Algoritmer

Tony Johansson
1MA239: Specialkurs i Matematik II
Uppsala Universitet

VT 2018

Algoritmer är ett grundläggande koncept inom matematik och datavetenskap, och deras relevans i våra dagliga liv lär inte ha undgått någon vid det här laget. I detta dokument introducerar vi några kända algoritmer, och besvarar de grundläggande frågorna som berör varje algoritm: gör algoritmen alltid det vi vill göra, och hur många steg tar algoritmen innan den är färdig?

Vi börjar med att definiera en algoritm, och går igenom ett par exempel på hur de ska läsas. Om du har programmerat förut kan du förmodligen hoppa över Sektion 1.

Efter det går vi över till ett enkelt, och mycket gammalt, exempel på en algoritm, nämligen Euklides algoritm för att beräkna största gemensamma delaren för två positiva heltal s, t, för att gå vidare till sorteringsalgoritmer och slutligen grafalgoritmer.

1 Introduktion till algoritmer

I grunden gör en algoritm följande. Den tar in ett objekt x, genomför en sekvens av instruktioner, och spottar ut (*returnerar*) ett objekt y. Objekten kan vara lite vad som helst, till exempel tal, ord eller *boolska variabler*.

En boolsk variabel är ett objekt x som antingen är Sant eller Falskt. I programmeringsspråk är det True eller False. Ibland motsvaras Sant av 1, och Falskt av 0. En boolsk variabel beskrivs enklast som ett påstående som är sant eller falskt, till exempel "det är måndag", "$10 > \pi$", "$\sqrt{24}$ är ett heltal", och så vidare.

Vi börjar med ett grundläggande exempel på en algoritm, se Algoritm 1 nedan. Algoritmen är skriven i så kallad *pseudokod*, en beskrivning av en algoritm som inte är specifik till något särskilt programmeringsspråk.

Algoritm 1 anropas med $x = (a, b)$, där a, b är två heltal med $a \geq 0$ och $b \geq 1$ och returnerar $y = a \mod b$. För $b \geq 1$ är $y = a \mod b$ definierat som resten vid division a/b, det vill säga y är det unika talet i $\{0, 1, \ldots, b-1\}$ för vilket det finns ett heltal q så att

$$a = qb + y.$$

Vi kan beräkna $a \mod b$ genom att upprepade gånger subtrahera b från a. Till exempel kan vi ta $a = 19$ och $b = 4$, och dra bort 4 tills $a < b$. Så vi får en sekvens $a = 19, 15, 11, 7, 3$, och konstaterar att $a \mod b = 3$. Detta är precis vad vi gör i Algoritm 1.

Algorithm 1 Moduloberäkning $a \mod b$ med $a \geq 0$ och $b \geq 1$.

```
1: procedure MOD(a, b)
2:     while a > b do
3:         a ← a − b
4:     return a
```

Det finns flera saker som behöver förklaras här. Först och främst: en algoritm körs rad för rad från rad 1 och nedåt. Vi börjar med rad 1 som sätter igång processen, genom att någon anropar MOD(19, 4).

Vi går till rad 2, som sätter igång en *loop* av typen **while**. En while-loop består av en boolsk variabel (i detta fallet "$a > b$") och en sekvens av instruktioner. I detta fall består sekvensen av en instruktion, $a \leftarrow a - b$. Det som avgör vilka rader som ingår i while-loopen är indenderingen: eftersom rad 4 har samma indentering som rad 2, så ingår inte "**return** a" i while-loopen.

While-loopen gör följande: avgör om den boolska variabeln $a > b$ är sann. Om så är fallet, utför sekvensen av instruktioner, och gå tillbaka till rad 2 och upprepa. Förhoppningsvis når vi till slut en punkt då "$a > b$" = FALSE, och då avslutas loopen. Då går algoritmen vidare till nästa rad med samma indendtering som rad 2, det vill säga rad 4.

I vårt exempel är "$19 > 4$" = TRUE, så vi går till rad 3. På rad 3 uppdaterar vi värdet av a genom $a \leftarrow a - b$. Innan detta utförs är $(a, b) = (19, 4)$, och efter det utförs är $(a, b) = (15, 4)$. Detta skrivs ibland $a := a - b$, och i de flesta programmeringsspråk används det något förvirrande $a = a - b$.

Return är kommandot som returnerar värdet som försöker beräknas. När en **return**-rad nås avslutas algoritmen, oavsett var i koden som **return** används, och algoritmen spottar ut det som står till höger om **return**.

Låt oss gå igenom exemplet MOD$(19, 4)$ rad för rad.

1 : Vi startar MOD$(19, 4)$.
2 : Vi har $19 > 4$ så gå till rad 3.
3 : Uppdatera $a \leftarrow 19 - 4 = 15$. Så $(a, b) = (15, 4)$.
vi har nått slutet på **while**-loopen och går tillbaka till rad 2.
2 : Vi har $15 > 4$ så gå till rad 3.
3 : Uppdatera $a \leftarrow 15 - 4 = 11$. Så $(a, b) = (11, 4)$.
2 : Vi har $11 > 4$ så gå till rad 3.
3 : Uppdatera $a \leftarrow 11 - 4 = 7$. Så $(a, b) = (7, 4)$.
2 : Vi har $7 > 4$ så gå till rad 3.
3 : Uppdatera $a \leftarrow 7 - 4 = 3$. Så $(a, b) = (3, 4)$.
2 : Vi har $3 < 4$ så gå till rad 4.
4 : Returnera $a = 3$.

Exemplet ovan förklarar vad en while-loop är, och vad "return" innebär. Låt oss se ett till exempel på en algoritm, som introducerar for-loopar och if-satser. Algoritm 2 tar in ett positivt heltal och returnerar två mängder D, N av tal $1 \leq i < n$, där $i \in D$ om $n \mod i = 0$, och annars är i i N.

Algorithm 2 Hitta delare och icke-delare till $n \geq 1$

```
1: procedure DELARE(n)
2:     D ← ∅
3:     N ← ∅
4:     for i = 1 to n − 1 do
5:         if MOD(n, i) = 0 then
6:             D ← D ∪ {i}
7:         else
8:             N ← N ∪ {i}
9:     return D, N
```

Låt oss börja med att förklara **for**-loopen. Kommandot "**for** $i = 1$ to $n - 1$ **do**" säger att det som ingår i loopen (rad 5 – 8, som indenteringen avslöjar) ska göras i turordning för $i = 1, 2, \ldots, n - 1$. Första gången rad 4 nås går vi till rad 5–8 med $i = 1$. När slutet av loopen nås upprepar vi detta med $i = 2$, och så vidare upp till $n - 1$. I de flesta situationer kan både en for- och en while-loop användas, men båda har sina för- och nackdelar.

I steg 5 når vi en **if**-sats. Vi har två tal i och n, och beräknar $n \mod i$ (förslagsvis med Algoritm 1). Om svaret på frågan i rad 5 ("är $n \mod i = 0$?") är Ja, så går vi till nästa rad (indenteringen avslöjar igen vad som omfattas av ett Ja-svar). Om svaret är Nej hoppar vi istället till det som står under "else", det vill säga vi utför rad 8. Ibland finns ingen else-rad: då leder ett Nej-svar till att if-satsen inte utför något alls.

