# DD1352 - Mästarprov 1

millasvs

September 2016

## 1 Uppgiftsutdelning

En lärare ska dela ut uppgifter till n elever och vill inte att några två elever som känner varandra ska få samma uppgift, eftersom de då kan samarbeta. En algoritm ska skapas som kan avgöra om det räcker med två uppgifter eller inte. Indata är en lista med par av elever som känner varandra, och utdata ska alltså vara ett "Ja" eller "Nej" på frågan om två uppgifter räcker. Frågan kan ställas som följande: om en graf skapas där varje nod är en elev och varje kant mellan två elever innebär att eleverna är bekanta; är då alla komponenter i grafen med $|V| > 2$ (om $|V| \leq 2$ är problemet trivialt) bipartita? Det vill säga, går de att dela upp i två mängder S1 och S2 med egenskaperna att $S1 \cap S2 = \emptyset$ och varje kant $e$ i grafen kan skrivas som $\{x, y\}$ där $x \in S1$ och $y \in S2$? I detta fall skulle S1 svara mot uppgift 1 och S2 uppgift 2. Inga av eleverna i S1 känner varandra och detsamma i S2, vilket innebär att ingen kan samarbeta med någon de känner. Eleverna kan känna någon i den andra gruppen, men det spelar ingen roll då de har en annan uppgift.

Ett sätt att avgöra om en graf är bipartit är att ställa följande fråga: Går det att ordna grafen i blå och röda noder så att ingen kant går mellan två likfärgade noder? Denna fråga kan enkelt besvaras genom att göra bfs (bredd-först-sökning) och färga startnoden blå, alla barnnoder till startnoden röda, alla barnbarn till startnoden blå, o.s.v., generation efter generation. Alla jämna lager är blå och alla udda röda. Sedan undersöks grafen för att se om det finns någon kant som går mellan två olikfärgade noder. I så fall är grafen inte bipartit och vi måste meddela läraren att denne bör göra fler än två uppgifter till sina elever.

I följande pseudokod kommer jag ta bitar från föreläsning 4, där en pseudokod för bredd-först-sökning förekom.

**Uppgiftsutdelning(L)**
**Data:** Lista L med par av elever som känner varandra. Varje elev representeras som en unik siffra $\in [1, n]$.
**Result:** "Ja" eller "nej"
**for** *varje element* $\{x, y\}$ *i* $L$ **do**
  **if** $x \notin V$, *där* $V$ *är en hörnmängd* **then** lägg till $x$ i $V$
  **if** $y \notin V$ **then** lägg till $y$ i $V$
  lägg till kant $e = \{x, y\}$ i kantmängd $E$
**end**
**for** *varje hörn* $u \in V$ **do** $d[u] \leftarrow \infty$
$s \leftarrow$ slumpmässigt vald nod från $V$
**return** BFS($V$,$E$,$s$)

**BFS(V, E, s)**
**Data:** Hörnmängd V, kantmängd E och startnod s
**Result:** "Ja" eller "nej"
skapar kantmängd $E2$;
$d[s] \leftarrow 0$;
$color[s] \leftarrow$ blå;
$Q \leftarrow \{s\}$;
**while** $Q \neq \emptyset$ **do**
  $u \leftarrow DEQUEUE(Q)$;
  tar bort $u$ från $V$;
  **for** *varje granne* $v$ *till* $u$ **do**
    **if** $d[v] = \infty$ **then**
      $d[v] \leftarrow d[u] + 1$;
      lägger till $e = \{u, v\}$ i $E2$;
      $ENQUEUE(Q, v)$;
    **end**
  **end**
  **if** $d[u] \bmod 2 = 0$ **then** $color(u) \leftarrow$ blå;
  **else** $color[u] \leftarrow$ röd;
**end**
**for** *varje kant* $e = \{x, y\}$ *i* $E2$ **do**
  `// om en kant går mellan två likfärgade noder`
  **if** $color[x] = color[y]$ **then return** NEJ;
**end**
`// fler komponenter med` $|V| > 2$ `att kolla bipartithet hos`
**if** $|V| > 2$ **then**
  $v \leftarrow$ slumpmässigt vald nod från $V$;
  BFS($V$, $E$, $v$);
**end**
**return** JA;

