Sjekkliste

Sjekklisten er basert på kap. 10 i «The Algorithm Design Manual» av Steven S. Skiena.

A. Forstår jeg virkelig problemet?

B. Kan jeg finne en enkel algoritme?

C. Kan jeg bruke en pensum-algoritme?

D. Kan jeg løse spesialtilfeller?

E. Hvilke designmetoder er mest relevante?

A. Forstår jeg virkelig problemet?

B. Kan jeg finne en enkel algoritme?

C. Kan jeg bruke en pensum-algoritme?

D. Kan jeg løse spesialtilfeller?

E. Hvilke designmetoder er mest relevante?

- Hva består input av, helt presist?
- Hva er ønsket output, helt presist?
- Kan jeg konstruere et input-eksempel som jeg kan løse for hånd? Hva skjer da?
- Hva slags strukturer har jeg å jobbe med? Sekvenser, strenger, grafer, tall, mengder…?

A. Forstår jeg virkelig problemet?

B. Kan jeg finne en enkel algoritme?

C. Kan jeg bruke en pensum-algoritme?

D. Kan jeg løse spesialtilfeller?

E. Hvilke designmetoder er mest relevante?

- Kan jeg finne en korrekt *brute force*-løsning?
  - Hvordan vet jeg at den er korrekt?
- Kan jeg bruke en enkel regel gjentatte ganger? Alltid velg den største/minste?
  - Når fungerer det, og når fungerer det ikke?

A. Forstår jeg virkelig problemet?

B. Kan jeg finne en enkel algoritme?

C. Kan jeg bruke en pensum-algoritme?

D. Kan jeg løse spesialtilfeller?

E. Hvilke designmetoder er mest relevante?

- Finnes det en algoritme som passer helt?
- Finnes det noen som *nesten* passer?
- Hva slags reduksjoner kan jeg få til?
- Jobb systematisk med de algoritmene du kan!

A. Forstår jeg virkelig problemet?

B. Kan jeg finne en enkel algoritme?

C. Kan jeg bruke en pensum-algoritme?

D. Kan jeg løse spesialtilfeller?

E. Hvilke designmetoder er mest relevante?

- Hva om jeg ignorerer deler av problemet?
- Hva om jeg setter noen parametre til trivielle verdier som 0 eller 1?
- Kan jeg forenkle problemet til noe jeg *kan* løse effektivt?
- Hvorfor kan ikke denne spesial-løsningen generaliseres?
- Er problemet et spesialtilfelle av noe kjent?

A. Forstår jeg virkelig problemet?

B. Kan jeg finne en enkel algoritme?

C. Kan jeg bruke en pensum-algoritme?

D. Kan jeg løse spesialtilfeller?

E. Hvilke designmetoder er mest relevante?

- Hjelper det å sortere?
- Kan jeg dele i to, kanskje med binærsøk? Store/små, Høyre/venstre? Splitt-og-hersk?
- Kan jeg finne en rekkefølge på elementer/delproblemer? Kan jeg bruke det til DP?
- Noe som gjøres ofte (f.eks. søk)? Kan jeg bruke en datastruktur til speed-up? Søketre/hashtabell/heap?
- Kan jeg formuelere problemet som et lineært program?
- Ligner problemet på et NPC-/NPH-problem? Kan jeg redusere fra et slikt problem?

A. Forstår jeg virkelig problemet?

B. Kan jeg finne en enkel algoritme?

C. Kan jeg bruke en pensum-algoritme?

D. Kan jeg løse spesialtilfeller?

E. Hvilke designmetoder er mest relevante?

F. Ikke få panikk!

Antagelig ikke forelest i dag, men jeg tar det med her, så dere kan lese på det :-)

Algdat-ninja på 60 minutter:
Et galskapsprosjekt

Magnus Lie Hetland

15. november, 2002

Advarsel: Tettpakkede og overfladiske foiler forut!

