Tolv forelæsninger i algoritmik

ved
Keld Helsgaun

Forelæsning 1: Algoritmebegrebet

Hvad er en ”algoritme”? Denne første forelæsning besvarer spørgsmålet og giver en introduktion til algoritmebegrebet. Der gives en række definitioner af begrebet, spændende fra en simpel, intuitiv definition til en stringent matematisk definition.

Ofte er der mere end én algoritme, der kan løse et givet problem. Vi diskuterer, hvad der forstås ved en ”god” algoritme. Egenskaber som *korrekthed*, *effektivitet* og *simpelhed* er ofte værd af efterstræbe, men de kan ofte være i konflikt med hinanden.

Hvorfor bekymre sig om effektivitet, når man har adgang til dagens hurtige computere? Vi diskuterer det stadige behov for effektive algoritmer, trods den teknologiske udvikling.

Vi ser nærmere på en klassisk algoritme, Euclids algoritme. Algoritmen kan benyttes til bestemmelse af største fælles divisor for to heltal, og er bl.a. er nyttig ved forkortelse af brøker. Den blev udviklet cirka 300 år f. Kr. og er en af de første ikke-trivielle algoritmer i historien. Vi ser forskellige udgaver af algoritmen (udtrykt i Java) og sammenligner deres effektivitet på et konkret regneeksempel.

At algoritmen generelt er meget effektiv kan konstateres ved hjælp af såkaldt *algoritmeanalyse*. Med dette værktøj har man påvist, at algoritmen har et tidsforbrug, der vokser logaritmisk. Resultaterne fra denne analyse vises.

Der gives en kort indføring i logaritmer. Deres langsomme vækst fremhæves.

Efter gennemgangen af Euclids algoritme præciseres algoritmebegrebet. En algoritme defineres nu som en sekvens af operationer, der opfylder fire egenskaber: *endelighed*, *entydighed*, *effektfuldhed* og *korrekthed*. Vi ser på, om Euclids algoritme har disse egenskaber, og vi diskuterer, hvorvidt opskrifter i en kogebog er algoritmer efter denne definition.

Algoritmer skal på en eller anden måde kommunikeres. En række notationsformer for algoritmer omtales, bl.a. natursproglig beskrivelse, rutediagrammer og programmeringssproglig beskrivelse.

Forelæsningens anden del giver en introduktion til design og analyse af algoritmer. Det sker gennem løsning af følgende konkret algoritmiske problem.

Lad der være givet et stort netværk af computere, der kan kommunikere indbyrdes For hvert par af computere er oplyst, om der eksisterer en direkte forbindelse imellem dem. Vi ønsker at designe en algoritme, der hurtigt er i stand til at afgøre, om der eksisterer en forbindelse imellem to computere, enten i form af en direkte forbindelse eller i form af en serie af sådanne forbindelser. Dette problem er det såkaldte *Union-find-problem* i forklædning.

Vi ser på mulige løsningsalgoritmer og sammenligner deres effektivitet. For et netværk på størrelse med Internettet kan en af løsningsalgoritmerne risikere at skulle bruge flere tusind år, mens en anden af algoritmerne bruger under et minut!

Forelæsning 2: Elementære datastrukturer

Formålet med denne forelæsning er at give en indføring i elementære datastrukturer og abstrakte datatyper.

Forelæsningen indledes med definitioner af begreberne *type*, *datatype*, *abstrakt datatype* og *datastruktur*. Der lægges vægt på at fremhæve fordele ved brug af abstrakte datatyper. Deres realisering i Java omtales.

Nu følger en gennemgang af arrays. Vi ser på, hvorledes arrays noteres og tilgås i Java. Som et konkret eksempel på anvendelse gennemgås et Java-program, der implementerer "Eratosthenes si", en algoritme til generering af primtal. Effektiviteten af dette program undersøges ved hjælp af målinger af køretiden for programmet. Det vises, hvorledes sådanne tidsmålinger kan foretages i Java. Der gives en kort gennemgang af 2-dimensionale arrays, herunder notation, lagring og et simpelt eksempel på anvendelse.

