4: Rekursjon

Hva er rekursjon

En rekursiv funksjon er en funksjon som kaller seg selv. Dette er tillatt i de alle fleste programmeringsspråk, og er viktig programmeringsteknikk. Rekursjon brukes når et problem kan løses gjennom å løse mindre utgaver av samme problem.

Eksempel: Fakultet

Den matematiske operasjonen Fakultet, n!, kan defineres på følgende måte:

0! = 1

n! = n * ((n-1)!)

I denne definisjonen er n! lik (n-1)! ganger n. Så for å finne n! må du finne (n-1)!. Og for å finne (n-1)! må du finne (n-2)! og så videre helt ned til 0!. 0! kalles for basetilfellet. Alle rekursive algoritmer trenger ett eller flere basetilfeller. Ellers får du den rekursive versjonen av ei evig løkke. En slik evig løkke vil gi en StackOverflowError siden kallstabelen til slutt blir full.

Eksempel: Euclid's Algoritme for største felles divisor

Problem: Gitt to heltall a og b, finn det høyeste heltallet c hvor både a og b er delelig på c.

Algoritme – gcd(a, b)

- Hvis b=0, returner a
- Ellers returner gcd(b, a%b)

Rekursive vs. iterative algoritmer

Alle rekursive algoritmer kan i teorien oversettes til iterative algoritmer. Hvor enkelt dette er avhenger av algoritmen. Begge de to foregående algoritmene kan skrives med det som kalles halerekursjon. Halerekursjon vil si at funksjonen har bare ett kall til seg selv, og det kallet er det siste som skjer. En halerekursiv algoritme kan enkelt oversettes til en iterativ algoritme gjennom å bruke en eksplisitt variabel for å lagre mellomresultatene. En del mer avanserte kompilatorer for kompilerte språk vil automatisk oversette en halerekursiv algoritme til en iterativ versjon.

Om man foretrekker rekursive eller iterative algoritmer avhenger av programmeringsspråk og kodestil. Generelt sett er iterative algoritmer ofte raskere selv med samme kjøretid i O-notasjon siden det å manipulere kallstabelen krever mer arbeid enn å manipulere de ekstra variablene en iterativ versjon krever. Det finnes imidlertid en programmeringsstil kalt funksjonell programmering hvor man ønsker å unngå det som kalles side-effekter. I funksjonell programmering skal en funksjon ta noen verdier inn og returnere noen andre verdier. Og alt i programmet er funksjonskall. Funksjoner skal ikke gjøre noen andre endringer slik som å tilordne verdier til variabler som kan brukes etterpå. Fordelen med denne programmeringsteknikken er at funksjonelle programmer er lette å parallellisere. I funksjonell programmering foretrekker man rekursjon framfor iterasjon siden iterative algoritmer som regel krever side-effekter som tilordning til en løkke-variabel.

De tre lovene for rekursive algoritmer

- **Alle rekursive algoritmer må ha et basetilfelle hvor rekursjonen stopper.** Ellers får du den rekursive versjonen av ei evig løkke.

- **Hvert rekursive kall må endre tilstanden i retning mot basetilfellet.** Ellers får du også ei rekursiv evig løkke.
- **Den rekursive algoritmen må kalle seg selv.** Dette er definisjonen på rekursjon

Enkelte inkluderer også en fjerde lov:

- **Ikke gjenta deg selv.** Ikke gjør samme jobb flere ganger i ulike deler av kjøringen av algoritmen. Konsekvensen av å bryte denne loven er at kjøretida blir veldig mye dårligere enn den hadde trengt å være. Et eksempel på dette kommer i temaet «dynamisk programmering».

Fraktaler

Et eksempel på rekursive algoritmer er tegning av fraktaler. En fraktal er et mønster som gjentar seg selv på ulike zoom-nivåer. Hele figuren består av mindre kopier av figuren, som igjen består av mindre kopier av figuren, osv. For matematiske fraktaler kan man zoome så langt med man vil. For å tegne fraktaler trenger man et basetilfelle hvor man stopper.

Eksempler:

- 2d-tre. Kjør gjerne tester med ulikt antall nivå for å se hvordan figuren endrer seg.
- Mønster av firkanter. Kjør gjerne tester med ulikt antall nivå for å se hvordan figuren endrer seg.
- Juletre

Algoritmetype: Splitt og Hersk

Dette er en type algoritme som man ofte skriver som rekursive algoritmer. Prinsippet er at man deler problemet inn i mindre uavhengige delproblemer, løser hvert delproblem for seg, og kombinerer løsningene på delproblemene til en løsning på hele problemet.