Algtdat i 6 punkter

1. Grunnbegreper og basisverktøy
2. Rekursjon og induksjon
3. Graftraversering og avhengighetsgrafer
4. Splitt og hersk
5. Dynamisk programmering og grådighet
6. Iterative algoritmer

1. Grunnbegreper og basisverktøy

- Kjøretid, Algoritmer og Problemer
- Problemklasser: P, NP, og NPC
- Prioritetskøer
- Oppslagstabeller

Kjøretid, Algoritmer og Problemer

En algoritme løser et problem. Kjøretiden til algoritmen er avhengig av problem-instansen. Forskjellige instans-typer kan gi forskjellige kjøretider (best-case, worst-case, average-case). Gitt en instans-type er kjøretiden kun avhengig av problemstørrelse. Vi skriver da $T(n)$.

Vi kan beskrive $T(n)$ med en grov øvre (O) eller nedre (Ω) grense. For eksempel kan vi skrive

$$T(n) \in O(n^2)$$

hvis n^2 er en øvre grense for $T(n)$. Θ uttrykker både øvre og nedre grense samtidig.

Merk: Ingen direkte sammenheng mellom f.eks. worst-case og O. Likevel: Hvis vi beskriver en vilkårlig problem-instans må den øvre grensen gjelde også for worst-case. (Tilsvarende for Ω og Θ.)

Spesialtilfelle: Når vi snakker om Quicksort er det ofte implisitt at vi snakker om average-case ($O(n \log n)$) og ikke worst-case ($O(n^2)$).

Problemklasser: P, NP, og NPC

P, NP og NPC beskriver problemklasser, ikke algoritmer.

$P \subseteq NP$ og $NPC \subseteq NP$.

P-problemer kan løses i polynomisk tid (P for Polynomial).

Løsningen til et NP-problem kan testes i polynomisk tid (N for Nondeterministic).

En eventuell løsning på et NPC-problem kan brukes til å løse et hvilket som helst NP-problem med et polynomisk kjøretidstillegg (C for Complete).

Altså: Hvis $NPC \subseteq P$ så $P = NP$. Med andre ord, hvis et NPC-problem kan løses i polynomisk tid, så kan alle NP-problemer løses i polynomisk tid.

Er $P = NP$? Vi vet ikke, men det virker svært usannsynlig.

Prioritetskøer

Man kan effektivt finne/fjerne minste (evt. største) element i en prioritetskø. En implementasjon er hauger (heaper). Den lar deg sette inn elementer og fjerne minste element i $O(\log n)$ tid.

Haug-egenskapen: Foreldre er alltid mindre (evt. større) enn sine barn. Dette er mindre strengt enn enn søketre-egenskaper, og dermed billigere å opprettholde.

Enkle søketrær gir også samme kjøretid for average-case, men kan lett bli ubalanserte, og har $O(n)$ som worst-case-kjøretid for extract-min. Heaper er automatisk balanserte.

Oppslagstabeller

Oppslagstabeller (dictionaries, mappings) lar deg knytte en vilkårlig nøkkel til en vilkårlig verdi, som f.eks. i en telefonkatalog. En implementasjon er hashtabeller.

En hashtabell baserer seg på en hashfunksjon, som fra en nøkkel k beregner en indeks $h(k)$ i en vanlig tabell (array). Det er ikke plass til alle mulige nøkler i denne tabellen, så det kan oppstå kollisjoner (nøkler med samme hashverdi). Hashfunksjonen er konstruert for å unngå dette i størst mulig grad, men hash-algoritmen må likevel kunne takle kollisjoner.

Mulig løsning: Ved kollisjon, lagre alle nøkler med samme hash-verdi i en liste på den angitte posisjonen i hash-tabellen.

Hashfunksjoner beregner ofte tilsynelatende tilfeldige hashverdier, men for f.eks. heltallsnøkler kan det fungere med $h(k) = k \bmod n$, der n er antall plasser i tabellen.

2. Rekursjon og induksjon

- Enkel induksjon
- Enkel rekursjon
- Rekurrensligninger og iterasjonsmetoden

Enkel induksjon

Vil bevise p_i for alle $i \in \mathbb{N}$.

1. Bevis p_1.

2. Bevis $p_{i-1} \Rightarrow p_i$ for alle $i > 1$.

Og det var det ...

For en vilkårlig i har vi da:

$$p_1 \Rightarrow p_2 \Rightarrow \cdots \Rightarrow p_{i-2} \Rightarrow p_{i-1} \Rightarrow p_i$$

Enkel rekursjon

Rekursjon: En funksjon definert (direkte eller indirekte) ut fra seg selv.