Herefter defineres begrebet *hægtet liste*, og vi diskuterer fordele og ulemper ved at bruge hægtede lister i stedet for arrays. Implementering af hægtede lister med tilhørende grundoperationer (indsættelse og sletning) vises ved hjælp af konkret Java-kode.

Hvis der er tale om en enkelthægtet liste, kan visse operationer på listen ikke udføres effektivt. Her kommer brugen af dobbelthægtede lister ind i billedet. Vi ser på implementeringen af en generel Java-pakke til håndtering af dobbelthægtede lister.

Efter gennemgangen af de elementære datastrukturer, behandles de abstrakte datatyper *stak* og *kø*.

Vi ser, hvorledes en stak og en kø kan realiseres i Java. Den konkrete repræsentation har form af et array eller en hægtet liste.

En stak kan for eksempel benyttes til evaluering af aritmetiske udtryk. Det demonstreres, hvorledes en stak kan benyttes til at omskrive et sædvanligt aritmetisk udtryk til postfix-repræsentation, og hvorledes en stak derefter kan benyttes til at evaluere udtrykket ud fra dets postfix-repræsentation. Desuden vises, hvorledes en stak kan benyttes til håndtering af procedurekald i højere programmeringssprog.

Gennemgangen af de abstrakte datatyper kø og stak afrundes med en beskrivelse af deres realisering ved hjælp af Javas API-klasser `Vector` og `Stack`.

Anden del af forelæsningen omhandler træer. Den centrale terminologi, der knytter sig til træer, gennemgås, og der nævnes en række eksempler på anvendelse.

Et vigtigt særtilfælde af et træ er et såkald "binært træ". Faktisk kan ethvert træ repræsenteres som et binært træ.

Et parsetræ for et aritmetisk udtryk er et eksempel på et binært træ. Vi ser, hvorledes et sådan træ kan repræsenteres i Java, og hvorledes det er muligt at opbygge et parsetræ ud fra postfix-repræsentationen for det aritmetiske udtryk.

Endelig gennemgås, hvorledes et binært træ kan gennemløbes systematisk. Brug af stak og kø til dette formål demonstreres gennem konkret Java-kode.

Forelæsning 3: Design, verifikation og analyse af algoritmer

Design, *verifikation* og *analyse* er algoritmikkens tre grundelementer. Design omhandler metoder og teknikker til konstruktion af algoritmer. Verifikation benyttes til at påvise algoritmers korrekthed, mens analyse benyttes til at vurdere deres ressourceforbrug (tid og plads).

I denne forelæsning gives en introduktion til hvert af disse områder. Tiden tillader ikke, at der gås i dybden, men forhåbentlig kan forelæsningen give inspiration til videregående studier inden for områderne. Af forelæsningens indhold indgår kun rekursion og O-notation i pensum.

Design af algoritmer er en kreativ proces. Der findes ingen generel mekanisk metode til udvikling af en algoritme. Derimod findes der en række teknikker, eller rettere "tænkeregler", som ofte fører til både korrekte og effektive algoritmer. Nogle af disse teknikker er baseret på matematisk bevisførelse, mens andre blot har karakter af "gode råd", også kaldet "heuristikker". I forelæsningen vises eksempler på begge metodetyper.

Et eksempel på en heuristisk baseret teknik er Polyas problemløsningsteknik. Teknikken beskrives i korte træk, og vi ser, hvorledes teknikken kan bruges i praksis til løsning af et problem (det såkaldte *berømthedsproblem*).

Herefter ser vi på teknikker baseret på matematisk bevisførelse. *Bevisførelse ved modstrid* og *bevisførelse ved induktion* beskrives, og deres anvendelse illustreres ved hjælp af simple eksempler.

Induktion er ikke blot en bevisteknik. Induktion kan også benyttes konstruktivt til design af algoritmer. Dette illustreres gennem en række simple eksempler (sortering, beregning af polynomier og beregning af den maksimale delsekvenssum).

Del-og-hersk er en vigtig problemløsningsteknik. Brug af denne teknik vil ofte resultere i algoritmer, der er både overskuelige og effektive. Teknikkens grundide forklares og illustreres gennem et eksempel (potensopløftning).