I

2 Euklides algoritm

Definition 1. *Låt $s, t \geq 1$ vara heltal. Den* största gemensamma delaren för s och t, *betecknad $\text{sgd}(s,t)$, är det största heltal $k \geq 1$ sådant att k delar s och t.*

På engelska heter detta *greatest common divisor* och betecknas $\gcd(s,t)$. Det förekommer även att det betecknas (s,t), men vi håller oss till $\text{sgd}(s,t)$ såvida jag inte gör något misstag. Till exempel är

$$\text{sgd}(42,54) = 6,\ \text{sgd}(18,19) = 1,\ \text{sgd}(100,36) = 4,\ \text{sgd}(12,12) = 12.$$

Euklides algoritm beskrevs först omkring 300 f.Kr., och är fortfarande mycket relevant. I ord kan den beskrivas såhär. Anta att $s > t$ (om $s = t$ är $\text{sgd}(s,t) = s = t$, och om $s < t$ kan vi bara byta plats på dem). Låt $q_0 = \lfloor a/b \rfloor$ (dvs a/b avrundat neråt) och $r_0 = s - q_0 t \in \{0, 1, \ldots, t-1\}$. Vi kan skriva

$$s = q_0 t + r_0.$$

I termer av lågstadiematematik är q_0 kvoten i s/t, och r_0 är resten. Vi kan skriva $r_0 = s \mod t$. Vi gör nu samma sak med t och r_0 (notera att $r_0 < t$), dvs låter $q_1 = \lfloor t/r_0 \rfloor$ och $r_1 = t - q_1 r_0$. Vi upprepar sen för r_0 och r_1, tills $r_k = 0$ för något $k \geq 0$. Sekvensen av beräkningar är

$$\begin{aligned} s &= q_0 t + r_0, \\ t &= q_1 r_0 + r_1, \\ r_0 &= q_2 r_1 + r_2, \\ r_1 &= q_3 r_2 + r_3, \\ &\vdots \\ r_{k-2} &= q_k r_{k-1} + r_k, \end{aligned} \tag{1}$$

där $r_k = 0$. För att passa in i notationen kan vi låta $s = r_{-2}$ och $t = r_{-1}$. Vi drar slutsatsen att

$$\text{sgd}(s, t) = r_{k-1}.$$

Nedan följer en så kallad *pseudokod* som beskriver algoritmen. Det finns för- och nackdelar med att beskriva en algoritm i ord eller som pseudokod. Jag brukar föredra att skriva ut en algoritm i ord, för att det finns mer utrymme för förklaring, men det är en smaksak och det finns inget rätt och fel.

Algorithm 3 Euklides algoritm

```
1: procedure EUKLIDES(s, t)
2:     r ← s mod t
3:     while r ≠ 0 do
4:         s ← t
5:         t ← r
6:         r ← s mod t
7:     return b
```

I denna beskrivningen håller vi inte reda på sekvensen $r_0, r_1, \ldots, r_k$, men den kan lätt anpassas för att göra det. Vi specificerar till exempel inte hur $s \mod t$ ska beräknas.

I studien av en algoritm finns tre grundläggande huvudfrågor:

(i) Ger algoritmen alltid ett korrekt svar?

(ii) Terminerar algoritmen alltid efter ett ändligt antal steg?

(iii) Hur många steg tar det för algoritmen att terminera?

Tekniskt sett ger ett svar till (iii) också ett svar till (ii), men jag har med (ii) för att betona hur viktig den är. Traditionellt kräver vi att svaret ska vara JA till (i) och (ii), men nuförtiden är det många algoritmer där vi nöjer oss om svaret är "Ja med hög sannolikhet" eller liknande.

Proposition 1. *Låt $s \geq t \geq 1$.*

(i) EUKLIDES(s, t) *terminerar efter ett ändligt antal steg.*

(ii) EUKLIDES(s, t) *returnerar alltid sgd(s, t).*

Bevis. Vi börjar med att bevisa (i). Låt $r_{-2} = s$ och $r_{-1} = t$ och definiera $r_0, \ldots, r_k$ som i (1). Vi behöver visa att det finns ett ändligt k så att $r_k = 0$. Men för varje i definieras $r_i = r_{i-2} \mod r_{i-1} \in \{0, 1, \ldots, r_{i-1} - 1\}$. Så vi

har en sekvens av heltal $s = r_{-2} > r_{-1} > r_0 > \cdots > r_k \geq 0$, så vi måste ha $0 \leq r_k < s - k - 2$. Alltså måste det finnas ett $k \leq s - 1$ sådant att $r_k = 0$. Detta visar (i).

Nu för (ii). Vi behöver visa att om $r_k = 0$ så är $r_{k-1} = \text{sgd}(s, t)$. För varje $i \leq k$ har vi

$$r_{i-2} = q_i r_{i-1} + r_i.$$

Låt $a = \text{sgd}(r_{i-2}, r_{i-1})$. Då är

$$r_i = r_{i-2} - q_i r_{i-1} = a\left(\frac{r_{i-2}}{a} - q_i \frac{r_{i-1}}{a}\right),$$

och eftersom r_{i-2}/a och r_{i-1}/a är heltal har vi även att a delar r_i. Så $\text{sgd}(r_{i-1}, r_i) \geq \text{sgd}(r_{i-2}, r_{i-1})$.

Anta för motsägelse att det finns en gemensam delare $b > a$ till r_{i-1} och r_i. Eftersom $r_{i-2} = q_i r_{i-1} + r_i$ måste vi då även ha att b delar r_{i-2}, så $b > \text{sgd}(r_{i-2}, r_{i-1})$ delar r_{i-2} och r_{i-1}, en motsägelse. Vi har visat att

$$\text{sgd}(r_{i-2}, r_{i-1}) = \text{sgd}(r_{i-1}, r_i) \quad \text{för } i = 0, 1, \ldots, k.$$

Så vi har

$$\text{sgd}(r_{-2}, r_{-1}) = \text{sgd}(r_{-1}, r_0) = \cdots = \text{sgd}(r_{k-2}, r_{k-1}) = r_{k-1}.$$

Den sista likheten följer från att $r_k = 0$, så $r_{k-2} = q_k r_{k-1}$. Detta bevisar (ii). □

Istället för att gå in på hur snabb Euklides algoritm är (vilket är förvånansvärt jobbigt att besvara) så går vi nu vidare till andra algoritmer.

3 Binärsökning

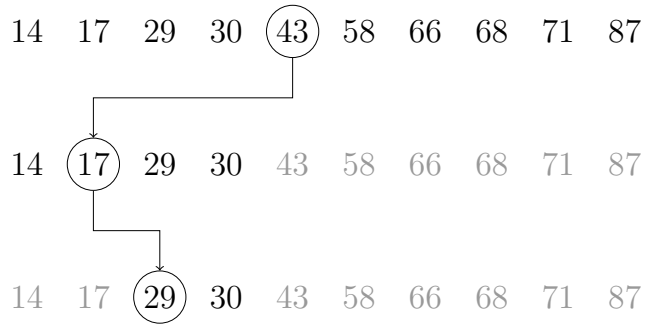
Binärsökning är en algoritm för att hitta ett element ur en sorterad lista. Låt säga att vi har en lista av par (födelsedatum, namn), som är sorterad

efter födelsedatum. Till exempel kan listan vara

(790816, Simon)
(790527, Martin)
(840501, Josefin)
(860126, Anton)
(860213, Sandra)
(870712, Albin)
(890114, Elinor)
(890520, Jonas)
(921201, Marcus)

Binärsökningsalgoritmen används i situationer där vi har ett födelsedatum och vill ta reda på namnet på personen i vår databas som är född det datumet. Vi antar för enkelhetens skull att inget födelsedatum förekommer mer än en gång.

I beskrivningen av algoritmen kommer vi förenkla situationen: vi har en lista av tal $k_1 < k_2 < \cdots < k_n$ (k står för *key*), och vi ignorerar eventuell annan information som kan vara förknippad med talen. Algoritmen anropas med ett tal k (låt oss anta att vi vet att k finns i listan), och returnerar det index i för vilket $k_i = k$.



Figur 1: Exempel på binärsökning, där vi söker efter positionen för nyckeln 29. I varje steg jämförs medianen (inringad) med vår sökta nyckel, och ungefär hälften av de kvarvarande elementen utesluts.

Till algoritmen ger vi en nyckel k och listan $k_1, \ldots, k_n$. Säg att vi anropar BINÄR$(k; k_1, \ldots, k_n)$. Algoritmen börjar med att ta medianen $k_{\lfloor n/2 \rfloor}$

(vi avrundar nedåt om n är udda — ett godtyckligt val), och avgöra om $k < k_{\lfloor n/2 \rfloor}, k > k_{\lfloor n/2 \rfloor}$ eller $k = k_{\lfloor n/2 \rfloor}$. I det sista fallet har vi hittat vår nyckel, och algoritmen returnerar $i = \lfloor n/2 \rfloor$. Om $k < k_{\lfloor n/2 \rfloor}$ anropar vi rekursivt BINÄR$(k; k_1, \ldots, k_{\lfloor n/2 \rfloor - 1})$, och om $k > k_{\lfloor n/2 \rfloor}$ anropar vi rekursivt BINÄR$(k; k_{\lfloor n/2 \rfloor + 1}, \ldots, k_n)$. Figur 1 visar ett exempel på hur binärsökning fungerar.