Det som görs här i algoritmen är att vi tar listan av eleverna som alla är re-

presenterade som heltal mellan 1 och n, gör om dem till en graf med varje elev representerad som en nod och vänskapsförhållande som kant, väljer en slumpmässig nod av dessa och gör bfs i grafen. Skapar en lista *color* med n element där color[x] = färgen för noden som korresponderar för elev x. Färgar noder i jämna lager blå och udda lager röda. Skapar en kantmängd av alla noder som gåtts igenom i denna bfs-iterationen. Det kan vara alla kanter, om grafen är sammanhängande, eller så är det bara kanterna hos en komponent som undersökts. Anledningen till att denna mängd skapas är för att jämförelsen mellan noderna ska kunna genomföras, och också för att slippa gå igenom alla kanter i onödan. Alla kanter i denna mängd gås igenom, och om en kant går mellan två likfärgade noder returnerar algoritmen ett nej, för då har vi att det finns åtminstone en komponent i grafen som inte är bipartit. Varje gång en nod färgas så tas den ur nodmängden V, vilket gör att V minskar efterhand och att rekursionen så småningom slutar. Om ingen av komponenterna kuggar bipartitstestet så returnerar vi ett ja.

Komplexiteten av bredd-först-sökning är $O(|V|+|E|)$, där $|V|$ är antalet hörn och $|E|$ är antalet kanter i grafen. Bredd-först-sökningen genomförs flera gånger om det är flera komponenter som gås igenom. För varje graf kommer alltså $|V|$ och $|E|$ variera, men tillsammans kommer de bilda den ovannämnda komplexiteten. I algoritmen ovan tillkommer också en lista *color* där varje nod sätts till antingen röd eller blå. Detta ger en till jämförelse och tilldelning när noderna gås igenom. Det kommer dock inte ändra den asymptotiska tidskomplexiteten. I denna uppgift gäller att $|V|$ som högst är $n$, alltså antalet studenter. Detta är om man förutsätter att alla elever förekommer i listan, det vill säga att alla känner någon annan i klassen. $|E|$ kan variera, men har en övre gräns på $\frac{n(n-1)}{2}$, som gäller om alla elever känner varandra. Innan bfs tillkommer konverteringen från listan med par av elever till graf, i vilken man går igenom alla element i listan och lägger till en, två eller noll hörn i grafen och en kant. Dessa operationers tidskomplexitet varierar beroende på vilken datastruktur man väljer att representera grafen med, men de är konstanta med både grannmatris och grannlista, förutsatt att man vet dimensionen för grannmatrisen från början och därför slipper allokera en ny matris varje gång man lägger till en nod. I detta fall skulle man kunna ha dimensionen n*n eftersom det är den övre gränsen. Eftersom detta händer för alla element i listan, d.v.s. $|E| \in O(\frac{n(n-1)}{2})$ gånger, så blir komplexiteten $O\left(n + \frac{n(n-1)}{2} + \frac{n(n-1)}{2}\right) \in O(n^2)$. Algoritmen är alltså polynomisk, vilket var ett krav för uppgiften.

# 2 Optimal Pokémonvandring med jaktfri ruvning

En person vi kallar Arrietty ska fånga pokémon. Hon har $n_2$ ägg som kräver 2 km vandring för att ruvas, $n_5$ ägg som kräver 5 km vandring för att ruvas och $n_{10}$ ägg som kräver 10 km vandring för att ruvas. Hon ska gå en sträcka som är $2n_2 + 5n_5 + 10n_{10}$ km lång och hon kommer bara fånga pokémon när hon

stannar för att lägga ett nytt ägg i inkubatorn. Varje ställe hon kan stanna på har ett visst antal pokéstopp i närheten, vid vilka det finns pokémon för henne att fånga. Frågan är i vilken ordning hon ska ruva sina ägg för att kunna fånga så många pokémon som möjligt. Svaret ska anges i antal pokéstopp hon kommer stanna vid. Antal pokéstopp som nås från en viss plats ges av en array $P[2...len]$ där $len = 2n_2 + 5n_5 + 10n_{10}$.