Teknikken realiseres sædvanligvis ved hjælp af rekursion. Emnet rekursion behandles herefter indgående. Der fremhæves en række fordele ved brug af rekursion, og der gives en række eksempler på anvendelse. Vi ser såvel eksempler på rekursive datastrukturer (lister og binære træer) som på rekursive algoritmer (fakultetsfunktionen, gennemgang af binære træer, tårnene i Hanoi, søgning i en labyrint og fraktaler). For alle eksempler vises den komplette Java-kode.

Der gives en række tænkeregler til brug for udvikling af rekursive algoritmer. Det fremhæves, at disse regler har et nært slægtskab med induktion.

Rekursion har omkostninger i tid og plads, og i nogle tilfælde kan det være ønskeligt at fjerne rekursionen. Det påvises, at en rekursiv algoritme altid kan transformeres til en

ikke-rekursiv algoritme. Transformationsprocessen illustreres ved hjælp af eksempler: generel fjernelse af halerekursion og fjernelse af rekursionen i en preorder-gennemgang af et binært træ.

I anden del af forelæsningen tager vi fat på verifikation og analyse af algoritmer. Behovet for at kunne ræsonnere om algoritmer fremhæves, hvorefter de mest centrale begreber omkring algoritmeverifikaton defineres (partiel korrekthed, total korrekthed, programpåstande, før- og efter-betingelser og løkkeinvarianter). Der gives et simpelt eksempel på verifikation, nemlig et bevis for, at en algoritme til heltalsdivision er totalt korrekt.

Herefter gives en indføring i den såkaldte O-notation, en notation, der benyttes til at angive en øvre grænse for en algoritmes ressourceforbrug. Notationen beskrives såvel gennem en intuitiv definition som en matematisk stringent definition. Nogle af de vigtigste regneregler for O-notationen forklares, og der vises eksempler på brug i forbindelse med en række simple algoritmer. O-notationens begrænsninger nævnes.

Forelæsning 4: Sortering I

Ved sortering forstås en proces, hvor elementerne i en datamængde ordnes i rangorden. Tænk for eksempel på en telefonbog, hvor abonnenterne er ordnet alfabetisk efter deres navn.

Sortering indgår på en eller anden måde i de fleste programmer. Mange problemer kan nemlig løses mere effektivt, hvis inddata er sorteret. Som eksempler kan nævnes opslag i en telefonbog og bestemmelse af poster, der er fælles for to filer.

Denne forelæsning er den første af to forelæsninger om sortering. Forelæsningen omhandler følgende elementære metoder: *Sortering ved udvælgelse*, *Sortering ved indsættelse*, *Shellsort*, *Sortering ved tælling* og *Sortering ved adresseberegning*.

Indledningsvis gives såvel en uformel som en matematisk definition af sortering. Den matematiske definition er baseret på permutationer. En permutation er en nyttig matematisk abstraktion, ikke blot i forbindelse med sortering, men også i forbindelse andre typer af algoritmer.

Begreber som *intern* og *ekstern sortering* indføres (også selvom ekstern sortering ikke behandles yderligere i kurset).

Herefter beskrives de forskellige sorteringsalgoritmer. For hver algoritme beskrives det bagvedliggende princip, og en tilsvarende Java-metode implementeres. Vi ser algoritmerne i funktion ved hjælp af animering, og vi vurderer deres effektivitet ved hjælp af algoritmeanalyse og ved hjælp af empiriske tidsmålinger.

For metoderne Sortering ved udvælgelse og Sortering ved indsættelse gives såvel iterative som rekursive algoritmer. De rekursive udgaver fremkommer ved brug af induktion som designteknik (jævnfør forelæsning 3).

Forelæsningen afsluttes med en definition af *stabile* sorteringsmetoder samt en kort gennemgang af en teknik til sortering af ”store” poster.

Forelæsning 5: Sortering II

I denne forelæsning gennemgås en række avancerede sorteringsmetoder. Metoderne er en smule vanskeligere at implementere end de elementære metoder, men de udmærker sig ved at være meget effektive i praksis.