I varje steg halverar vi längden på söklistan, och inom $\lceil \log_2 n \rceil$ steg har vi hittat vårt index i. Detta ber jag er visa i Problem 1.

Binärsökningsalgoritmen presenteras i pseudokod i Algoritm 4. Notera att så som algoritmen är skriven förväntar den sig en lista vars index börjar med 1, så när vi vill anropa algoritmen med $(k_{\lfloor n/2 \rfloor + 1}, \ldots, k_n)$ får vi definiera en identisk lista $(\ell_1, \ldots, \ell_{n - \lfloor n/2 \rfloor})$ med de index algoritmen förväntar sig.

Algorithm 4 Binärsökningsalgoritmen

```
1:  procedure BINÄR(k; k_1, k_2, ..., k_n)
2:      if n = 1 then
3:          return 1
4:      else
5:          p ← k_⌊n/2⌋
6:          if k = p then
7:              return ⌊n/2⌋
8:          if k < p then
9:              return BINÄR(k; k_1, k_2, ..., k_⌊n/2⌋−1)
10:         if k > p then
11:             (ℓ_1, ℓ_2, ..., ℓ_n−⌊n/2⌋) ← (k_⌊n/2⌋+1, ..., k_n)
12:             return ⌊n/2⌋+ BINÄR(k; ℓ_1, ℓ_2, ..., ℓ_n−⌊n/2⌋)
```

4 Sorteringsalgoritmer

Låt oss säga att vi har en lista $(a_1, a_2, \ldots, a_n)$ av tal, alla olika för enkelhetens skull. Vi är ute efter en algoritm som sorterar listan, alltså tar in $(a_1, \ldots, a_n)$ och returnerar samma lista i en ny ordning $(a_{(1)}, a_{(2)}, \ldots, a_{(n)})$ där $a_{(1)} < a_{(2)} < \cdots < a_{(n)}$.

Det finns en mängd olika sorteringsalgoritmer[1] som lämpar sig olika väl beroende på situation — ibland kanske vi vet i förväg att $(a_1, \ldots, a_n)$ redan är någorlunda sorterad. Låt oss börja med Bubblesort.

[1] Visualisering av sorteringsalgoritmer: `https://www.youtube.com/watch?v=kPRA0W1kECg`

4.1 Bubblesort

Bubblesort är värdelös om n är stort[2], men brukar presenteras först eftersom den är enkel att förklara och koda. Det händer att jag själv använder Bubblesort i kod om jag behöver skriva en enkel sorteringsrutin som bara kommer användas på 10–20 tal, säg.

Algorithm 5 Bubblesort

```
1: procedure BUBBLESORT(a_1, ..., a_n)
2:     (a_(1), ..., a_(n)) ← (a_1, ..., a_n)
3:     for i = 1 to n − 1 do
4:         for j = 2 to n do
5:             if a_(j−1) > a_(j) then
6:                 swap(a_(j−1), a_(j))
       return (a_(1), ..., a_(n))
```

Här är 'swap' en funktion som byter plats på $a_{(j-1)}$ och $a_{(j)}$. Låt oss gå igenom ett exempel av Bubblesort, med listan $(a_1, \ldots, a_5) = (6, 1, 7, 4, 3)$.

$$
\begin{aligned}
&(6, 1, 7, 4, 3) && && \\
&(\underline{1}, \underline{6}, 7, 4, 3) && i = 1,\ j = 2 && (2)\\
&(1, \underline{6}, \underline{7}, 4, 3) && i = 1,\ j = 3 && (3)\\
&(1, 6, \underline{4}, \underline{7}, 3) && i = 1,\ j = 4 && (4)\\
&(1, 6, 4, \underline{3}, \underline{7}) && i = 1,\ j = 5 && (5)\\
&(\underline{1}, \underline{6}, 4, 3, 7) && i = 2,\ j = 2 && \\
&(1, \underline{4}, \underline{6}, 3, 7) && i = 2,\ j = 3 && \\
&(1, 4, \underline{3}, \underline{6}, 7) && i = 2,\ j = 4 && \\
&(1, 4, 3, \underline{6}, \underline{7}) && i = 2,\ j = 5 && (6)\\
&(\underline{1}, \underline{4}, 3, 6, 7) && i = 3,\ j = 2 && \\
&(1, \underline{3}, \underline{4}, 6, 7) && i = 3,\ j = 3 && \\
&(1, 3, \underline{4}, \underline{6}, 7) && i = 3,\ j = 4 && \\
&(1, 3, 4, \underline{6}, \underline{7}) && i = 3,\ j = 5 && (7)\\
&(\underline{1}, \underline{3}, 4, 6, 7) && i = 4,\ j = 2 && \\
&(1, \underline{3}, \underline{4}, 6, 7) && i = 4,\ j = 3 && \\
&(1, 3, \underline{4}, \underline{6}, 7) && i = 4,\ j = 4 && \\
&(1, 3, 4, \underline{6}, \underline{7}) && i = 4,\ j = 5 && (8)
\end{aligned}
$$

[2]Källa: `https://www.youtube.com/watch?v=koMpGeZpu4Q`

Proposition 2. *Bubblesort returnerar alltid en lista* $(a_{(1)}, \ldots, a_{(n)})$ *med* $a_{(1)} < \cdots < a_{(n)}$.

Bevis. Vi kan först notera att när $i = 1$ och $j = n-1$, som i (5), så är största talet i listan $(a_1, \ldots, a_n)$ i position $a_{(n)}$. Detta är vad som ger algoritmen dess namn: de största elementen "bubblar upp" till den högsta positionen. Låt oss först bevisa detta.

Låt oss säga att a_ℓ är det största elementet i $(a_1, \ldots, a_n)$ för något $1 \leq \ell \leq n$. Vi går igenom de steg då $i = 1$. I exemplet ovan är $a_\ell = 7$ och $\ell = 3$.

1. När $i = 1$ och $j < \ell$ flyttas bara elementen $a_1, \ldots, a_{\ell-1}$, vilket inte påverkar a_ℓ. Detta är steg (2).

2. När $i = 1$ och $j = \ell$ jämförs $a_{(\ell-1)}$ (som kan skilja sig från $a_{\ell-1}$) och $a_{(\ell)} = a_\ell$. Eftersom $a_\ell > a_{(\ell-1)}$ då a_ℓ är det största talet, så flyttas inte a_ℓ. Detta är steg (3).

3. När $i = 1$ och $j = \ell + 1$ konstaterar vi att $a_\ell = a_{(\ell)} > a_{(\ell+1)} = a_{\ell+1}$ och byter plats på de två, som i steg (4).

4. När $i = 1$ och $j = \ell + 2, \ldots, n$ så kommer vi hela tiden ha $a_{(j-1)} = a_\ell$, och vi byter plats i varje steg. I slutändan är $a_{(n)} = a_\ell$. I exemplet är detta bara steg (5).

Detta visar att stadiet av algoritmen då $i = 1$ slutar med att det största elementet är i position $a_{(n)}$. Det är inte svårt att se att det kommer stanna där.

På samma sätt kan vi visa att stadiet $i = 2$ slutar med att rätt element hamnar i $a_{(n-1)}$, och kommer stanna där. Generellt är det så att i slutet av stadie i har vi rätt element i position $n - i + 1, n - i, \ldots, n$. Efter stadiet $i = n-1$ har vi alltså en lista $(a_{(1)}, a_{(2)}, \ldots, a_{(n)})$ där $a_{(2)} < a_{(3)} < \cdots < a_{(n)}$ är de $n - 1$ största elementen. Då måste $a_{(1)}$ vara det minsta elementet, och vi är klara. □

Antalet steg en sorteringsalgoritm tar brukar mätas i hur många jämförelser som görs, alltså hur många gånger vi går igenom steg 5 i Algoritm 5. Detta eftersom jämförelsen ofta är den del som tar längst tid. För Bubblesort (i den form vi använder) är svaret **alltid** $(n-1)(n-1)$. Detta kan förbättras lite: till exempel är steg (6), (7) och (8) i exemplet ovan helt onödiga. I Problem 2 ber jag er ändra algoritmen så att inga garanterat onödiga steg utförs.

