
    int V, E;
    cout << "Antall noder og kanter: ";
    cin >> V >> E;

    edge.resize(V);

    int u, v;
    for (int i = 0; i < E; i++) {
        cout << "Fra node, til node: ";
        cin >> u >> v;
        edge[v].push_back(u); //u og v er byttet om
                               // for aa reversere kantene
    }

    cout << "Fagene kan tas i denne rekkefoelgen:\n";
    visited.assign(V, false);
    for (int i = 0; i < V; i++)
        skrivUt(i);
}

```

Siden vi bruker dybde først søk har algoritmen en kompleksitet lik $\mathcal{O}(E)$, der E er antall kanter.

6.8 Oppgaver

Grafstrukturer:

ID	Navn
599	The Forrest for the Trees (*)
10895	Matrix Transpose (*)
10928	My Dear Neighbours
11550	Demanding Dilemma
11991	Easy Problem from Rujia Liu? (*)

Graf-traversering (DFS / BFS):

ID	Navn
118	Mutant Flatworld Explorers
280	Vertex
10116	Robot Motion
10377	Maze Traversal
11831	Sticker Collector Robot (*)
11906	Knight in a War Grid (*)
12376	As Long as I Learn, I Live
12442	Forwarding Emails (*)

Flood Fill (DFS / BFS):

ID	Navn
352	The Seasonal War
459	Graph Connectivity
469	Wetlands of Florida
572	Oil Deposits
871	Counting Cells in a Blob
11094	Continents (*)
11953	Battleships (*)

Topologisk sortering:

ID	Navn
200	Rare Order
872	Ordering (*)
10305	Ordering Tasks (*)
11060	Beverages (*)

Fargelegging:

ID	Navn
10004	Bicoloring (*)
11080	Place the Guards (*)
11396	Claw Decomposition (*)

All-Pairs Shortest Paths:

ID	Navn
567	Risk
821	Page Hopping (*)
10171	Meeting Prof. Miguel... (*)
11015	05-2 Rendezvous
11463	Commandos (*)
274	Cat and Mouse (*)
1056	Degrees of Separation (*)

Minimum Spanning Tree:

ID	Navn
10034	Freckles
11228	Transportation system. (*)
11747	Heavy Cycle Edges (*)
10048	Audiophobia (*)
10369	Arctic Network (*)
10397	Connect the Campus
10600	ACM Contest and Blackout (*)

Single Source Shortest Path (upektet, BFS):

ID	Navn
336	A Node Too Far
429	Word Transformation (*)
762	We Ship Cheap
924	Spreading The News (*)
10653	Bombs! NO they are Mines!! (*)
314	Robot (*)
11101	Mall Mania (*)
12160	Unlock the Lock (*)

Single Source Shortest Path (vektet, Dijkstra):

ID	Navn
1112	Mice and Maze
10389	Subway
10986	Sending email
10187	From Dusk Till Dawn
10356	Rough Roads
10801	Lift Hopping (*)
11492	Babel (*)
12047	Highest Paid Toll (*)

Single Source Shortest Path (negative sykler, Bellman Ford):

ID	Navn
558	Wormholes (*)
10449	Traffic (*)
10557	XYZZY (*)

7 Dynamisk programmering

Dynamisk programmering er en teknikk som ofte blir brukt i programmeringsoppgaver. Navnet kan være litt misvisende, men teknikken er svært nyttig i **NIO**, **BOI** og **IOI**.

7.1 Memoisering (top-bottom)

Top-bottom også kalt memoisering er én teknikk for å løse dynamisk programmeringsoppgaver. Vi skal nå se på hvordan vi kan bruke denne teknikken til å løse et kjent problem. Si vi er i et land med noen spesielle myntenheter. Si landet har kun mynter med verdiene 1, 4 og 5. Vi skal så kjøpe noe til 9 kr. Hva er det minste antall mynter vi trenger for å lage 9 kr? Den enkleste måten er å ha 9 en-kroninger, men det krever hele 9 mynter! En annen måte er å ha en 5-kroning og fire 1-kroninger. Vi har da redusert fra 9 mynter til 5 mynter. Men vi kan gjøre bedre! Vi kan nemlig ha én 4-kroning og én 5-kroning. Da har vi gått fra 5 mynter til 2 mynter som er det minste antallet vi trenger. I Norge har vi mynter med verdi 1, 5, 10 og 20. Med disse verdiene er ikke oppgaven så vanskelig siden man kan alltid gi den største mynten hele tiden til man er ferdig. F.eks. hvis vi skal betale 57 kr kan vi første betale 20 kr. Vi står igjen med 37 kr. Vi betaler enda en 20 kroning, og står igjen med 17 kr. Nå er 20 kroner for mye, så vi betaler heller med en 10 kroning. Vi står igjen med 7 kr. Nå er 10 kroningen for stor og vi betaler med 5 kroningen. Vi står igjen med 2 kroner. Til slutt betaler vi de siste 2 kronene med to 1-kroninger. Derimot når vi skulle lage 9 kr med myntene 1, 4 og 5 går ikke denne teknikken. For da ender vi opp med at $9 = 5 + 1 + 1 + 1 + 1$ som er 5 mynter! Vi bruker

så dynamisk programmering! (Er teknikken vanskelig å skjønne? Ta en titt på <http://nio.no/bootcamp/?p=205>).