Eksempel: Hvis du ønsker å finne høyeste verdi ei liste kan du splitte lista i to. Høyeste verdi i hele lista er enten høyeste verdi i første halvpart eller høyeste verdi i andre halvpart. Jeg vil ikke vise koden for denne siden det viser seg at du ikke vinner noe på dette sammenliknet med å gå gjennom lista.

Eksempel: Sortering. To av sorteringsalgoritmene som blir presentert i temaet «sortering» er splitt-og-hersk algoritmer som sorterer hver halvpart av lista for seg og deretter kombinerer de to listene til en sortert liste. Dette vinner du på, men hvordan og hvorfor blir tatt i temaet «sortering».

En variant av splitt-og-hersk er det enkelte kaller reduser-og-hersk, engelsk decrease-and-conquer. Dette er problemer hvor man bare trenger å undersøke en av delproblemene for å finne løsningen.

Eksempel: Binærsøk

For problemet «Finn et oppgitt element i ei liste» har algoritmen sekvensielt søk blitt presentert. Vet du ikke noe mer om lista så er det det beste du får til. Hvis du vet at lista er sortert på det du ønsker å leite etter, kan du imidlertid gjøre bedre gjennom å bruke reduser-og-hersk teknikken.

Algoritme Binærsøk(liste) -> indeks:

- Sjekk det midterste elementet i lista.
- Er det det du leiter etter er du ferdig

- Er midterste element mindre enn det du leiter etter, må det du leiter etter ligge i øverste halvpart. Kall binærsøk med øverste halvpart av lista.
- Ellers må det du leiter etter ligge i nederste halvpart. Kall binærsøk med nederste halvpart av lista.

Rekursiv og iterativ versjon ligger begge ute.

En variant av binærsøk som brukes hvis å sjekke likhet på elementene i lista er svært dyrt er interpolasjonssøk. Den fungerer som binærsøk bortsett fra at du prover å regne deg fram til omtrent hvor elementet ligger i lista og splitter der i stedet. Dette forutsetter at elementene er omtrent jevnt spredt slik at det er relativt sannsynlig å treffe omtrent riktig. For eksempel hvis du leiter etter en person med etternavn «Ås» i ei sortert liste på etternavn, vil vedkommende ligge nær slutten av lista.

Analyse av kjøretida til rekursive algoritmer: Rekurrenslikninger

Siden en rekursiv algoritme kaller seg selv med mindre versjoner av samme problem, må man sette opp en likning og så løse den for å finne kjøretida.

I disse formlene står T(n) står «Tida det tar å løse et problem som er n stort».

En rekurrenslikning vil typisk se slik ut:

T(n) = <antall rekursive kall>*T(<størrelsen til hvert delproblem>) + <kjøretida til algoritmen utenom de rekursive kallene>

T(1) eller T(0) = <kjøretida til basetilfellet>

Eksempel: Rekursiv fakultet.

- Den gjør ett rekursivt kall
- Delproblemet er én mindre
- Kjøretida til algoritmen uten rekursjonen er O(1). Dette gjelder også basetilfellet

Så: T(n) = T(n-1) + O(1)

Eksempel: Binærsøk

- Den gjør ett rekursivt kall
- Delproblemet er halvparten så stort
- Kjøretida til algoritmen uten rekursjonen er O(1). Dette gjelder også basetilfellet

Så: T(n) = T(n/2) + O(1)

Løsningsmåte: Gjentatt substitusjon

Dette er den mest generelle løsningsmetoden. Den er best illustrert gjennom et eksempel:

Rekursiv fakultet: T(n) = T(n-1) + O(1)

I denne metoden fjerner man ofte O-en i det siste leddet siden man ønsker å telle hvor mange ganger leddet dukker opp. Hvis det viser seg å være en funksjon an problemstørrelsen så teller det i den endelige O-notasjonen. Så rekurrenslikningen skrives:

T(n) = T(n-1) + 1

Første steg: Skriv formelen for T(n-1) gjennom å sette inn n-1 i stedet for n: T(n-1) = T((n-1)-1) + 1= T(n-2) + 1