5 Quicksort

Jag presenterar nu en snabbare sorteringsalgoritm, nämligen Quicksort. I värsta fall kan även Quicksort ta uppåt n^2 steg, men i snitt tar den cirka $2n \log n$ steg, vilket är betydligt snabbare än Bubblesort.

Quicksort är, likt binärsökning, en rekursiv algoritm. Jag tycker att den enklast beskrivs i ord[3], men pseudokod presenteras i Algoritm 6. Vi börjar igen med en lista $(a_1, \ldots, a_n)$ av distinkta tal. Vi väljer ett "pivotelement", säg a_1 (mer om valet av pivotelement senare). Vi ger a_1 två säckar, L och R (som a_1 håller i vänster och höger hand, säg). Vi går därefter igenom $a_2, \ldots, a_n$, och lägger de tal som är mindre än a_1 i L och de tal som är större än a_1 i R. Detta ger två listor $L = (\ell_1, \ldots, \ell_s)$ och $R = (r_1, \ldots, r_t)$, som sorteras var för sig med hjälp av Quicksort — rekursivt. När de sorterade listorna $(\ell_{(1)}, \ldots, \ell_{(s)})$ och $(r_{(1)}, \ldots, r_{(t)})$ är klara får vi vår sorterade lista som $(a_{(1)}, \ldots, a_{(n)}) = (\ell_{(1)}, \ldots, \ell_{(s)}, a_1, r_{(1)}, \ldots, r_{(t)})$.

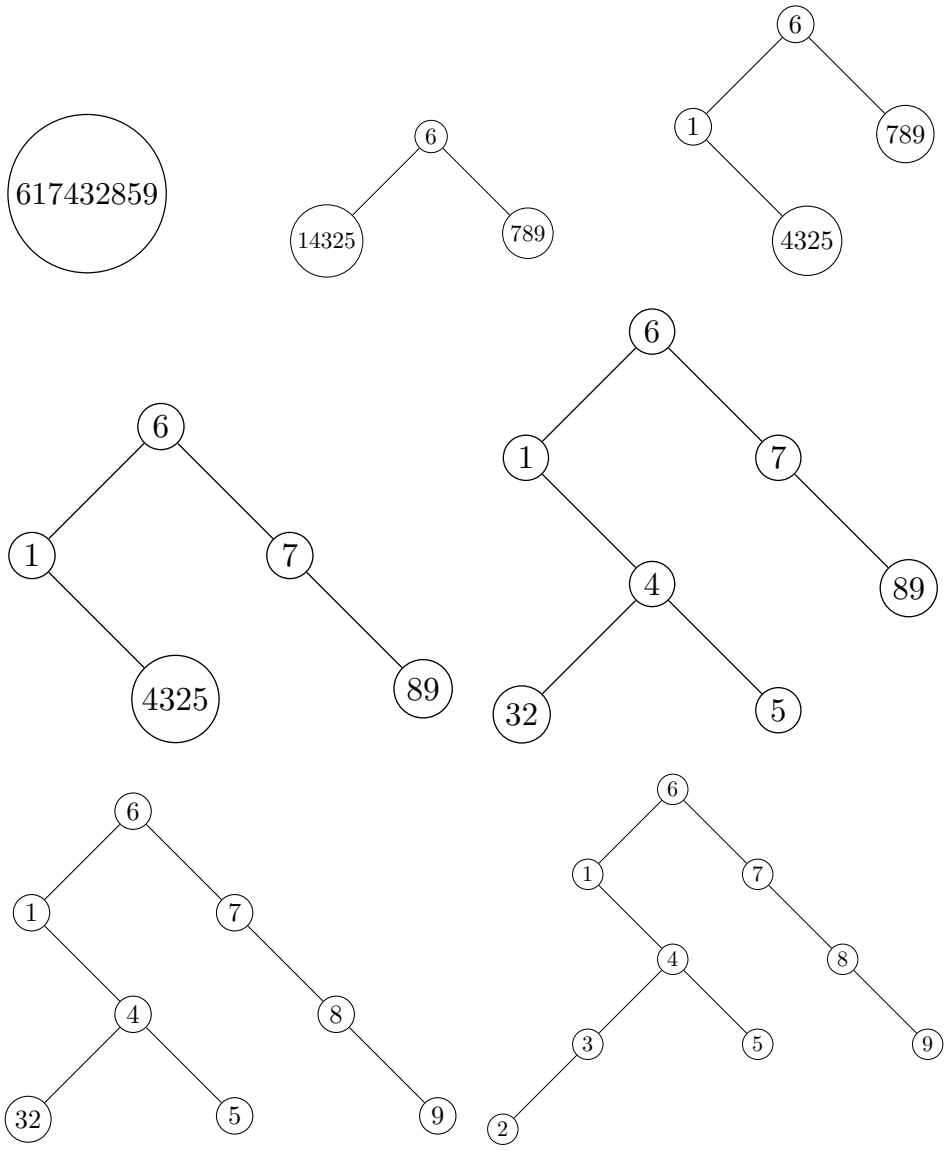
Denna rekursionen måste stanna någonstans — det räcker inte att säga att vi delar in en lista i mindre listor och sorterar dem. Detta löser vi genom att notera att en lista som bara innehåller ett tal redan är sorterad, och inte behöver brytas ner längre. Figur 2 visar hur Quicksort bryter ner listan $6, 1, 7, 4, 3, 2, 8, 5, 9$, och Figur 3 visar hur den sorterade listan återfås.

Algorithm 6 Binärsökningsalgoritmen

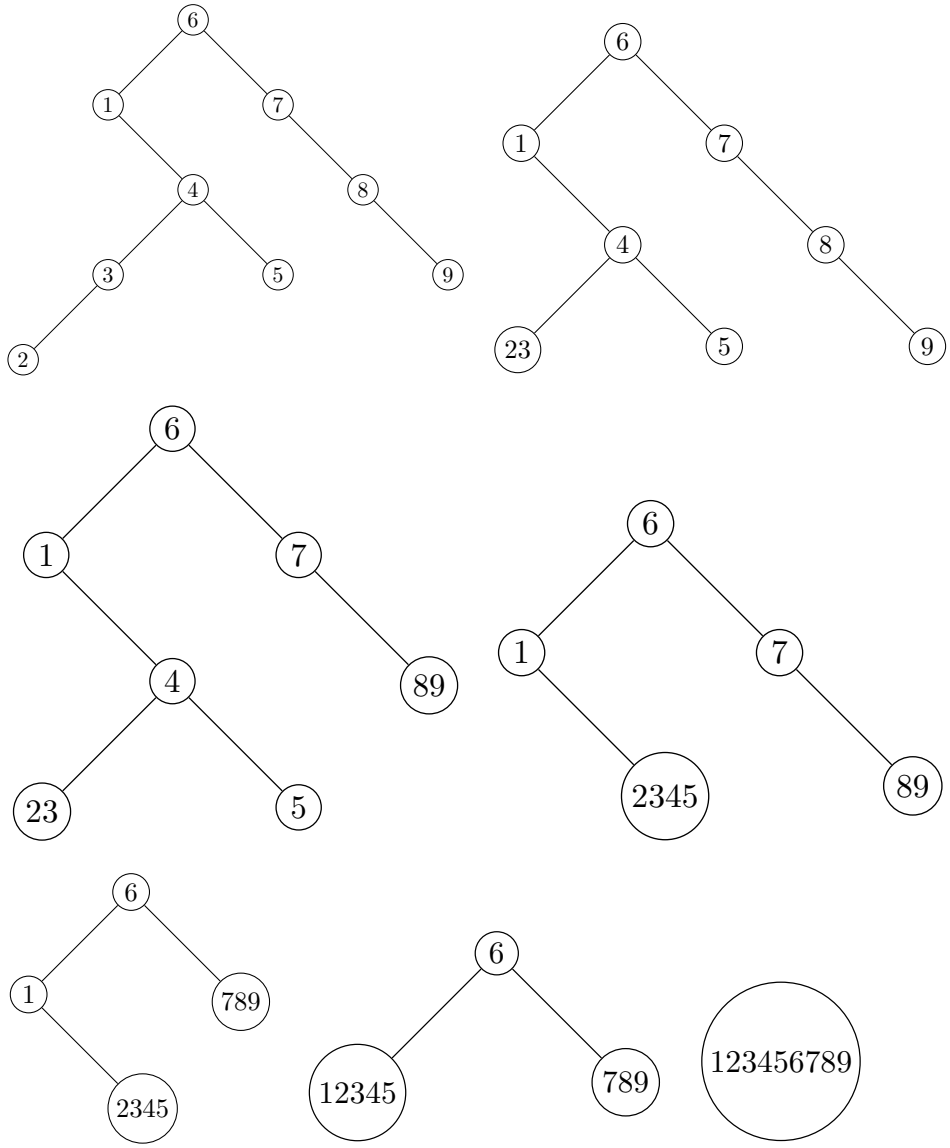
```
1:  procedure QuickSort(a_1, a_2, ..., a_n)
2:      if n = 1 then
3:          return a_1
4:      else
5:          L ← ∅
6:          R ← ∅
7:          for i = 2 to n do
8:              if a_i < a_1 then
9:                  L ← L ∪ {a_i}
10:             else
11:                 R ← R ∪ {a_i}
12:         return (QuickSort(L), a_1, QuickSort(R))
```