```
int N, M;
int DP[1000];
//DP[i] er minste antall mynter
//du trenger for aa lage beloeper i
//hvis DP[i] = -1 betyr det et beloeper
//som er umulig aa lage med myntene
vector<int> verdi;

int rec(int m) {
    if (m < 0)         return -1;
    if (DP[m] != -1)   return DP[m];

    int ans = -1;
    for (int i = 0; i < N; i++) {
        int tmp = rec(m - verdi[i]);
        if (tmp != -1) {
            if (ans == -1) ans = tmp + 1;
            else           ans = min(ans, tmp + 1);
        }
    }
    return DP[m] = ans;
}

int main() {
    cout << "Antall myntenheter: ";
    cin >> N;

    verdi.resize(N);
    cout << "Myntverdier: ";
    for (int i = 0; i < N; i++)
        cin >> verdi[i];

    memset(DP, -1, sizeof DP);
    DP[0] = 0;
    cout << "Beloeper: ";
    cin >> M;

    cout << "Minste antall mynter for aa lage "
          << M << "kr er: " << rec(M);
}
```

Teknikken vi nå drev med kalles top-bottom siden vi starter ovenifra med kalkulasjonene, vi starter med **rec(m)** og jobber oss ned til **rec(0)**. Hele poenget med **DP** arrayen er at vi ikke skal kalkulere noe mer enn én gang. Vi husker nemlig svaret fra forrige gang.

7.2 Bottom-up

Bottom-up gjør det samme som top-bottom, men regner først ut svaret til de små problemene for så at de større problemene kan bruke svarene på de små problemene senere. Hvis vi skal løse det samme problemet som ovenfor med denne teknikken vil det se slikt ut,

```
int main() {
    int N;
    cout << "Antall myntenheter: ";
    cin >> N;
```

```

vector<int> verdi(N);
cout << "Myntverdier: ";
for (int i = 0; i < N; i++)
    cin >> verdi[i];

vector<int> DP(1000, -1);
//DP[i] er minste antall mynter vi
//trenger for aa lage i kr
//DP[i]=-1 betyr at beloeper
//i kr er umulig aa lage med de
//gitte mynt-verdiene
DP[0] = 0;
for (int i = 0; i < 1000; i++) {
    if (DP[i] == -1) continue;
    for (int j = 0; j < N; j++) {
        int tmp = i + verdi[j];
        if (tmp < 1000) {
            if (DP[tmp] == -1)
                DP[tmp] = DP[i] + 1;
            else
                DP[tmp] = min(DP[tmp], DP[i] + 1);
        }
    }
}