Andre steg: Sett likningen fra første steg inn for T(n-1) i opprinnelig likning: T(n) = T(n-2) + 1 + 1 = T(n-2) + 2

Tredje steg: Skriv formelen for T(n-2): T((n-2)-1) + 1 = T(n-3) + 1

Fjerde steg: Sett inn likningen fra tredje steg inn i likningen fra andre steg: T(n) = T(n-3) + 1 + 2 = T(n-3) + 3

Femte steg: Se etter et mønster. Ser du det ikke, fortsett noen steg til for å se om det dukker opp. Et mønster i dette tilfellet vil være T(n) = T(n-k) + k

Sjette steg: Se hvor mønsteret leder. For å få basetilfelle kan du sette k=n. Da får du T(n) = T(n-n) + n =T(0) + n. Basetilfellet har kjøretid O(1). Så T(n) = O(1) + n, som er O(n)

Hvordan vise at dette stemmer: Induksjonsbevis

Hvis du har funnet en kjøretid på denne måten, men ikke er sikker på at den stemmer kan du bruke et induksjonsbevis. Teknikken er som følger:

- Bevis påstanden for basetilfellet
- Bevis at hvis påstanden gjelder for n-1 så gjelder den også for n

Da har du bevist det for alle verdier over basetilfellet siden du kan gjenta det andre punktet helt fra basetilfellet og hvor langt opp du ønsker.

Eksempel: Kjøretida til rekursiv fakultet.

Hypotesen er at T(n) = n. For basetilfellet er n=1, så kjøretid O(1) stemmer. For det andre tilfeller har du likningen T(n) = T(n-1) + 1. Gitt at det gjelder for n-1 så kan man sette inn n-1 i formelen og få T(n) = n-1+1 = n.

Eksempel: Summen 1 + 2 + 3 + 4 + … + n = n(n+1)/2

For n=1 gir formelen 1(1+1)/2 = 1*2/2 = 2/2 = 1, som stemmer.

Gitt at formelen gjelder for n-1, gjelder den også for n. Summen 1 + 2 + 3 + … + n kan skrives som S(n) = n + S(n-1). Da kan man sette inn formelen for S(n-1): S(n) = n + (n-1)*n/2. For å regne ut dette bør alt ha felles nevner, så erstatt n med 2n/2. S(n) = (2n + (n-1)n)/2. Gang ut det indre produktet for å få $S(n) = (2n + n^2 – n)/2 = (n + n^2)/2 = \underline{n(n+1)/2}$.

Løsningsmetode: Master Theorem

Man kan finne kjøretida til splitt-og-hersk (og reduser-og-hersk) algoritmer med rekurrenslikning på formen $T(n) = a*T(n/b) + O(n^k)$ gjennom å sjekke de tre verdiene a, b og k på følgende vis:

- Hvis $a > b^k$ er kjøretida $O(n^{log\_b(a)})$
- Hvis $a = b^k$ er kjøretida $O(n^k*log(n))$
- Hvis $a < b^k$ er kjøretida $O(n^k)$

En måte å tenke på det er at i det siste tilfellet er det det opprinnelige kallet av funksjonen som dominerer, de rekursive kallene er så små i sammenlikning at de forsvinner. I det første tilfellet er det de rekursive kallene som dominerer. I det andre tilfellet er de to jevnstore.

Eksempel: Binærsøk. For denne algoritmen er a=1, b=2 og k=0 (siden $n^0 = 1$). Så b^k blir 2^0 som blir 1. Siden a=1 så har man tilfelle 2, som gir $O(n^0log(n)) = O(log(n))$.

Algoritmetype: Dynamisk Programmering

Splitt og hersk kan brukes hvis man kan dele opp et problem i uavhengige delproblem. Hvis man ikke kan det, men likevel kan kombinere løsninger på delproblemer til løsninger på hele problemet, så bruker man dynamisk programmering. Poenget er å lagre mellomresultatene slik at man ikke trenger å beregne dem mer enn én gang.

Eksempel: Fibonacci-tallene

Fibonacci-tallene er en tallsekvens som kan defineres slik, hvor F(n) er det n-te Fibonacci-tallet:

F(0) = 0

F(1) = 1

F(n) = F(n-1) + F(n-2)

Dette er en sekvens som enkelt lar seg oversette til en rekursiv algoritme. Læreboka i DAT110 grunnleggende programmering har en versjon av denne algoritmen.