Lemma 1. *Låt $a_1, a_2, \ldots, a_n$ vara en lista av distinkta tal. QuickSort($a_1, \ldots, a_n$) returnerar alltid den sorterade listan $(a_{(1)}, a_{(2)}, \ldots, a_{(n)})$.*

[3]Eller genom ungersk folkdans: `https://www.youtube.com/watch?v=ywWBy6J5gz8`



Figur 2: De rekursiva anropen till Quicksort.



Figur 3: Hur den sorterade listan återfås i Quicksort.

Bevis. Vi använder induktion. Basfallet $n = 1$ är uppenbart.

Anta att listan innehåller $n > 1$ tal, och att QuickSort är garanterat korrekt på alla storlekar $k < n$. Vi delar in listan i tre delar: a_1, L och R. Vi har $|L| + |R| = n - 1$, så $|L| < n$ och $|R| < n$. När QuickSort används på L och R garanterar induktionsantagandet att vi får en sorterad lista. Eftersom alla tal i L är mindre än a_1 och alla tal i R är större än a_1, så är listan

$$(\textsc{QuickSort}(L), a_1, \textsc{QuickSort}(R))$$

en sortering av $a_1, \ldots, a_n$. □

Nu blir det väldigt informellt för en stund. Låt oss diskutera hur många jämförelser som Quicksort använder för att sortera en lista. Låt $T(a_1, \ldots, a_n)$ vara antalet jämförelser som krävs för att sortera en lista $(a_1, \ldots, a_n)$ med hjälp av Quicksort. Listan delas in i två listor L av längd ℓ och R av längd r, där $r + \ell = n - 1$. Det krävs $n - 1$ jämförelser för att bygga upp L och R, eftersom a_1 jämförs med $a_2, a_3, \ldots, a_n$. Så vi har (denna rekursion är exakt)

$$T(a_1, \ldots, a_n) = n - 1 + T(L) + T(R).$$

Talet $T(a_1, \ldots, a_n)$ kommer bero mycket på ordningen på listan, men om vi *gissar* att $T(a_1, \ldots, a_n)$ bara beror på längden n, och *har tur* och väljer en pivot a_1 så att $|L| \approx |R| \approx n/2$, så får vi

$$T(n) \approx n + T(n/2) + T(n/2).$$

Rekursionen

$$T(n) = n + 2T(n/2)$$

har lösningen $T(n) = 2n \log_2 n$ (om n är en potens av 2).

En stor mängd steg och formalia har utelämnats här, men i mångt och mycket är det så här det visas att vi kan förvänta oss att $T(n) \approx 2n \log_2 n$, om vi väljer vår pivot slumpmässigt.

I Problem 3 ber jag er visa att om Quicksort appliceras på en sorterad lista $(a_1, \ldots, a_n)$, dvs en där $a_1 < \cdots < a_n$, och första elementet alltid används som pivotelement, så är antalet jämförelser $n(n-1)/2$, vilket väsentligen är lika dåligt som Bubblesort. Detta exempel visar varför det spelar roll vilken typ av lista som matas in, och framförallt hur pivotelementet väljs. I tillämpningar väljer man oftast pivotelementet slumpmässigt. Ibland väljs tre potentiella pivotelement ut slumpmässigt, och medianen av de tre används som pivotelement, för att öka chanserna att vi delar vår lista på mitten. Notera att det krävs extra jämförelser för att bestämma medianen av tre tal.

Ny forskning visar att det förväntade antalet jämförelser i Quicksort kan tas ner från $2n \log n$ till $\frac{9}{5} n \log n$ genom att använda två pivotelement och bryta ner listan i tre grupper, och detta är numera standard i några av de större programmeringsspråken.

6 Grafalgoritmer

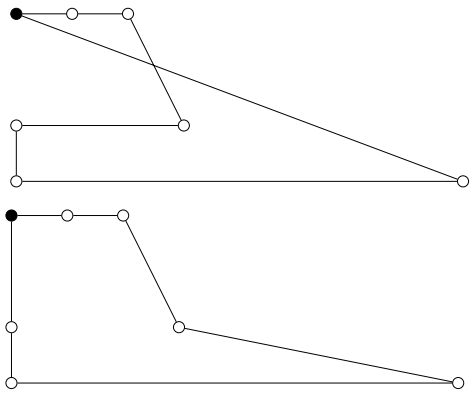
Grafer förekommer ofta inom datavetenskap. Vi har redan sett exempel i form av träden i Figur 2 och 3, där träd får representera strukturen på den rekursion som Quicksort inducerar. Ofta är problemen mer explicita grafproblem. Låt oss börja med att se två situationer där grafer uppstår i optimeringsproblem (så kallad kombinatorisk optimering).

6.1 Handelsresandeproblemet

Handelsresandeproblemet ("Traveling salsesperson problem", "TSP") är ett av de mest berömda optimeringsproblemen, till stor del på grund av dess användbarhet, och till stor del på grund av hur svårt det är att lösa.

Problemformuleringen är enkel: en handelsresande befinner sig på plats 1, och vill besöka plats $2, 3, \ldots, n$. Mellan varje par i, j finns ett avstånd $d_{ij} > 0$. Vad är den kortaste sträckan som krävs för att besöka varje plats precis en gång, och slutligen återvända till plats 1? I Figur 4 ser vi ett exempel. Här är avstånden Euklidiska, det vill säga avstånden mellan noder är precis som de är ritade i bilden. Generellt behöver detta inte vara fallet — vi kan till exempel inte i allmänhet utgå från att triangelolikheten $d_{ij} + d_{jk} \geq d_{ik}$ håller på generella grafer. Figur 4 visar den grundläggande svårigheten med problemet: att göra ett "girigt" val från sin nuvarande situation (det vill säga att välja närmsta oanvända plats) gör ofta att man i slutskedet måste ta väldigt långa steg för att besöka eventuellt obesökta platser. Om platserna är slumpmässigt placerade är den giriga algoritmen hyfsad, med en total reslängd som är ungefär 25% längre än den optimala, men Gutin, Yeo och Zverovich visade år 2002 att det finns (icke-Euklidiska) grafer där den giriga algoritmen ger den sämsta möjliga resvägen.

Stora mängder forskning har ägnats åt problemet sen det introducerades på 1930-talet. År 1972 visade Richard Karp att handelsresandeproblemet är "NP-komplett", vilket innebär att det kanske inte finns en effektiv lösning till problemet (se mer nedan). Mer användbara är istället så kallade approximeringsalgoritmer. Christofides algoritm (från 1976) är en algoritm som hittar en rutt som garanterat är högst 50% längre än den optimala, och gör så inom polynomiell tid, det vill säga antalet steg som algoritmen tar



Figur 4: Två försök att lösa handelsresandeproblemet, med start i den svarta noden. I den första figuren ser vi resultatet av en "girig" algoritm som alltid väljer närmsta tillgängliga grannen, medan figur två visar en smartare lösning.

på n noder är högst n^C för någon konstant $C > 0$. Christofides algoritm utgår bara från att avstånden är "metriska", vilket framförallt betyder att triangelolikheten håller.