Eksempel: En funksjon som beregner $\sum_{i=1}^{n} i$.

$$f(1) = 1$$
$$f(n) = f(n-1) + n$$

Utsagnet p_n blir her at $f(n)$ er korrekt.

1. Vi ser at p_1 holder (summen av 1 er 1).

2. Hvis p_{n-1} holder så er $f(n-1) = \sum_{i=1}^{n-1}$. Vi har da at $f(n) = f(n-1) + n = \sum_{i=1}^{n}$, dvs. at p_n holder.

Dermed er $f(n)$ korrekt for alle n.

Rekursjon er (mer eller mindre) omvendt induksjon.

Rekurrensligninger og iterasjonsmetoden

Kjøretider kan ofte være definert rekursivt, fordi algoritmen vi beskriver er rekursiv. F.eks. (binærsøk):

$$T(1) = \Theta(1)$$
$$T(n) = T(n/2) + \Theta(1)$$

Enkleste løsningsmetode (ikke i læreboka): Iterativ substitusjon. Kjør de "rekursive kallene" selv, og let etter et mønster:

$$\begin{aligned} T(n) &= T(n/2) + \Theta(1) \\ &= (T((n/2)/2) + \Theta(1)) + \Theta(1) \\ &= T(n/4) + 2 \cdot \Theta(1) \\ &= T(n/8) + 3 \cdot \Theta(1) \\ &\vdots \\ &= T(n/2^i) + \Theta(i) \end{aligned}$$

Variabelen i kan velges fritt. Vi velger den som lar oss sette inn $T(1)$, dvs. rekursjonsdybden. Vi setter $n/2^i = 1$ og får $i = \log_2 n$. Med andre ord, $T(n) = 1 + \log_2 n \cdot \Theta(1) = \Theta(\log n)$.

3. Graftraversering og avhengighetsgrafer

- Grafer: Eksplisitte og implisitte
- Traversering: DFS, BFS og venner
- Topologisk sortering
- Eksempel: Prims algoritme
- Avhengighetsgrafer: Dyp innsikt

Grafer: Eksplisitte og implisitte

En (urettet) graf er et nettverk som består av noder, koblet sammen av kanter. Kantene i en rettet graf har retning. I en vektet graf har hver kant en vekt. I et flytnettverk har hver kant en kapasitet.

En rettet asyklisk graf (DAG, Directed Acyclic Graph) er en rettet graf uten rettede sykler (man kan ikke følge kantenes retning i en sykel). Et tre er en rettet eller urettet graf uten sykler (man kan ikke følge kantene i en sykel uansett retning).

En sti er en serie med noder der hvert par med etterfølgende noder er bundet sammen av en kant. I en rettet sti går alle kantene i samme retning ("fremover"). I en sammenhengende graf er alle noder forbundet med en (urettet) sti.

Grafer kan være eksplisitte (representert ved nabomatriser, nabolister e.l.) eller implisitte (del av problemstrukturen, men ikke representert som datastruktur).

Traversering: DFS, BFS og venner

Når vi traverserer en graf har vi kun lov til å gå langs kantene (i riktig retning). Ved full traversering besøker vi alle nodene (potensielt flere ganger). Ved søk besøker vi bare de vi trenger for å finne noden vi leter etter.

Vi har en huskeliste over noder vi skal besøke. Til å begynne med inneholder denne bare startnoden. Så lenge denne huskelisten ikke er tom så krysser vi av en node og besøker denne. For hver node vi besøker oppdager vi kanskje noen nye noder (naboer). Vi legger disse til i huskelisten vår og fortsetter.

Hvis huskelisten er en kø (FIFO) får vi bredde-først-søk (BFS). Hvis huskelisten er en stakk (LIFO) får vi dybde-først-søk (DFS). DFS kan enkelt implementeres med rekursjon. Hvis huskelisten er en mer komplisert prioritetskø får vi straks interessante algoritmer som Prims og Dijkstras.

Topologisk sortering

Kantene i en rettet graf kan ses på som avhengigheter. Vi vil gjerne kunne ordne nodene i en rekkefølge slik at hver node kun er avhengig av de tidligere nodene. Nodene er da topologisk sortert. (Ofte mange mulige rekkefølger.) Dette er kun mulig hvis grafen er en DAG.