int M;
cout << "Beloeper: ";
cin >> M;
cout << "Minste antall mynter for aa lage "
    << M << "kr er: " << DP[M];
}

```

Vi får da løsningen for alle beløp fra 0 til og med 999.

7.3 Longest increasing sequence (LIS)

Longest Increasing Sequence også forkortet til LIS er et veldig vanlig problem i programmeringsoppgaver. Det handler om at gitt en liste med tall

4, 2, 3, 1, 8, 5, 9

Så skal velge å ta med kun noen tall fra listen i den rekkefølgen de allerede står i, f.eks. 4, 5, 9 slik at de danner en ny liste (subliste) som er sortert fra minst til størst. Ikke bare skal man velge en slik subliste, men du skal finne den sublisten som har disse egenskapene og som er den lengste. I den gitte listen er den lengste økende sublisten lik 2, 3, 8, 9. Du kan ikke plukke ut en annen subliste med disse egenskapene som inneholder mer enn fire elementer. Så hvordan programmerer vi et program som finner LIS? Vi kan jo opplagt prøve alle utvalgene som det finnes 2^n av, også sjekke om de oppfyller egenskapene, også holder styr på den lengste av disse igjen. Men med den måten vil vi få en kompleksitet på $\mathcal{O}(2^n * n)$. Noe som er alt for tregt for store n . Vi kan derimot bruke dynamisk programmering for å løse denne oppgaven. Jeg skal presentere to forskjellige algoritmer for å finne LIS, den første er enkleste å implementere og har en kompleksitet lik $\mathcal{O}(n^2)$, og en annen med kompleksiteten $\mathcal{O}(n \log n)$.

7.3.1 LIS (n^2)

Si vi har gitt listen vi skal finne LIS til. La oss kalle denne listen for v . Vi skal så lagre en tabell med navn **DP**, der vi kan slå opp på **DP[i]** og vi får vite hvor

lang den lengste longest increasing sequence som slutter på indeks i i v er. Si vi har den opprinnelige listen 4, 2, 3, 1, 8, 5, 9. Vi har også at $DP[0] = 1$. Dette er fordi hvis du skal finne en LIS som ender på indeks nummer 0 har vi kun én mulighet, nemlig listen 4. Tilsvarende har vi at $DP[1] = 1$ nemlig LIS lik 2, siden listen 4, 2 er ugyldig (ikke stigende). Derimot ser vi at $DP[2] = 2$, siden nå har vi 2, 3 som LIS som slutter på indeks nummer 2. Men hvordan finner vi ut hva $DP[2]$ skal være i koden vår? Vi vet at $v[2] = 3$, så vi vet at tallet foran 3 må være 3 eller mindre. Vi kan da lete etter den lengste LIS som har siste tall mindre eller lik 3, og som har en indeksnummer mindre enn 2 (siden vi må bevare rekkefølgen fra den opprinnelige listen). Vi ser at vi ikke kan bruke $DP[0] = 1$ siden $v[0] = 4 > v[2] = 3$. Derimot kan vi bruke $DP[1] = 1$ siden $v[1] = 2 \leq v[2] = 3$. $DP[1]$ er også den lengste (og eneste) LIS som vi kan skjøte tallet $v[2] = 3$ på. Vi ser da at $DP[2] = DP[1] + 1$. Her er koden,

```
int LIS(vector<int> v) {
    int DP[v.size()];
    int ans = 1;

    for (int i = 0; i < v.size(); i++) {
        int maxLis = 0;
        for (int j = 0; j < i; j++)
            if (v[j] <= v[i])
                maxLis = max(maxLis, DP[j]);