6.2 Tilldelningsproblemet

Detta problem kallas "the assignment problem" på engelska, och jag har valt att översätta det till "tilldelningsproblemet".

I detta problem har vi m uppgifter som ska utföras, och $n \geq m$ aktörer som kan utföra uppgifterna. För varje par (i, j) av aktör i och uppgift j associeras en positiv kostnad c_{ij}. Varje aktör kan utföra högst en uppgift, och mer än en aktör kan inte samarbeta för att lösa ett problem. Målet är att alla problem ska lösas samtidigt till en så liten kostnad som möjligt.

Föreställ dig att du har tre containrar i Göteborg som ska fraktas via fartyg till New York, São Paolo och Hong Kong. Det finns sex fraktbolag tillgängliga, som alla erbjuder olika priser för att utföra frakten. Varje bolag kan utföra högst en av leveranserna. Vilka tre ska du välja? Problemet kan representeras med en graf, där varje kant har en kostnad. Låt oss säga att kostnaderna är som i Tabell 1

Vi kan få en girig kandidatlösning genom att ta den billigaste resan till New York (Vandelay), följt av den billigaste resan till São Paolo som använder något annat bolag (Inte nationellt vatten), följt av den billigaste

Bolag	New York	São Paolo	Hong Kong
Frakta i Backarna AB	100	125	500
Vandelay Industries	50	300	350
Frachtung Baby GmbH	200	150	400
Paketfrakt i rakettakt punkt nu	500	500	500
Inte nationellt vatten AB	500	100	300

Tabell 1: Fraktkostnader (kkr) för Exempel 2.

resan till Hong Kong från kvarvarande bolag (Frachtung Baby). Denna kan representeras av en matchning M_{girig} som täcker alla resmål, se Figur 5.

Definition 2. *En* matchning *i en graf är en mängd M av kanter sådana att varje nod i grafen rör högst en kant i M.*

I Figur 5 ser vi också den optimala matchningen M_{opt}. Vi kan skissa en algoritm som börjar med att konstruera M_{girig}, och utifrån den gör förändringar tills M_{opt} är funnen. Idén är att utgå från M_{girig} och göra små förbättringar tills inga förbättringar kan göras längre. Första steget i Figur 5 visar hur M_{girig} kan förbättras via en *cykel*: om FB och INV byter resmål blir den totala kostnaden lägre.

Från M_1 kan vi göra en förbättring: låt FiB ta över São Paolo-jobbet från FB. Detta är förbättring via en *stig*.

I båda fallen hittade vi en mängd F av kanter (antingen en stig eller en cykel) som förbättrade matchningen, och vi kan kort säga att vi får vår nya matchning genom $M_1 = M_{girig} \triangle F$, där $\triangle$ betecknar symmetrisk differens av mängder (alltså $A \triangle B = (A \setminus B) \cup (B \setminus A)$). I vaga termer kan vi sammanfatta algoritmen som i Algoritm 7.

Algorithm 7 Lösning av tilldelningsproblemet med prismatris C

```
1: procedure ASSIGN(C)
2:     M ← girig matchning
3:     while ∃ stig eller cykel F som förbättrar M do
4:         M ← M△F
5:     return M
```

Algoritm 7 kräver att de individuella delarna kan göras effektivt, till exempel att vi snabbt kan avgöra om det finns en mängd F som förbättrar matchningen. Dijkstras algoritm nedan ger en hint om hur detta kan göras.

6.3 Bredden-först-sökning

Bredden-först-sökning ("Breadth-first search", BFS) är en algoritm som används för att utforska en graf, genom att konstruera ett *träd* som innehåller alla grafens noder (vi antar att grafen är sammankopplad). Om BFS:en startar vid en nod v kan algoritmen användas för att beräkna avståndet $d(v, u)$ från v till alla andra noder u. Figur 6 ger ett exempel på hur algoritmen beräknar avstånd från noden a i en viss graf. De tjocka kanterna bildar ett träd, det vill säga en graf som inte innehåller några cykler. Ett sådant här träd, som innehåller alla noder, kallas ett *spanning tree* ("spännande träd"?) och det finns massvis av tillämpningar där det är av intresse att konstruera ett spanning tree utifrån en graf.

Pseudokod till BFS ser ganska komplicerad ut, se Algoritm 8. Att beräkna avstånd är inte strikt talat en del av BFS, och raderna i blått (3 och 13) kan tas bort. En liten detalj: i rad 8 antar vi att grannarna till v ordnas på något sätt (i Figur 6 är de ordnade i bokstavsordning).

Algorithm 8 Bredden-först-sökning på en graf G med start i nod a

```
1:  procedure BFS(G, a)
2:      K ← (a)   (K börjar som en kö som bara innehåller a)
3:      d(a, a) ← 0   (avståndet från a till a är 0)
4:      V ← ∅   (V kommer vara de noder vi behandlat)
5:      T ← ∅   (T kommer vara kanterna i trädet vi bygger)
6:      while K ≠ ∅ do
7:          låt v vara den första noden i K, flytta v från K till V
8:          låt u_1, u_2, ..., u_d vara grannarna till v i G
9:          for i = 1 to d do
10:             if u_i ∉ K ∪ V then   ("om vi inte redan sett u_i")
11:                 lägg u_i sist i K
12:                 lägg till kanten (v, u_i) till T
13:                 d(a, u_i) ← d(a, v) + 1
14:     return T
```

I en graf med n noder och m kanter tar BFS högst $C(m + n)$ för någon konstant $C > 0$.

6.4 Djupet-först-sökning

Det finns en naturlig motpart till BFS som kallas djupet-först-sökning ("Depth-first search", DFS). Vi startar algoritmen i en nod a, och låter $V = \{a\}$ vara

de noder vi har besökt (här börjar vi med a i V till skillnad från BFS). Vi kallar a den *aktiva noden*. Som i BFS skapar vi ett träd T (en mängd av kanter som inledningsvis är tom). Upprepa följande tills alla noder är besökta:

— Om den aktiva noden u har en granne v som inte besökts (alltså inte är i V), lägg kanten uv till T och låt v vara den aktiva noden. Lägg v till V, och säg att u är v:s *förälder*.

— Om alla grannar till den aktiva noden u redan är besökta, gå tillbaka till u:s förälder v och låt v vara aktiv.

Man kan tänka på DFS som en naturligt sätt att komma ur en labyrint (och det var så den först uppfanns på 1800-talet). Börja med att gå så långt du kan i en riktning. Om du kommer till en plats där alla riktningar leder till platser du redan besökt, börja gå tillbaka därifrån du kom tills det finns en ny tillgänglig väg. Figur 7 visar hur DFS ser ut på samma graf som vi använde som exempel för BFS. Notera att det krävs en regel för hur vi väljer nästa aktiva nod när det finns flera val: i exemplet väljer vi noder i bokstavsordning.

6.5 Dijkstras algoritm

Vi använde ovan BFS för att beräkna avstånd från en nod a i en graf till alla andra noder. Här antog vi implicit att avståndet $d(u, v)$ mellan u och v var det minsta antalet kanter på en stig mellan u och v. Om inget annat anges är detta standarddefinitionen av avstånd på en graf.

Vi antar nu att kanterna har olika längd, så att mellan noder u, v finns en kostnad $c_{uv} > 0$. Längden på en stig (en sekvens av sammankopplade kanter) är den totala summan av de involverade kanterna, och avståndet mellan två noder är längden på den kortaste stigen mellan noderna.

För algoritmen vi ska prata om här krävs att avstånden är positiva. I BFS-fallet har vi $c_{uv} = 1$ för varje kant (u, v) i grafen, och $c_{uv} = \infty$ för icke-kanter (u, v).

Principen är samma som för BFS, men uttryckt lite annorlunda. Vi börjar algoritmen med att ge varje nod v ett tillfälligt värde t_v. Om u betecknar noden vi börjar vid sätter vi $t_u = 0$, och $t_v = \infty$ för alla $v \neq u$. Precis som i BFS definierar vi en mängd V av "besökta" noder, som från början är tom.