I første del af forelæsningen behandles Quicksort og Mergesort. For begge metoder beskrives de bagvedliggende principper, og en tilsvarende Java-metode implementeres. Algoritmerne animeres, og deres tidsforbrug bestemmes (såvel analytisk som empirisk).

Vi ser, hvorledes effektiviteten af en basal udgave af Quicksort kan forbedres ved en række simple ændringer. I tilknytning til Quicksort, ser vi også, hvorledes algoritmens grundbestanddele kan benyttes til effektivt at udvælge det k'te mindste element af en mængde.

Både Quicksort og Mergesort er baseret på del-og-hersk-teknikken, og det demonstreres, hvorledes begge algoritmer kan udledes ud fra en og samme skabelon.

Anden del af forelæsningen indledes med en introduktion til prioritetskøer. Vi ser, hvorledes en prioritetskø kan implementeres i Java. Som datastruktur kan vi for eksempel benytte et uordnet eller et ordnet array. Mere effektivt er det dog, at benytte sig af en såkaldt *binær hob*.

Vi ser, at brugen af en binær hob muliggør effektive algoritmer for både indsættelse og sletning i en prioritetskø, og at en binær hob kan realiseres ved hjælp af et array. Den komplette Java-kode for indsættelse og sletning gennemgås. Desuden angives en lineær algoritme til konstruktion af en binær hob.

En binær hob kan benyttes til at opnå en sorteringsmetode, Heapsort, med et garanteret køretidsforbrug, der er O(n logn). Det nævnes, at enhver sorteringsalgoritme, der er baseret på nøglesammenligninger, nødvendigvis kræver mindst O(n logn) sammenligninger i værste tilfælde.

Radixsortering er en sorteringsmetode, der ikke benytter nøglesammenligninger. Metoden er lineær i tid (som en funktion af antallet af poster). Den beskrives kort, men indgår ikke i pensum.

Forelæsningen afsluttes med en oversigt over kriterier for valg af sorteringsmetode.

Forelæsning 6: Søgning I

Søgning omhandler genfinding af lagret information. I mange programmer er søgning den mest tidsforbrugende proces. Kendskab til effektive søgemetoder er derfor særdeles nyttigt.

Denne forelæsning er den første af to forelæsninger om søgning. Forelæsningen omhandler *sekventiel* og *binær* søgning.

Indledningsvis defineres begrebet søgning, og der gives en overordnet klassifikation af søgemetoder.

Herefter introduceres en *ordbog* (engelsk: dictionary), en abstrakt datatype til søgning. Det konstateres, at valg af bagvedliggende datastrukturer er af afgørende muligheden for at opnå effektiv søgning. Hvis vi benytter et usorteret array eller en usorteret liste, må vi indskrænke os til sekventiel søgning, en metode, der er lineær i tid. Hvis datastrukturen derimod er et sorteret array, kan vi benytte binær søgning og opnå logaritmisk tidsforbrug. Vi ser også, at det er muligt at opnå dobbeltlogaritmisk tidsforbrug ved brug af interpolationssøgning. Den konkrete Java-kode, der skal til for at realisere disse søgemetoder, gennemgås i detaljer.

Dernæst ser vi på *binære søgetræer*. Operationerne *indsættelse*, *søgning* og *sletning* i et binært søgetræ illustreres ved hjælp af figurer, og de implementeres i Java. Såvel iterative som rekursive udgaver af algoritmerne præsenteres.

I anden del af forelæsningen ser vi på, hvorledes et binært søgetræ kan ”balanceres”, så tidsforbruget for operationerne bliver logaritmisk.

Først forklares grundideen bag de såkaldte *2-3-4-træer* ved hjælp af figurer, og det konstateres, at såvel tiden for indsættelse og søgning er logaritmisk. Implementeringen af indsættelse i 2-3-4-træ skitseres i Java.

Formålet med at introducere 2-3-4-træer er primært at forenkle forklaringen af operationerne i et *rød-sort-træ*, en af de mest effektive datastrukturer til at opnå balancerede binære søgetræer. Ethvert 2-3-4-træ kan nemlig transformeres til et rød-sort-træ, og omvendt. Så alle operationer i et rød-sort-træ kan umiddelbart forklares ud fra operationer i det tilsvarende 2-3-4-træ.