        DP[i] = maxLis + 1;
        ans = max(ans, DP[i]);
    }
    return ans;
}
```

En funksjon som returnerer lengden av den lengste LIS. Der v er listen som beskrevet ovenfor.

Derimot er vi noen ganger ikke kun interessert i lengden av LIS, men også hvilke elementer/tall som er inneholdt i LIS. Vi kan konstruere oss tilbake til hvilke elementer som er inneholdt i LIS med å bruke en tabell, vi kaller den lst som står for last. Den skal holde styr på hva er det forrige elementet i LISen. La oss se på den opprinnelige listen,

4, **2**, **3**, 1, **8**, 5, **9** der jeg har uthevet de tallene som utgjør LIS av disse tallene. Så hvordan finner vi fram til indeksene av tallene? Vi er nemlig ute etter å få tilbake listen 1, 2, 4, 6 som er indeksene av de tallene som utgjør LIS. Så ideen bak lst er at den skal holde styr på det forrige tallet i LISen for hver tall i LISen. Så $lst[6] = 4$ siden tallet før 9 (indeks 6) er tallet 8 (indeks 4). Slår vi så opp på $lst[4] = 2$ er det fordi tallet foran 8 (indeks 4) er tallet 3 (indeks 2). Vi kan da starte på det siste tallet i LISen også rekonstruere tilbake til de opprinnelige tallene/indeksene med å gå ett og ett hakk bakover i listen. Her er koden for en funksjon som tar inn en liste v som beskrevet ovenfor, og som gir tilbake en liste med indeksene til de elementene i v som utgjør LIS i v ,

```
vector<int> LIS(vector<int> v) {
    int DP[v.size()];
    int lst[v.size()];
    int ansLen = 0;
    int ansPos;

    //vi nullstiller lst
    for (int i = 0; i < v.size(); i++)
```

```

        lst[i] = -1;

    for (int i = 0; i < v.size(); i++) {
        int maxLis = 0;
        int bestLis = -1;
        for (int j = 0; j < i; j++) {
            if (v[j] <= v[i] && DP[j] > maxLis)
            {
                maxLis = DP[j];
                bestLis = j;
            }
        }

        DP[i] = maxLis + 1;
        lst[i] = bestLis;

        if (DP[i] > ansLen) {
            ansLen = DP[i];
            ansPos = i;
        }
    }

    //rekonstruere LIS
    vector<int> ans;
    int pos = ansPos;
    do {
        ans.push_back(pos);
        pos = lst[pos];
    } while (pos != -1);
    reverse(LIS.begin(), LIS.end());
    return ans;
}

```

7.3.2 LIS ($n \log n$)

Det finnes en algoritme for å finne LIS i $\mathcal{O}(n \log n)$ tid. Det vi trenger til å kjøre denne algoritmen er arrayer og binærsøking. Algoritmen går igjennom tabellen \mathbf{v} fra indeks 0 til indeks $n-1$ og for hver indeks utfører vi et binærsøk. Underveis holder vi styr på den lengste LIS funnet hittil. Vi bruker den samme tabellen \mathbf{v} som i $\mathcal{O}(n^2)$ løsningen. Etter vi har passert element $v[i]$ har algoritmen lagret verdier i to arrayer:

- $\mathbf{M}[j]$ - lagrer indeksen k med den minste verdien $\mathbf{v}[k]$ slik at det finnes en økende subsekvens med lengde j som slutter på $\mathbf{v}[k]$ der $k \leq i$.
- $\mathbf{lst}[k]$ - lagrer, som vi tidligere har sett, indeksen til det elementet som er foran $\mathbf{v}[k]$ i LISen som slutter med $\mathbf{v}[k]$.

I tillegg til disse to arrayene har vi en variabel \mathbf{L} som representerer lengden av den lengste økende subsekvensen funnet hittil. Vi bruker heller aldri verdien $\mathbf{M}[0]$, siden en LIS med lengde 0 ikke gir mening. Legg også merke til at sekvensen,

$$v[M[1]], v[M[2]], \dots, v[M[L]]$$

er ikke-avtagende. For hvis det er en økende subsekvens med lengde i som slutter på $\mathbf{v}[M[i]]$ så finnes det også en subsekvens med lengde $i-1$ som slutter på en mindre verdi. Nemlig den som slutter på $\mathbf{v}[lst[M[i]]]$. Siden følgen alltid er økende kan vi bruke binærsøk på den.

Her er implementasjonen,