Ved å skrive ut nodene etter den rekkefølgen et dybde-først-søk forlater dem i (såkalt post-order traversal) får vi en topologisk sortering, siden hver skrives ut etter alle nodene den avhenger av.

En enkel rekursiv traversering kan altså gi oss en topologisk sortering. Hvis vi ikke husker hvilke noder vi allerede har besøkt kan vi risikere at noen noder blir med flere ganger; vi har da ikke en normal topologisk sortering, men nodene vil uansett skrives ut i en lovlig rekkefølge.

Eksempel: Prims algoritme

Spenntre: Et subsett av nodene i en graf som utgjør et tre og som knytter sammen alle noder i grafen. Minimalt spenntre: Spenntre som har minimal vekt-sum over kantene.

Prims algoritme finner et minimalt spenntre ved hjelp av enkel graf-traversering. Huskelisten er en prioritetskø der prioriteten til hver ubesøkt node er vekten på den korteste kanten som forbinder den med spenntreet.

Avhengighetsgrafer: Dyp innsikt

Et problem kan nesten alltid deles opp i delproblemer. For et gitt problem har man ofte mange delproblemer som igjen kan deles opp i delproblemer, og imellom disse delproblemene har man avhengigheter. Disse avhengighetene kan representeres med en rettet graf. Hvis problemet ikke har sykliske avhengigheter (som leder til uendelig rekursjon) kan det representeres med en DAG.

For veldig enkle problemer er avhengighetsgrafen rett og slett en sti: Hvert problem har kun ett delproblem. Den rekursive sum-funksjonen har for eksempel følgende avhengighetsgraf:

$$f(1) \to f(2) \to \cdots \to f(n-1) \to f(n)$$

For å beregne $f(n)$ må alle delproblemene til venstre beregnes. Dette kan gjøres enten iterativt (fra venstre mot høyre) eller rekursivt ("tilsynelatende" fra høyre mot venstre).

Avhengighetsgrafer: Dyp innsikt (2)

For mer kompliserte avhengighetsgrafer: Finn en topologisk sortering, og beregn delproblemene i denne rekkefølgen.

Husk: En topologisk sortering kan alltid finnes ved hjelp av et rekursivt dybde-først-søk.

(Begrepet “avhengighetsgraf” er ikke pensum.)

4. Splitt og hersk

- Når avhengighetsgrafen er et tre
- Typiske kjøretider
- Eksempel: Binærsøk og søketrær
- Eksempel: Quicksort
- Eksempel: Randomized select
- Eksempel: Mergesort

Når avhengighetsgrafen er et tre

For mange problemer er avhengighetsgrafen et tre (der alle kantene peker "oppover"). Det vil si at vi kan dele problemet opp i delproblemer som er uavhengige av hverandre. Slike problemer kan ofte løses effektivt med rekursjon.

Algoritmer som utnytter denne problemstrukturen kalles ofte splitt-og-hersk-algoritmer (divide and conquer algorithms).

Typiske kjøretider

Rekurrensligningene for splitt-og-hersk-algoritmer har et typisk mønster. Hvis vi antar at problemet hele tiden deles i to like deler vil vi få rekurrensligninger av typen

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

hvis kun den ene halvparten må behandles (binærsøk), eller

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

hvis begge halvparter må behandles. Funksonen $f(n)$ er avhengig av hvor mye arbeid som må utføres før/etter de rekursive kallene.

Eksempel: Binærsøk og søketrær

Problem: Søk etter element i sortert tabell.
Delproblemer: Søk i første og andre halvdel.
Avhengighetsgrafen er altså et binærtre.

Ved å undersøke det midterste elementet i det tabellsegmentet vi søker i kan vi ekskludere ett av de to delproblemene. Arbeidet for hvert delproblem er $\Theta(1)$. Vi får dermed:

$$T(1) = 1$$
$$T(n) = T(n/2) + \Theta(1)$$

Iterasjonsmetoden gir oss uttrykket $T(n) = T(n/2^i) + \Theta(i)$. Vi setter $n/2^i = 1$ og får $i = \log_2 n$. Med andre ord:

$$T(n) = \Theta(\log n)$$

Hvis vi gjør om avhengighetsgrafen til et eksplisitt tre så får vi et binært søketre. Ved å dele problemet i flere deler enn 2 (og å dele etter ulike kriterier) kan vi få andre søketrær, som B-trær. Søketrær har gjerne andre operasjoner enn søk knyttet til seg, for å holde balansen.