Denne algoritmen har imidlertid et stort problem: Den bryter regelen om at du ikke skal gjenta deg selv.

Funksjonskall som skjer

Første nivå F(n) = F(n-1) + F(n-2). Setter inn kallene F(n-1) og F(n-2) gjør for å få:

Andre nivå: F(n) = F(n-2) + F(n-3) + F(n-3) + F(n-4). Her kalles F(n-3) to ganger! Jeg tar ett nivå til av dette:

Tredje nivå: F(n) = F(n-3) + F(n-4) + F(n-4) + F(n-5) + F(n-4) + F(n-5) + F(n-5) + F(n-6). Her kalles F(n-4) og F(n-5) begge tre ganger!

Kjøretid

Rekurrenslikning T(n) = T(n-1) + T(n-2) + 1, T(0) = T(1) = 1

Denne blir fort grisete, men vi kan si at

T(n) < 2*T(n-1) + 1

T(n) > 2*T(n-2) +1

Disse er lettere å løse:

T(n) = 2*T(n-1) + 1

Setter opp for n-1: T(n-1) = 2*T(n-2) + 1

Erstatter: T(n) = 2*(2*T(n-2) + 1) +1 = 4*T(n-2) + 3

Setter opp for n-2: T(n-2) = 2*T(n-3) + 1

Erstatter: T(n) = 4*(2*T(n-3) + 1) + 3 = 8*T(n-3) + 7

Ser etter mønster: $T(n) = 2^k*T(n-k) + 2^k-1$

Setter inn k=n-1 for å få T(1):

$T(n) = 2^{n-1}*T(1) + 2^{n-1} – 1$

Kjøretid $O(2^n)$

$T(n) = 2*T(n-2) + 1$

Setter opp for n-2: $T(n-2) = 2*T(n-4) + 1$

Erstatter: $T(n) = 2*(2*T(n-4) + 1) + 1 = 4*T(n-4) + 3$

Setter opp for T(n-4): $T(n-4) = 2*T(n-6) + 1$

Erstatter: $T(n) = 4*(2*T(n-6) + 1) + 3 = 8*T(n-6) + 7$

Ser etter mønster: $T(n) = 2^k*T(n-2k) + 2^k-1$

Setter inn k=n/2 for å få T(0) som er lik T(1)

$T(n) = 2^{n/2}*T(0) + 2^{n/2}-1$

$2^{n/2} = (2^{1/2})^n = sqrt(2)^n$

Kjøretid $\Omega(sqrt(2)^n)$

(Ω i stedet for O siden dette er den nedre grensa. Denne har ingen Θ kjøretid siden O og Ω er ulike)

Konklusjon kjøretid

Denne har eksponensiell kjøretid!

Bedre algoritme: Rekursiv med dynamisk programmering

- Lag ei liste med n verdier
- Når du skal regne ut et Fibonacci-tall, sjekk om det allerede er i lista og returner det som ligger der hvis det er noe
- Regn det ut som før hvis det ikke ligger der
- Legg inn resultatet i lista

I video-en tegner jeg en figur som er den beste måten å vise kjøretida til denne, som er O(n). Dette er mye bedre enn den forrige! Den bruker også O(n) plass siden den lager ei liste som er n lang.

Beste algoritme: Start fra bunnen med iterativ dynamisk programmering

- Lag to variabler n2 og n1 som lagrer de to foregående verdiene og initialiser dem med verdiene 0 og 1
- For-løkke fra 2 til n:
    - Regn ut Fibonacci-tallet F(i) = n2 + n1
    - Sett n2 = n1
    - Sett n1 = F(i)
- Returner n1

Denne har dere allerede lært hvordan å analysere. Den allokerer bare fire variabler uansett hvor stort Fibonacci-tall du ønsker. Den bruker O(n) tid og O(1) plass.

Algoritmetype: Grådige algoritmer

I et tilfelle hvor man må ta valg, men det er vanskelig å beregne hva som vil være et optimalt valg for problemet som helhet, så velger man å gjøre et «lokalt optimalt» valg og håper at det vil virke.

Eksempel: Du skal gi vekslepenger og skal gi et minimalt antall mynter av ulike verdier. Dette eksemplet kan bruke en grådig algoritme hvis du har vanlige norske mynter på 1, 5, 10 og 20 kroner, men ikke hvis du har et sært myntsystem.