```
vector<int> LIS(vector<int> v) {
    int n = v.size();

    vector<int> M(n + 1); //bruker ikke indeks 0
    vector<int> lst(n, -1);

    int L = 0;
    for (int i = 0; i < n; i++) {
        //Vi binaersoker for stoerste j <= L
        //slik at v[M[j]] < v[i]
        int lo = 1, hi = L;
        while (lo <= hi) {
            int mid = (lo + hi) / 2;
            if (v[M[mid]] < v[i])
                lo = mid + 1;
            else
                hi = mid - 1;
        }

        //Naa har vi funnet en posisjon (lo)
        //der vi kan legge til v[i]
        int newL = lo;

        //Indeksen til tallet som er foer
        //v[i] i LIS'en er da M[newL-1]
        //nemlig den forrige subsekvensen
        //som var 1 mindre i lengden
        lst[i] = M[newL-1];

        if (newL > L) {
            //Vi fant den lengste LISen hittil
            M[newL] = i;
            L = newL;
        } else if (v[i] < v[M[newL]]) {
            //Hvis siste tallet i subsekvensen
            //som har lengde newL er stoerre
            //enn v[i], erstatter vi siste tallet
            //i den sekvensen med v[i]
            M[newL] = i;
        }
    }

    //rekonstruer LIS
    vector<int> ans(L);
    int pos = M[L];
    for (int idx = L - 1; idx >= 0; idx--) {
        ans[idx] = v[pos];
        pos = lst[pos];
    }
    return ans;
}
```

Siden for hvert element binærsøker vi blir kompleksiteten lik $\mathcal{O}(n \log n)$.

Her er oppgave 1 fra NIO-finalen 2003/2004 (Aksjekursen)

Oppgaven er som følger: Gitt verdiutviklingen for en OptiTron aksje, som en rekke heltall, fjern så få som mulig av disse for at sekvensen som er igjen ikke har synkende partier.

Input:

Et heltall $0 < N < 10000$, antall målepunkter for aksjeverdien. Deretter følger N heltall M_i som representerer aksjens verdi. $0 < M_i < 10000$.

Output:

Den lengste ikke-synkende sekvensen med heltall som er mulig å få til ved å fjerne tall fra input. Dersom flere sekvenser er like lange, er det likegyldig hvilken du skriver ut.

Siden $N < 10000$ er det samme om vi bruker LIS med kjøretid $\mathcal{O}(n^2)$ eller $\mathcal{O}(n \log n)$. Løsningen er nedenfor,

```
//implementasjonen av LIS-funksjonen...
int main() {
    int N;
    cin >> N;

    vector<int> M(N);
    for (int i = 0; i < N; i++)
        cin >> M[i];

    vector<int> ans = LIS(M);
    cout << ans[0];
    for (int i = 1; i < ans.size(); i++)
        cout << " " << ans[i];
}
```

7.4 Bitmask (*)

Bitmask i dynamisk programmering er at vi kan lage en DP-state ved hjelp av bittene i et tall. Poenget er bare å se hvordan dette kan gjøres. La oss gå direkte til en oppgave (problemet er omskrevet fra UVA - 11218),

Det finnes 9 personer, du skal dele disse 9 personene i 3 disjunkte grupper med 3 personer i hver gruppe, sånn at hver person er med i kun én gruppe. Uheldigvis er det slikt at noen personer ikke vil være i gruppe med visse andre personer, og noen kombinasjoner er dårligere enn andre kombinasjoner (poengsum etter hvor godt personene jobber sammen). Gitt poengsummen for alle mulige kombinasjoner av 3 personer, hva er det største mulige poengsummen for alle 3 gruppene?

Input:

Inputten består av maksimalt 1,000 testcases. Hver case begynner med en linje med heltallet $0 < n < 81$, antall mulige kombinasjoner. De neste n linjene inneholder 4 positive heltall a, b, c, s ($1 \leq a < b < c \leq 9, 0 < s < 10,000$), som betyr at poengsummen s er for kombinasjonen (a, b, c) . Siste testcasen er etterfulgt av tallet null.

Output

For hver testcase, skriv ut testcasenummeret og største poengsum. Hvis grupperingen er umulig, skriv ut -1 .

For å løse dette problemet kan vi bruke bitmask. Vi kan nemlig ha en array som vist under,