Eksempel: Quicksort

Problem: Sorter elementene i en tabell.
Delproblemer: Sorter de små og store elementene hver for seg. Avhengighetsgrafen er altså et binærtre.

Her må vi løse begge delproblemene før vi kan løse hovedproblemet. Å dele elementene i små (venstre) og store (høyre) koster oss $\Theta(n)$ (partition). Under noen forenklende antagelser (f.eks. best-case-datasett) får vi følgende rekurrens:

$$T(1) = \Theta(1)$$
$$T(n) = 2T(n/2) + \Theta(n)$$

Denne rekurrensen kan beregnes enten med iterasjonsmetoden (litt prakk, men det går) eller Masterteoremet. Resultatet er:

$$T(n) = \Theta(n \log n)$$

(Average-case gir kjøretid $O(n \log n)$, men er vanskelig å beregne.)

Eksempel: Randomized select

Problem: Finn det "k-te største" elementet i en usortert tabell. (Ved å sette $k = n/2$ finner vi medianen.) Igjen kan vi lage oss flere delproblemer ved å dele opp tabellen. Vi søker etter elementet enten blant de "små" eller de "store" elementene, avhengig av hvor mange små og store elementer vi har. For å dele inn i små og store elementer bruker vi igjen partition.

Vi har altså to delproblemer, hvorav vi kun trenger å løse det ene. For å konstruere disse delproblemene må vi utfører $\Theta(n)$ operasjoner (partition). Vi får da følgende rekurrens for best-case:

$$T(1) = 1$$
$$T(n) = T(n/2) + \Theta(n)$$

Iterasjonsmetoden gir oss uttrykket $T(n/2^n) + \sum_{j=1}^{i} \Theta(n/2^{j-1})$. For $i = \log_2 n$ får vi $T(n) = \Theta(n)$. Dette er også average-case. Worst-case er $\Theta(n^2)$. I pensum finnes også en select-algoritme med worst-case kjøretid $\Theta(n)$.

Eksempel: Mergesort

Problem: Sorter elementene i en tabell.
Delproblem: Sorter venstre og høyre halvdel.
Avhengighetsgrafen vår er igjen et binærtre der vi må behandle begge halvdeler.

For å kombinere løsningene til delproblemene må vi kjøre en flette-operasjon (merge) som har kjøretid $\Theta(n)$. Siden vi alltid deler på midten vil worst-case for Mergesort bli akkurat det samme som best-case for Quicksort.

5. Dynamisk programmering . . .

- Når avhengighetsgrafen er en DAG
- Eksempel: Fibonacci-tallene
- Snu problemet på hodet
- Typiske kjøretider
- Eksempel: korteste vei i en DAG
- Eksempel: Dijkstras algoritme
- Eksempel: LCS
- Eksempel: Floyd-Warshall

. . . og grådighet

- Når vi har flaks med avhengighetsgrafen
- Eksempel: Huffman-koding
- Eksempel: Kruskals algoritme

Når avhengighetsgrafen er en DAG

Det er ikke alle problem som er snille nok til at splitt-og-hersk-taktikker fungerer. Enkelte ganger vil delproblemene avhenge av samme delproblem (på et lavere nivå). Vi sier da at vi har overlappende delproblemer, og avhengighetsgrafen blir en generell DAG.

Selv om antall delproblemer er polynomisk vil et naivt dybde-først-angrep sannsynligvis beregne delproblemene flere ganger – antagelig et eksponensielt antall.