I varje runda av algoritmen väljer vi den obesökta nod $v \notin V$ som har lägst värde t_v. För varje granne w till v uppdaterar vi värdet t_w via

$$t_w \leftarrow \min\{t_w, t_v + c_{vw}\}.$$

När detta är gjort lägger vi till v till V och går till nästa runda. När algoritmen är färdig är t_v avståndet från u till v.

7 Problem

Varje problem är värt totalt 15p.

1. Visa att BINÄR$(k; k_1, k_2, \ldots, k_n)$ terminerar efter högst $\lceil \log_2 n \rceil$ jämförelser, förslagsvis enligt följande steg:

   (a) Låt $T(n)$ vara antalet jämförelser som krävs (i värsta fallet) om listan har längd n. Vi har $T(1) = 0$. Visa att för $n > 1$ är

   $$T(n) = 1 + T\left(\left\lceil \left(\frac{n}{2}\right) \right\rceil\right). \tag{9}$$

   (b) Visa med induktion att om $T(n)$ uppfyller (9) så är $T(n) \leq \lceil \log_2 n \rceil$ för alla heltal $n \geq 1$.

   OBS: Steg (a) och (b) ovan är ett av många sätt att lösa problemet. Om du kan visa att det krävs högst $\lceil \log_2 n \rceil$ steg på något annat sätt behöver du inte göra (a) och/eller (b).

2. Bubblesort som den presenteras i Algoritm 5 kräver alltid exakt $(n-1)^2$ jämförelser. I beviset av Proposition 2 noterar vi att när $i > 1$ är det garanterat onödigt att utföra steget $j = n$, eftersom vi vet att $a_{(n)}$ redan är det största elementet. På samma sätt är $j = n - 1$ onödigt när $i > 2$, och så vidare.

   Skriv om Algoritm 5 så att den använder exakt $1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$ steg, och fortfarande garanterar en sorterad lista.

3. Låt $(a_1, \ldots, a_n)$ vara en lista av distinkta tal, $n \geq 2$, och låt $T(a_1, \ldots, a_n)$ vara antalet jämförelser som Quicksort använder för att sortera listan, om det första elementet i listan alltid väljs som pivotelement. Visa att om $a_1 < a_2 < \cdots < a_n$, det vill säga vi matar in en sorterad lista, så är

   $$T(a_1, \ldots, a_n) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}.$$

   Du behöver bara visa första likheten. Notera att $T(a_1) = 0$ och $T(\emptyset) = 0$.

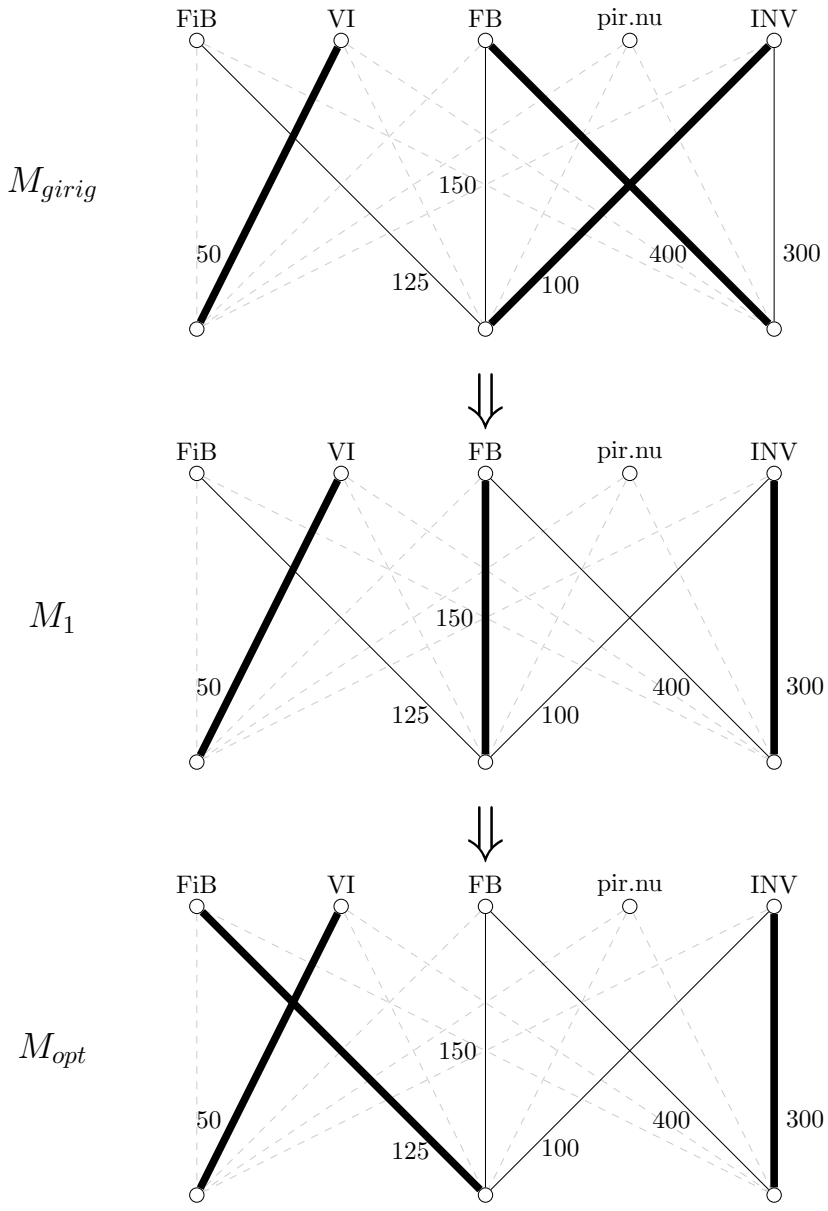
   Detta bevisar att QuickSort ibland kan vara lika dålig som BubbleSort.

   (**Ledtråd:** använd induktion.)

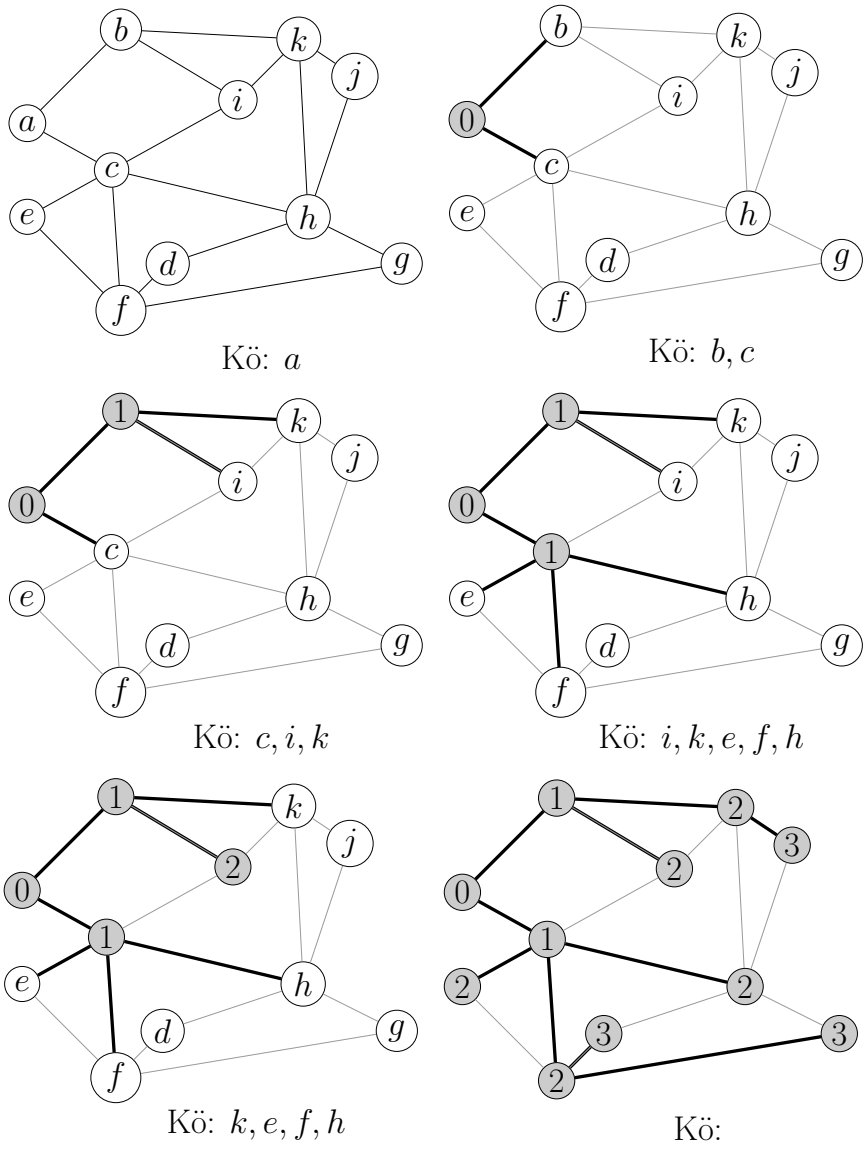
4. I exemplen för BFS och DFS, Figur 6 respektive 7, använder vi bokstavsordning för att bestämma vilken ordning vi väljer noder i. Om vi istället använder omvänd bokstavsordning ("högre" bokstäver först) blir de resulterande träden annorlunda. Gör både BFS och DFS med omvänd bokstavsordning och markera tydligt de kanter som respektive algoritm använder.