Vi ser på en Java-implementation af indsættelse i et rød-sort-træ. Koden benytter sig af rekursion og forekommer at være lettere at forstå end den iterative kode, der er angivet i lærebogen.

Forelæsningen afsluttes med en oversigt over nogle andre datastrukturer til realisering af binære søgetræer (bl.a. AVL-træer og splay-træer)

Forelæsning 7: Søgning II

I denne forelæsning gennemgås søgning ved brug af nøgletransformation, også kaldet *hashing*.

Efter en kort beskrivelse af den grundlæggende ide, ser vi på de principper, der skal lægges til grund ved konstruktion af hashfunktioner. Der gives eksempler på hashfunktioner, og der redegøres for, hvorfor hashtabellens størrelse normalt bør vælges som et primtal.

At to nøgler transformeres til den samme tabelindgang kaldes for en *kollision*. Kollisioner kan normalt ikke undgås, men de kan behandles mere eller mindre effektivt. Vi ser på en række kollisionsstrategier (*Separat kædning*, *Lineær prøvning* og *Dobbelt hashing*). Fordele og ulemper ved strategierne diskuteres.

Java indeholder klassen `Hashtable`, som gør det let at bruge hashing i Java-programmer. Koden for denne klasse gennemgås.

Hashing er en meget effektiv søgemetode. Med en vis påpasselighed kan opnås et tidsforbrug, der er uafhængigt af antallet af poster i filen! Hashing bør dog ikke altid vælges som søgemetode. Der gives en række grunde til at andre metoder, f.eks. metoder baseret på binære søgetræer, foretrækkes i nogle situationer.

Forelæsning 8: Strengbehandling

Denne forelæsning omhandler to emner inden for området strengbehandling, nemlig *strengsøgning* og *syntaksanalyse*.

Første del af forelæsningen omhandler strengsøgning. Efter præsentationen af en række grundlæggende begreber, der knytter sig til strenge, præsenteres den opgave, der ønskes løst , nemlig søgning efter en forekomst af en streng i en given tekst. Opgaven er velkendt fra tekstbehandlingssystemer.

Først præsenteres en simpel algoritme, der benytter sig af "rå kraft" til at løse opgaven. Algoritmen præsenteres i Java, og dens gennemsnitlige og maksimale tidsforbrug bestemmes. Vi ser nærmere på den metode, der tilbydes til strengsøgning i Javas API, metoden `indexOf` i klassen `String`. Det konstateres, at algoritmen foretager mange overflødige tegnsammenligninger, og at der således er mulighed for forbedringer.

En sådan forbedring udgør Knuth-Morris-Pratt-algoritmen. Denne algoritme gennemgås i detaljer. Vi ser, gennem et kørselseksempel, at effektiviteten er forbedret i forhold til den simple algoritme.

Det påvises, at algoritmen kan baseres på brugen af *endelige tilstandsmaskiner*. Desuden vises et andet eksempel på praktisk brug af endelige tilstandsmaskiner, nemlig *tabelstyret* indlæsning af decimaltal.

Derefter gennemgås en anden algoritme til strengsøgning, nemlig Boyer-Moore-algoritmen. Algoritmen regnes normalt for at være en af de mest effektive algoritmer til formålet. Endelig gennemgås Rabin-Karp-algoritmen. Denne algoritme er forholdsvis let at implementere og er interessant, fordi den benytter sig af hashing.

Denne del af forelæsningen afrundes med forevisning af resultaterne fra en empirisk undersøgelse af de gennemgåede algoritmers effektivitet.

I forelæsningens anden del beskæftiger vi os med syntaksanalyse af strenge. Vi ser på to værktøjer til at beskrive grammatikken for et sprog, nemlig *BNF-notation* og *syntaksdiagrammer*. Herefter udvikles et Java-program, der er i stand til at afgøre, om en indlæst streng er et aritmetisk udtryk. Programmet benytter sig af såkaldt *rekursiv nedstigning* og er et illustrativt eksempel på brugen af rekursion.