Eksempel: Fibonacci-tallene

Et enkelt eksempel på dette er Fibonacci-tallene:

$$f(0) = f(1) = 1$$
$$f(n) = f(n-1) + f(n-2)$$

En ren rekursiv beregning vil gi her en eksponensiell kjøretid:

```
int fib(int n) {
  if (n==0 || n==1) return 1;
  return fib(n-1) + fib(n-2);
}
```

Eksempel: Fibonacci-tallene (2)

Ved å lagre alle delproblem som allerede er beregnet kan vi sørge for at hvert delproblem kun blir beregnet én gang. Det er det vi gjør i dynamisk programmering. En direkte rekursiv implementasjon av dynamisk programmering kalles memoisering (uten r!).

```
int d[1000]; // Vilkårlig størrelse
d[0] = d[1] = 1;
int fib(int n) {
  if (d[n]==0) // Ikke beregnet
    d[n] = fib(n-1) + fib(n-2);
  return d[n];
}
```

Denne implementasjonen vil gi lineær kjøretid.

Snu problemet på hodet

Hvis du har tenkt ut en rekursiv løsning på problemet ditt, og har funnet ut en måte å lagre delproblem på, så trenger du ikke nødvendigvis beregne dem rekursivt. En rekursiv dybde-først-beregning vil gi deg en korrekt topologisk ordning av delproblemene, men husk at en hvilken som helst topologisk ordning vil gi korrekt svar. Ofte kan du beregne delproblemene iterativt.

For Fibonacci-tallene kan vi beregne fra $f(0)$ og oppover:

```
int d[1000]; // Vilkårlig størrelse
d[0] = d[1] = 1;
int fib(int n) {
  for (int i=2; i<=n; ++i)
    d[i] = d[i-1] + d[i-2];
  return d[n];
}
```

En smartere løsning kunne klare seg med å lagre de to forrige verdiene.

Typiske kjøretider

Det er vanskeligere å si noe generelt om kjøretider for algoritmer basert på dynamisk programmering. For å beregne kjøretiden for en gitt algoritme må du finnet ut hvor mange delproblemer du har, hvor mange kanter du har i avhengighetsgrafen, og hvor mye hver kant/node koster å beregne.

For Fibonacci-tallene har vi $\Theta(n)$ delproblem, og hvert delproblem har bare to avhengigheter. Hvert delproblem og hver avhengighet tar $\Theta(1)$ tid å beregne, så den totale kjøretiden blir $\Theta(n)$.

Eksempel: Korteste vei i en DAG

Finn den korteste veien mellom to noder i en DAG. Anta at startnoden ikke har noen inn-kanter og at sluttnoden ikke har noen ut-kanter.

Dette er et klassisk eksempel på dynamisk programmering, og her er vi så heldige at selve problemet gir oss avhengighetsgrafen eksplisitt. Delproblemene er nodene, og for å finne avstanden fra starten til en gitt node må vi ha avstandene til alle foreldrenodene (de med kanter inn til noden).

Løsning: Gjør en topologisk sortering og beregn avstandene fra start til slutt.

Avstanden $d[i]$ til en node i er minimum over $d[j] + w(j, i)$ for alle noder $j < i$ som har en kant til i.

Eksempel: Dijkstras algoritme

Dijkstras algoritme går ett skritt videre enn DAG-shortest-path: Den tillater sykler i grafen (forutsatt at vi ikke har negative kantvekter).

Også her kan vi si at hver node (hvert delproblem) er avhengig av alle foreldrenodene (de med kanter inn til noden). Problemet er at vi ikke kan løse et problem med en syklisk avhengighetsgraf direkte.

Dijkstras algoritme løser dette på en genial måte: En node kan jo bare være avhengig av de nodene med lavere avstandsverdi (siden vi ikke har negative kanter). Hvis vi beregner dem i rekkefølge etter denne avstanden vil vi få riktig svar.

Algoritmen baserer seg på en viktig egenskap: Hvis vi har beregnet de k nærmeste nodene, og hele tiden har kjørt relax på alle nyoppdagede noder, så vil noden med lavest avstandsestimat ha korrekt estimat.

Eksempel: Dijkstras algoritme (2)

Med andre ord: Vi kjører en traversering der huskelisten er en prioritetskø, og prioriteten er avstandsmålet. Vi må sørge for å holde avstandsmålet oppdatert med relax (og dermed oppdatere prioritetskøen).

Hvis prioritetskøen er en binær heap (og grafen er sammenhengende) blir kjøretiden $O(E \log V)$.