Antal sätt som Arrietty kan ordna sina ägg kan lösas med hjälp av kombinatorik: $\frac{(n_2+n_5+n_{10})!}{n_2!n_5!\cdot n_{10}!}$. Om vi säger att $n_2 = 2, n_5 = 1$ och $n_{10} = 3$ kommer antal sätt på vilket Arrietty kan ordna äggen bli $\frac{6!}{2!\cdot 1!\cdot 3!} = 60$. Eftersom många av dessa sätt att ordna äggen har delar av ordningen gemensamt så vore det slösaktigt att gå igenom alla sätten att ordna äggen på. Man kan istället formulera problemet som en rekursion $sum(x) = P[x] + max(sum(x-10), sum(x-5), sum(x-2))$ där man förstås bara gör de rekursiva anropen som man har råd med, det vill säga som man har ägg för. Man anropar bara $sum(x-2)$ om man har minst ett 2-kilometersägg kvar, till exempel.

Frågan är nu om man kan lösa problemet utan att ha en algoritm som anropar sig själv och istället lagra delresultaten, så man slipper beräkna samma sak flera gånger. Ett sätt man skulle kunna göra det på är att ha en tvådimensionell matris som man vandrar igenom genom att antingen gå till höger, vilket är samma som att gå vidare utan att spendera ett ägg, eller neråt, vilket innebär att man spenderar ett ägg. En cell i matrisen på rad $i$ och kolumn $j$ skulle då beskriva det maximala antal pokéstopp man når efter att ha stannat $i$ gånger och vandrat $j$ km. Basfallen skulle vara 1) om man inte stannat någon gång, 2) om man gått mindre än 2 km, eller 3) om man bara stannat en gång. I de två förstnämnda fallen kan man inte ha fångat några pokémon, eftersom 1) Arrietty fångar bara pokémon när hon stannat, inte medan hon går och 2) Arrietty stannar inte förrän hon gått minst 2 km. I nummer 3, det vill säga då i = 1, vilket svarar mot rad 1 i matrisen, så kommer antalet pokéstopp hon kan nå stämma mot arrayen P. Detta eftersom det är första gången hon stannar, varför hon inte kan ha ackumulerat några pokéstopp sen tidigare. Egentligen skulle man kunna låta bli att ha kolumn 0 och 1 i matrisen eftersom de värdena ändå bara kommer ge 0. Rad 0 skulle man också kunna låta bli att ha. Nu kommer jag ändå ha kvar dem eftersom indexnumrena då stämmer mot vad de representerar; det vill säga antalet km hon gått och antal gånger hon stannat. $i$ skulle i denna representation av problemet ha en övre gräns på $n_2 + n_5 + n_{10}$ eftersom det är antalet gånger hon stannar under promenaden och $j$ kommer ha en övre gräns på $2n_2 + 5n_5 + 10n_{10}$ eftersom det är längden på hennes promenad. Det maximala antal pokéstopp hon kan nå kommer då ges längst ner till höger i matrisen, då hon har gått hela promenaden och stannat de gånger hon behövt. Matrisen beräknas med fördel från vänster till höger uppifrån och ner då man kan använda sig av tidigare beräknade värden när man beräknar den nuvarande cellen. Ett delproblem kan skrivas som följande:

$$sum[i,j] = \begin{cases} 0 & \text{om } j < 2 \quad \text{eller } i < 1 \\ P[j] & \text{om } i = 1, j \geq 2 \\ P[j] + sum[i-1, j-2] & \text{om } i \geq 2, 2 \leq j < 5 \\ P[j] + max(sum[i-1, j-5], & \\ sum[i-1, j-2]) & \text{om } i \geq 2, 5 \leq j < 10 \\ P[j] + max(sum[i-1, j-10], & \\ sum[i-1, j-5], sum[i-1, j-2]) & \text{om } i \geq 2, j \geq 10 \end{cases}$$

och algoritmen skulle då kunna se ut så här:

**Pokémonvandring**
**Data:** $n_2, n_5, n_{10}$ och $P[2...2n_2 + 5n_5 + 10n_{10}]$
**Result:** Maximalt antal pokéstopp som kan nås under vandringen
$len \leftarrow 2n_2 + 5n_5 + 10n_{10}$
$numOfStops \leftarrow n_2 + n_5 + n_{10}$
**for** $j \leftarrow 2$ *to* $len$ **do**
  $sum[1, j] \leftarrow P[j]$
**end**
**for** $i \leftarrow 1$ *to* $numOfStops$ **do**
  $sum[i, 0] \leftarrow 0$
  $sum[i, 1] \leftarrow 0$
**end**
**assertion** Basfallen beräknade enligt rekursionen
**for** $i \leftarrow 2$ *to* $numOfStops$ **do**
  **invariant** $sum[k, j]$ beräknad enligt rekursionen
  för $2 \leq k < i$ och $2 \leq j \leq len$
  **for** $j \leftarrow 2$ *to* $len$ **do**
    $p \leftarrow P[j]$
    **if** $j < 10$ **then**
      **if** $j < 5$ **then**
        $sum[i, j] \leftarrow p + sum[i-1, j-2]$
      **end**
      $sum[i, j] \leftarrow p + max(sum[i-1, j-2], sum[i-1, j-5])$
    **end**
    **else**
      $p2 \leftarrow max(sum[i-1, j-2], sum[i-1, j-5])$
      $p2 \leftarrow max(p2, sum[i-1, j-10]$
      $sum[i, j] \leftarrow p + p2$
    **end**
  **end**
**end**
**return** $sum[numOfStops, len]$

Rekursionen motiverades ovan och algoritmen innehåller en assertion och en invariant som visar att algoritmen implementerar rekursionen på ett korrekt sätt.

Denna algoritm har tidskomplexitet $O((n_2+n_5+n_{10}-2)(2n_2+5n_5+10n_{10}-2))$ eftersom det som högst finns två nästlade for-loopar med de respektive längderna, i vilka det bara förekommer konstanta operationer såsom jämförelser och tilldelningar. Eftersom konstanta termer inte har någon signifikans i ordonotation så kan man skriva det som $O((n_2 + n_5 + n_{10})^2)$ eller $O(n^2)$ om man definierar $n$ som $n_2+n_5+n_{10}$. Den är alltså polynomisk. Den skulle kunna göras mer effektiv genom att inte räkna delkostnaderna för alla celler. Till exempel kommer inga celler i kolumn 0, 1 eller 3 vara intressanta, eller andra kolumner som inte kan nås med de ägg man har.

# 3 Boka klassrum optimalt

Det här problemet handlar om en skola där det finns $k$ stycken klassrum och $n$ stycken lektioner ska bokas i dessa klassrum inför kommande termin. Målet här är att så många lektioner som möjligt ska bokas i ett klassrum, så att inte mer än en lektion har bokat samma klassrum samtidigt. Indata är alltså k samt n st lektioner i form av par $(s_i, f_i)$ där $s_i$ är lektion i:s starttid och $f_i$ lektion i:s sluttid. Utdata ska vara en lista för varje klassrum med vilka lektioner som i tidsordning ska bokas i det rummet.

Problem som dessa - där man ska passa in så många aktiviteter som möjligt i en resurs - kan kallas aktivitetsproblemet, och det har en välkänd girig lösning. I kapitel 4 i boken *Algorithm Design* av Kleinberg och Tardos (2005) tar de upp den här algoritmen i detalj, som går ut på att man sorterar aktiviteterna efter deras sluttider och väljer dem i turordning. Man väljer hela tiden den aktivitet som slutar först och raderar samtidigt alla aktiviteter som är parallella med aktiviteten man just valt. I detta problem är det dock inte längre optimalt att sortera alla lektioner efter sluttid. Den logik man använder när man hela tiden prioriterar den lektion som slutar tidigast (det vill säga när man resonerar att man då kan få med så många lektioner som möjligt sedan) är inte aktuell här eftersom vi inte kan avvisa lektioner - alla ska bokas.