Eksempel: Gi 63 kroner i mynter, bruk så få mynter som mulig. Den grådige algoritmen ser slik ut:

Gi_mynter(beløp)

- Sett restbeløp lik beløp
- Fortsett til restbeløp er 0
    - Gi den største mynten som ikke er større enn restbeløpet
    - Senk restbeløpet med verdien til mynten du nettopp gav

Analyse

- For 63 kroner og vanlige norske mynter velger den først tre 20 kroners mynter. Så velger den tre 1 kroners mynter. Dette er optimalt.
- La oss isteden si at vi har mynter på 1, 3, 7, 15, 21 og 25 kroner. Greedy-algoritmen velger da følgende mynter: 25 + 25 + 7 + 3 + 1. Det optimale valget er 21 + 21 + 21.

Kursorisk pensum: P vs NP problemet

Et viktig skille i algoritmeteorien er skillet mellom algoritmer som kjører i polynomisk tid og algoritmer som ikke gjør det. Polynomisk tid vil si at kjøretida i O-notasjon kan uttrykkes gjennom et polynom. O(n) og $O(n^2)$ er begge polynomiske. O(log(n)) er også O(n) og derfor polynomisk. Naiv fibonacci kjører imidlertid i $O(2^n)$ tid. Dette er ikke polynomisk. Kjøretider som $O(2^n)$ og O(n!) er typisk for ikke-polynomiske algoritmer.

Et kjent men uløst matematisk spørsmål

Ett av de sju «millenniumsprisproblemene» innen matematikken er relevant for DAT200. En løsning på et an disse gir en million dollar i belønning.

Terminologi

- Man sier at et problem er i P hvis det finnes en algoritme som løser problemet i polynomisk tid.
- Man sier at et problem er i NP hvis man har en algoritme for å sjekke at en løsning er korrekt i polynomisk tid. Algoritmen man har for å løse problemet kan godt være ikke-polynomisk.

Spørsmålet er: Finnes det egentlig problemer som er i NP men ikke i P? Er P og NP ulike?

I leiting etter svaret har man funnet en mengde problemer kalt «NP-komplette problem». Dette er en delmengde av NP som er slik at et hvert problem i NP kan konverteres til et av de NP-komplette problemene. Så de NP-komplette problemene er så vanskelige som problemer i NP blir. Klarer man å finne en algoritme som løser ett av de NP-komplette problemene i polynomisk tid, har man vist at P = NP.

Det har også vist seg at nåværende teknikker for å bevise kjøretider ikke er kraftige nok til å bevise eller motbevise at P er ulik NP. Det finnes imidlertid mange problemer som er i NP hvor den beste kjente algoritmen for å løse problemet kjører i eksponentiell tid.

Et eksempel på konvertering av problem: Problemet «Finn personen med navn lengst bak i alfabetet i ei liste» kan konverteres til problemet «Finn høyeste tall i ei liste» siden navn kan konverteres til tall slik at desto lengre bak i alfabetet en person er desto høyere blir tallet.

Eksempel på et NP-komplett problem: Delmengdesum problemet:

Gitt ei liste med tall, kan du velge noen av tallene fra lista (flere enn 0) slik at summen av tallene du valgte blir 0? For eksempel gitt at du har lista [-7, -3, -2, 9000, 5, 8]. Da kan du velge tallene {-3, -2, 5} hvor summen av tallene er 0. Tallene {-3, -2, 5} er en delmengde av tallene i lista siden de tre tallene er med i lista.

For å finne ut om det er mulig å velge tall fra lista slik at summen blir 0 må man sjekke mange (ikke alle) mulighetene, og det er 2^n-1 muligheter. For hvert tall i lista så er det enten med eller ikke med, så det er 2 muligheter for hvert tall, og 2^n muligheter totalt. -1 fordi du ikke tillater at ingen av tallene velges.

Men å sjekke at en løsning er riktig går raskt. Man må sjekke at alle tallene er med i lista, og sjekke at summen er riktig. Kjøretida for å sjekke at et tall er med i ei liste er O(n). Ved å bruke en struktur kalt et HashSet, som blir presentert seinere i DAT200, er det O(1) tid å sjekke at et tall er med i en mengde.