Eksempel: LCS

For to sekvenser A og B, finn den lengste subsekvensen som forekommer i begge sekvensene. En subsekvens er et subsett av elementene elementer i samme rekkefølge som i originalen, ikke trenger ligge ved siden av hverandre. (F.eks. er $(2, 4)$ en subsekvens av $(1, 2, 3, 4, 5)$.)

Som vanlig gjelder det å finne en fornuftig avhengighetsgraf. Hva blir delproblemene? Hvordan avhenger de av hverandre? Det er dette som er det vanskelige spørsmålet når man bruker dynamisk programmering.

En intuisjon kan være å begynne å regne fra venstre, men vi har jo to sekvenser . . . Vi definerer et delproblem til å bestå av to prefix $(A[1{:}i], B[1{:}j])$, og kan da vise at løsningen kun avhenger av tre andre delproblem: $(A[1{:}i{-}1], B[1{:}j])$, $(A[1{:}i], B[1{:}j{-}1])$, og $(A[1{:}i{-}1], B[1{:}j{-}1])$.

(Se læreboka for detaljer.)

Eksempel: LCS (2)

Siden alle problemer har 3 delproblemer der problemstørrelsen kun er redusert med en konstant, så vil en naiv rekursiv løsning være eksponensiell. Men det er helt tydelig at det er stort overlapp mellom delproblemene. Hvis A og B har lengde m og n respektivt, så er det totale antall delproblem polynomisk, nemlig $n \cdot m$.

Vi kan altså sette opp avhengighetsgrafen som noder i et rutenett med koordinater (i, j) og kanter fra $(i-1, j)$, $(i, j-1)$, og $(i-1, j-1)$ til (i, j). Ved å plassere del-løsningene i en todimensjonal tabell kan vi beregne dem enten ved å iterere fra bunnen, $(0, 0)$, eller ved rekursjon (memoisering).

Eksempel: Floyd-Warshall

Beregn korteste avstand mellom alle par med noder i en graf. Igjen: Hvordan finner vi delproblemer, og hvordan henger de sammen?

Floyd-Warshall har en smart løsning: Nummerer nodene fra 1 til n og la delproblem k være de korteste veiene som bare har lov til å gå via nodene $1 \ldots k$.

Det smarte er som følger: Hvis vi har de korteste avstandene der kun nodene $1 \ldots k-1$ er mellomledd så er det en smal sak å beregne de korteste avstandene der også k er tillatt.

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & k \geq 1 \end{cases}$$

Ved å bruke en tredimensjonal tabell (`d[i,j,k]`) kan dette beregnes med memoisering. Mer effektivt er det å bruke en todimensjonal tabell (`d[i,j]`) og gjøre unna én og én verdi for k iterativt.

Når vi har flaks med avhengighetsgrafen

Av og til kan du ha flaks med avhengighetsgrafen. I disse tilfellene kan du finne delproblemer og avhengigheter som for dynamisk programmering, men istedenfor å løse alle delproblemer så kan du velge kun ett: Det som har best løsning.

Det ligger som regel en DP-algoritme under enhver grådig algoritme. Hensikten med grådighet er å slippe å beregne alle delproblemene ved å velge det riktige delproblemet direkte.

Eksempel: Huffman-koding

Gitt et sett med tegnfrekvenser for et dokument, finn en binær koding med variabel lengde slik at den totale lengden på det kodede dokumentet blir minimal. Disse kodene kan representeres som stier i et binærtre (0=venstre, 1=høyre) der tegnene er løvnoder.

Huffman-algoritmen finner kodene på grådig vis: Konstruer en skog (sett av trær) der hver bokstav er et tre. Kombiner de to trærne med lavest total frekvens helt til du bare står igjen med ett tre.

Vi kan se på hvert delproblem som en kombinasjon av to trær. Hvis vi ser på frekvensen til det nye treet som kostnaden ved et mulig valg, velger Huffman-algoritmen altså det delproblemet som har lavest kostnad.

(Korrekthetsbeviset står i boka.)

Eksempel: Kruskals algoritme

Kruskals algoritme løser min-spenntre-problemet på en måte som ligner Prims. Prims velger hele tiden den minste kanten som ligger inntil de nodene som allerede er med i spenntreet, og som ikke skaper en løkke. Kruskals algoritme velger rett og slett den minste kanten (hvor som helst i grafen) som ikke skaper en løkke i spenntreet.