I samma kapitel i *Algorithm Design* tar de också upp en algoritm som snarare handlar om att man har ett visst antal intervall (lektioner) och vill passa in dem i så få resurser (klassrum) som möjligt. Den algoritmen bygger på att man istället sorterar aktiviteterna efter ickeavtagande starttid, för att sedan indexera aktiviteterna (boka lektionerna i klassrum) i turordning, och öka indexet (allokera ett nytt klassrum) när man stöter på en aktivitet som krockar och som alltså inte kan ha samma index (klassrum). Det visar sig då att antalet resurser som behövs är minst lika stort som "djupet" av intervallen - i detta fall det högsta antalet lektioner som pågår samtidigt. Den här algoritmen tycks passa det här problemet bättre, eftersom vi som sagt kommer behöva boka alla klassrum, och vill helst behöva allokera så få klassrum som möjligt (allra helst inte fler än $k$ stycken, så rektorn slipper boka fler lokaler). Eftersom vi bara har

$k$ stycken klassrum så kan vi inte allokera fler när vi redan allokerat $k$ stycken. Utdatan presenteras som en lista M som som består av k element, där varje element i M är en egen lista med lektioner. M står då för mängden av klassrum, och varje element i M står för schemat hos motsvarande klassrum. En indexerad prioritetskö används för att underlätta valet av klassrum för lektionen man tittar på, genom att hela tiden prioritera det klassrum som är ledigt tidigast, samtidigt som man går igenom lektionerna i ordning, med de som börjar tidigast först.

**Klassrumsbokning**
**Data:** $k$, $L[(s_1, f_1), ..., (s_n, f_n)]$, där $k$ är antalet klassrum och $(s_i, f_i)$ är start- och sluttiden för lektion $i$
**Result:** En lista med $k$ element, där varje element $M[k]$ är en lista av lektionerna bokade i klassrum $k$, sorterade i tidsordning

```
skapar ny, tom lista M med k celler
for i ← 1 till k do
    skapar ny länkad lista w
    M[i].add(w)      // M består efter loopen av k länkade listor
end
sortera L[(s_1, f_1), ..., (s_n, f_n)] efter icke-avtagande starttid med mergesort
(s_1, f_1) = L[1]              // (antar att listan börjar på index 1)
M[1].add((s_1, f_1))               // Första lektionen bokas i första
 klassrummet
minPQ.insert(1, f_1)       // Klassrum 1 läggs i prioritetskön med
 [den första tillgängliga tiden] = [tiden då lektion 1 tar
 slut]
klassrumsnr ← 1
for i ← 2 till n do
    (s_i, f_i) ← L[i]
    j ← minPQ.minIndex()      // j är det tidigast tillgängliga
     klassrummet
    if s_i ≥ minPQ.keyOf(j) then
        M[j].add(L[i])             // bokar lektion i till klassrum j
        minPQ.increaseKey(j, f_i)
    end
    else
        // kollision: behöver allokera nästa klassrum
        // allokerar dock inte fler än k klassrum
        if klassrumsnr < k then
            klassrumsnr ← klassrumsnr + 1
            M[klassrumsnr].add(L[i])
            minPQ.insert(klassrumsnr, f_i)
        end
    end
end
return M
```

För att denna algoritm ska anses vara korrekt måste den uppfylla följande: 1) Inga lektioner får överlappa varandra i samma klassrum 2) Så många lektioner som möjligt tilldelas ett klassrum. För att bevisa att 1) uppfylls observerar vi två lektioner $i$ och $i + 1$ som överlappar varandra. Lektion $i + 1$ börjar senare än lektion $i$ men tidigare än vad lektion $i$ slutar ($=f_i$). Lektion $i$ bokas till ett klassrum. När det sedan är dags för lektion $i + 1$ att bokas säkerställs först att

lektionen börjar på en tid som klassrummet är ledigt. Eftersom klassrummet först är ledigt $f_i$ kommer lektion $i+1$ vara uteslutet. Alltså är krav 1) uppfyllt.