```
int DP[1<<9];
```

Hvis vi er interessert i hva den minste poengsummen er hvis vi skal kun lage grupper med person 0, 2, 4, 6, 7, 8. Kan vi lage oss et binært tall 111010101 der vi ser at bit nr. 0, 2, 4, 6, 7, 8 er “slått” på. Dette binære tallet tilsvarer tallet 469 i titallsystemet. Vi kan da slå opp på **DP[469]** og få svaret vårt. Helt på slutten er vi interessert i hva minste poengsum er der alle personene har formet gruppene. Vi slår da opp på det binære tallet der alle bittene er “slått” på, nemlig $((1 \ll 9) - 1)$. Siden s alltid er positiv kan **DP** $[(1 \ll 9) - 1]$ kun være negativ dersom vi ikke kan danne disse gruppene. Dette gjør vi med å starte med å sette **DP** $[(1 \ll 9) - 1] = -1$. Under er koden,

```
struct sKombinasjon {
    int a, b, c, s;
    sKombinasjon () {}
};

int main() {
    int DP[1<<9];

    int n, tc = 1;
    while (cin >> n, n != 0) {
        memset(DP, -1, sizeof DP);
        DP[0] = 0;
        vector<sKombinasjon> tab(n);

        for (int i = 0; i < n; i++) {
            cin >> tab[i].a >> tab[i].b >> tab[i].c >> tab[i].s;
            //0-indeksierer fra 1-indeksert
            tab[i].a--; tab[i].b--; tab[i].c--;
        }

        for (int i = 0; i < (1<<9); i++) {
            if (DP[i] == -1)
                continue;
            for (int j = 0; j < n; j++) {
                //sjekker at vi ikke lager en gruppe
                //med personer som allerede er brukt
                if ((i & (1<<tab[j].a)) || (i & (1<<tab[j].b))
                    || (i & (1<<tab[j].c)))
                    continue;

                int bitmask = i|(1<<tab[j].a)|(1<<tab[j].b)
                    |(1<<tab[j].c);
                DP[bitmask] = max(DP[bitmask], DP[i]+tab[j].s);
            }
        }
        cout << "Case " << tc++ << ": " << DP[(1<<9)-1] << endl;
    }
}
```

7.5 Oppgaver

Generelle:

ID	Navn
116	Unidirectional TSP
10003	Cutting Sticks
10036	Divisibility
10337	Flight Planner (*)
10400	Game Show Math
10446	The Marriage Interview :-)
10465	Homer Simpson
10721	Bar Codes (*)
10910	Marks Distribution
10912	Simple Minded Hashing
10943	How do you add? (*)
10980	Lowest Price in Town
11407	Squares
11420	Chest of Drawers
11450	Wedding shopping
11703	sqrt log sin

Max Range Sum:

ID	Navn
10684	The jackpot (*)
108	Maximum Sum (*)
836	Largest Submatrix
10667	Largest Block
10827	Maximum sum on a torus (*)
11951	Area (*)

LIS:

ID	Navn
437	The Tower of Babylon
481	What Goes Up (*)
10534	Wavio Sequence

Subset sum (Knapsack):

ID	Navn
562	Dividing coins
10616	Divisible Group Sums (*)

Coin Change:

ID	Navn
147	Dollars
357	Let Me Count The Ways (*)
674	Coin Change

Bitmask:

ID	Navn
10081	Tight Words
10536	Game of Euler (*)
10651	Pebble Solitaire (*)
10898	Combo Deal
10911	Forming Quiz Teams (*)
11088	End up with More Teams
11218	KTV