   För att vara exakt: i första steget av vår BFS lade vi i första steget de två grannarna b, c till a i kön i ordningen b, c (bokstavsordning). Om vi istället lägger dem i ordningen c, b, hur ser den sista bilden ut? På samma sätt valde vi i DFS att först gå från a till b, men jag vill veta hur det ser ut om vi istället går från a till c och konsekvent fortsätter i omvänd bokstavsordning.

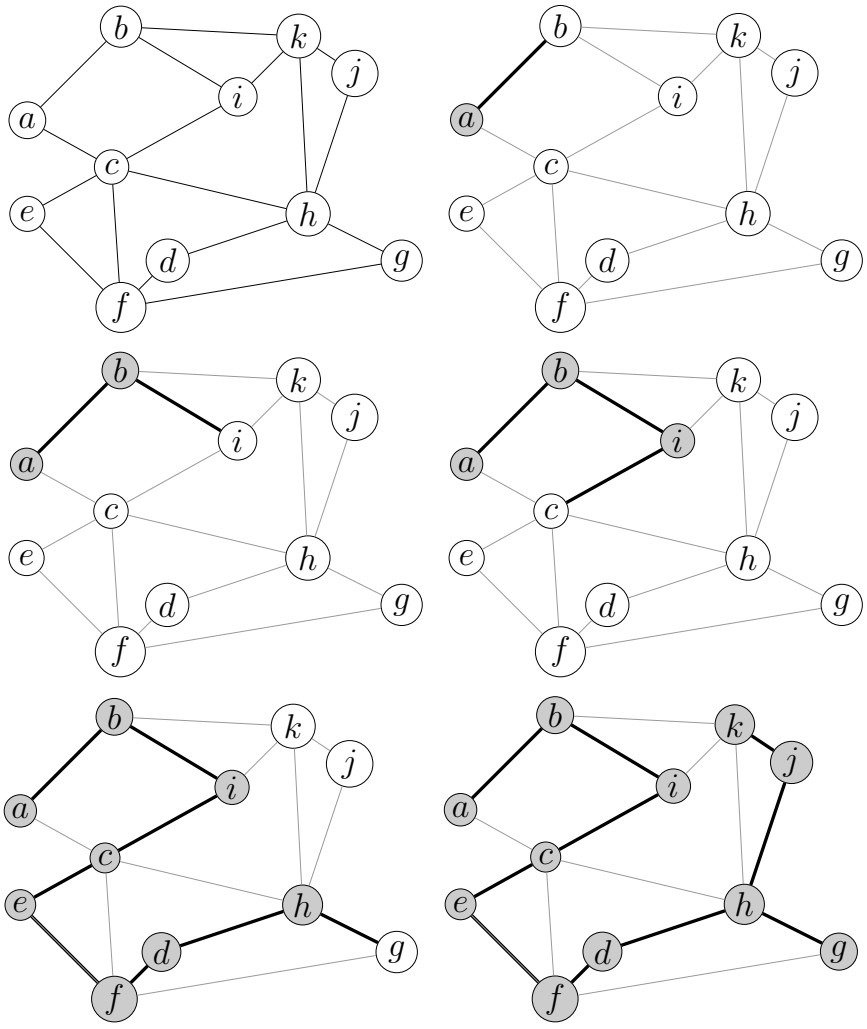
   På kurshemsidan ligger en mall för att rita grafen i LaTeXmed hjälp av paketet tikz, för de som vill använda det.



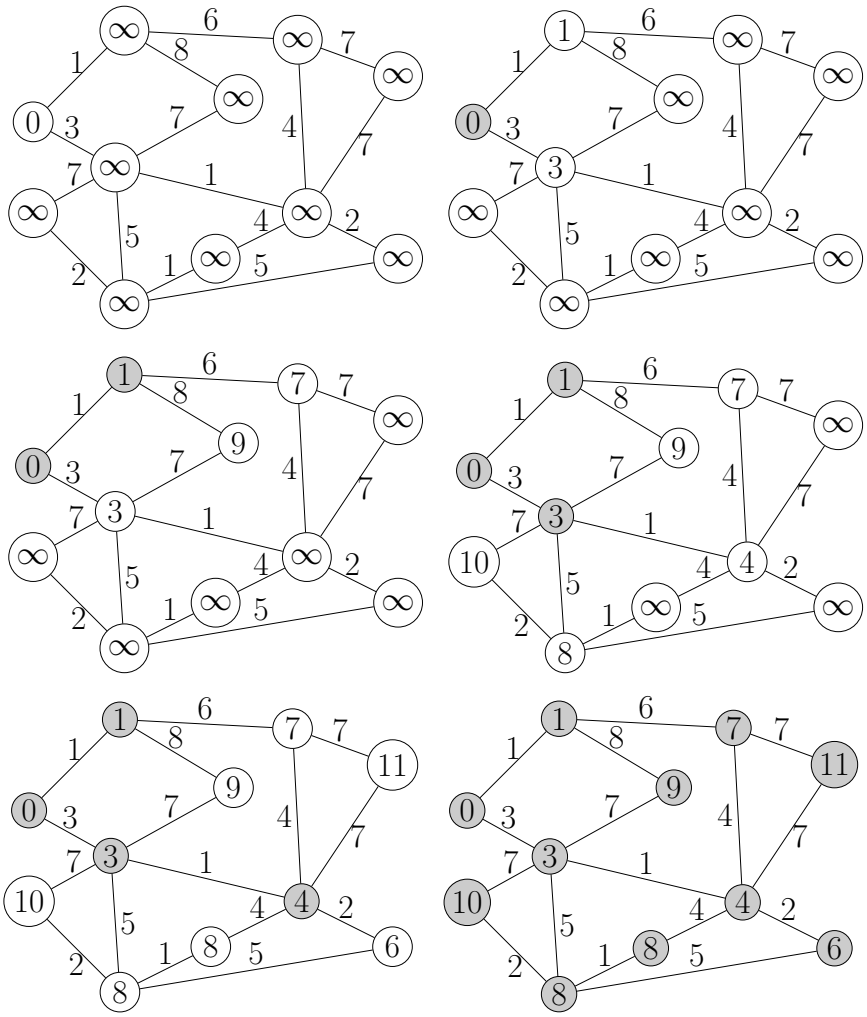
Figur 5: En girig matchning och vägen till den optimala matchningen. Vi döljer de kanter som (med facit i hand) inte påverkar utfallet.



Figur 6: Ett exempel på en bredden-först-sökning med start i nod a. Siffrorna representerar avstånd från a. Grå noder är noder som algoritmen har behandlat.



Figur 7: Ett exempel på en djupet-först-sökning med start i nod a. Grå noder är noder som algoritmen har behandlat. I den näst sista bilden är vi i en situation där den aktiva noden g inte har några obesökta grannar, och vi backar till h.



Figur 8: Ett exempel på Dijkstras algoritm med start i noden a som har värde 0. Siffrorna representerar avstånd från a. Grå noder är noder som algoritmen har behandlat.