Om vi istället kollar på krav 2) så är det så att det här är en algoritm som används för att hitta så få klassrum som möjligt för att tillgodogöra alla lektioner, vilket jag ovan resonerade även kommer uppfylla att så många lektioner som möjligt får ett klassrum. Anledningen till att den här algoritmen faktiskt minimerar antal klassrum som behövs är att man för varje klassrumsbokning tar den lektion som börjar tidigast och jämför den med det klassrum som är ledig tidigast. På detta sätt allokerar man bara ett nytt klassrum när det verkligen behövs, d.v.s. när inte ens det klassrum som är ledigt tidigast kan tillgodogöra den tidigaste lektionen. Alltså kommer algoritmen bara allokera nya klassrum när två lektioner krockar, och antalet klassrum som allokeras blir lika stor som djupet av intervallen. Färre klassrum än så är det omöjligt att allokera.

Tidskomplexiteten är O($n$ log $n$) för sorteringen, där mergesort används. För en komplexitetsanalys av mergesort hänvisar jag till samma bok som tidigare - *Algorithm Design*, kapitel 5.1. Sedan kommer en for-loop som går $n-2$ gånger, i vilken det görs vissa tilldelningar och jämförelser, insättningar i M-listan (vilket är en konstant operation), en uttagning ur en min-prioritetskö och en uppdatering av eller en insättning i samma prioritetskö. Komplexiteten för att ta ut något från en prioritetskö med m element implementerad med en heap är O(1), att uppdatera värdet av en nyckel är O(log m) och insättning av en nyckel är också O(log m). Eftersom det som högst kan vara $k$ element i prioritetskön blir det alltså O(log k). Alltså är komplexiteten av denna for-loop O($n$ log $k$), och den totala komplexiteten för algoritmen blir O($n$ log $n$ + $n$ log $k$) $\in$ O($n$ log $n$). Frågan kräver att lösningen är optimal, varför jag måste visa att det inte finns en lösning bättre än $n$ log $n$ för det här problemet. Detta kräver att algoritmen har denna gräns som både övre och undre gräns, det vill säga den tar åtminstone $n$ log $n$ tid (O), men kan inte heller gå snabbare än så ($\Omega$).

För det första: för att lösa problemet krävs att vi tittar på alla lektioner och deras start- och sluttider. Alltså har vi åtminstone en undre gräns på n. I sorteringsproblemet kunde man bevisa att den undre gränsen är O(n log n) eftersom det finns n! stycken permutationer av listan och varje gång kan man som högst halvera möjligheterna genom en jämförelse mellan två element och därför blir tidskomplexiteten log n! = O(n log n). I detta problem krävs också mer än att man bara går igenom lektionerna en efter en. Precis som i sorteringsproblemet finns ett visst antal permutationer. Kanske inte n!, eftersom vi inte har en lång lista, vi har k stycken kortare. Antal permutationer här är lite svårare att komma fram till, eftersom vi inte vet längden på listorna. Det kan finnas olika många i alla. Summan av alla element i listorna behöver inte ens vara n (eftersom vissa lektioner kanske inte blir bokade). Om jag skulle resonera om antalet permutationer skulle jag säga att det är [antal sätt att ordna n element] + [antal sätt att dela upp n element i k listor] = $\binom{n+k-1}{k-1} + n! \in O(n!)$. Tänk att man har n st lektioner framför sig, och först tittar man på alla sätt att ordna dem och sedan ska man dessutom hitta alla sätt att placera ut k-1 stycken pinnar för att avskilja dem i olika klassrum. Till skillnad från i sorteringsproblemet där man halverar antalet möjligheter vid varje jämförelse så krävs det här två jämförelser

innan man kan halvera antalet möjligheter - man måste jämföra både start- och sluttid. Detta ändrar dock inte asymptotisk tidskomplexitet, varför vi har $\Omega(n \log n)$.