Begge algoritmene kan ses på som grådige. For hvert trinn velger de den lokalt beste løsningen.

(Det generelle korrekthetsbeviset for MST-algoritmer står i boka og er verdt å få med seg.)

6. Iterative algoritmer

- Når avhengighetsgrafen er litt diffus
- Eksempel: Bellman-Ford
- Eksempel: Ford-Fulkerson/Edmonds-Karp

Når avhengighetsgrafen er litt diffus

Det er ikke alltid vi klarer å få en avhengighetsgraf som egner seg får splitt-og-hersk-algoritmer eller dynamisk programmering. Men det hender at vi likevel kan takle problemet bit for bit, og vise at vi til slutt vil ha løst hele problemet.

Eksempel: Bellman-Ford

Finn korteste vei fra én node til alle andre. Fungerer også med negative kantvekter; hvis det finnes negative løkker vil algoritmen si ifra om at ingen løsning finnes.

Idéen er enkel: Kjøre relax langs alle kanter helt til du er sikker på at alle noder må ha riktig avstandsestimat.

Hvor mange ganger må du gjøre det? Etter én gang vil alle naboene til startnoden ha riktig estimat. Etter to ganger vil naboene deres ha riktig estimat, etc. Hvis du har maks uflaks kan grafen være en sti, og du må da kjøre relax $V-1$ ganger for å nå frem til sluttnoden.

Til slutt sjekker algoritmen om det finnes noen noder som kan få forbedret avstanden sin med enda et kall til relax – hvis det finnes, så er det en negativ sykel i grafen.

Kjøretiden blir $O(VE)$.

Eksempel:
Ford-Fulkerson/Edmonds-Karp

Finn maksimal flyt i et flyt-nettverk. Her gjelder det å fylle på mer og mer flyt til det ikke går an å fylle på mer.

Ford-Fulkerson-metoden (generell metode mer enn en algoritme) sier følgende: Hvis vi gang på gang leter opp en sti fra kilde til sluk som har ledig kapasitet, og skyver så mye flyt igjennom, vil vi til slutt få maksimal flyt. (Ganske logisk, egentlig.)

Edmonds-Karp-algoritmen bruker Ford-Fulkerson-metoden, og velger alltid den korteste veien med ledig kapaitet (bruker BFS). Dette gir bedre kjøretid enn vilkårlig valg av sti.

Hovedpoenget er at "delproblemene" våre er "biter" av den maksimale flyten (ledig kapasitet). Vi leter opp disse "bitene" og løser dem ved å kjøre på mer flyt.

Avsluttende ninjatriks 1 (av 2)

Binærsøk kan brukes til mer enn å søke i tabeller ...

Anta at du har en algoritme $A(x)$ som kan sjekke om parameteren x er for stor eller for liten. Anta også at x kan ta verdier i en endelig ordnet mengde, f.eks. $1 \ldots n$. Hvis algoritmen $A(x)$ har kjøretid $T(n)$ så vil du med binærsøk kunne finne riktig verdi for x med kjøretid $T(n) \log m$.

Eksempel: Anta at du har en algoritme $A(G, x)$ som for en gitt graf G kan avgjøre om alle noder kan nå hverandre med maks x steg. Anta at denne algoritmen har kjøretid $\Theta(V^2)$. Da kan du lett vinne den laveste lovlige verdien for x i kjøretid $\Theta(V^2 \log V)$.

En faktor $\log n$ er som regel ganske liten.

Men dette problemet kan løses mer effektivt; og det leder oss videre til ...

Avsluttende ninjatriks 2 (av 2)

Jo mer du vet om problem-instansen (datasettet) ditt, jo bedre kan du løse problemet.

Eksempel: Generell sortering er $\Omega(n \log n)$, men hvis du vet at elementene er heltall i området $1 \ldots k$ (der k ikke er så alt for stor) kan du bruke tellesortering (eller radiks-sortering), og sortere i $\Theta(nk)$ tid.

Generelle algoritmer (f.eks. lineær programmering) trengs, og de kan brukes til å løse det meste, men er som regel dårligere enn skreddersydde algoritmer.

Derfor er det viktig at dere lærer å lage deres egne algoritmer!

Stå på, og lykke til på eksamen,

Magnus.