Til sidst udvides programmet, så det kan beregne værdien af et indlæst aritmetisk udtryk.

Forelæsning 9: Grafalgoritmer I

En ”graf” er nyttigt abstrakt begreb. En graf benyttes til at beskrive relationer imellem data.

En graf er et kraftfuldt værktøj til modellering og benyttes f.eks. til modellering af datanetværk, projektplanlægning og diagrammering (f.eks. UML-diagrammering).

Denne forelæsning er den første af to forelæsninger om grafer.

Først introduceres den terminologi, der knytter sig til grafer. Herefter ser vi på forskellige datastrukturer til repræsentation af grafer. Repræsentation ved hjælp af *kantmængde, nabomatrix* og *nabolister* gennemgås i detaljer. Vi ser på Java-koden for de tre repræsentationsformer, opgør deres pladsbehov og diskuterer deres hensigtsmæssighed i forhold til løsning af en række basale grafproblemer.

Anden del af forelæsningen omhandler metoder til søgning i grafer. Vi ser på systematisk gennemgang af en graf ved hjælp af *dybde-først-* og *bredde-først-søgning*.

Java-koden for de to søgestrategier gennemgås, og søgeprocessen visualiseres. Vi ser, hvorledes dybde-først-søgning kan anvendes til at løse visse problemer vedrørende s*ammenhæng* i grafer. For eksempel kan samtlige sammenhængende komponenter i en graf bestemmes ved dybde-først-søgning.

Forelæsning 10: Grafalgoritmer II

Denne forelæsning omhandler en række centrale algoritmer for *vægtede* grafer og for *orienterede* grafer.

Første del af forelæsningen omhandler algoritmer til løsning af to praktiske problemer i tilknytning til vægtede grafer, nemlig bestemmelse af *et minimalt udspændende træ* og bestemmelse af en *korteste vej* imellem to knuder i en graf.

Det vises, at begge problemer kan løses ved hjælp af bedste-først-søgning.

Mens dybde-først- og bredde-først-søgning anvender sig af henholdsvis en stak og en kø, anvender bedste-først-søgning sig af en prioritetskø. Vi ser, at dybde-først- og bredde-først-søgning blot er specialtilfælde af bedste-først-søgning.

Der præsenteres to algoritmer til bestemmelse af et minimalt udspændende træ, nemlig Prims algoritme og Kruskals algoritme. Algoritmerne har forskellig kompleksitet, men er begge baseret på anvendelse af en og samme grafteoretiske sætning.

Der præsenteres en algoritme til bestemmelse af alle korteste veje fra en given knude til enhver anden knude, nemlig Dijkstras algoritme. Det observeres, at Prims algoritme og Dijkstras algoritme kun adskiller sig kun ved deres specifikation af prioritet.

De gennemgåede algoritmer er alle eksempler på såkaldte ”grådige” algoritmer, d.v.s. algoritmer, der opnår optimum ved i hvert skridt at gøre det, der aktuelt synes at være bedst, uden hensyn til fremtidige konsekvenser. For samtlige algoritmer gennemgås den tilhørende Java-kode. Udførelsen visualiseres, og algoritmernes kompleksitet beregnes.

I anden del af forelæsningen ser vi på en række fundamentale algoritmer, der knytter sig til orienterede grafer.

Vi ser først, hvorledes dybde-først-søgning kan benyttes til at finde alle de knuder, der kan nås fra en given knude.

Hvis der er mange søgninger af denne art, kan det være en fordel, at bestemme grafens såkaldte *transitive afslutning*. Den transitive afslutning opnås ved at udvide grafen med orienterede kanter imellem de par af knuder, hvor der findes en vej fra den ene knude til den anden. Vi ser på Warshalls algoritme til effektiv bestemmelse af den transitive afslutning i grafer, der er repræsenteret ved en nabomatrix. Muligheden for at benytte parallelitet i implementeringen illustreres ved brug af Javas indbyggede klasse `BitSet`. Det påvises, at Floyds algoritme til bestemmelse af alle korteste veje i en vægtet, orienteret graf fremkommer ved en simpel ændring af Warshalls algoritme.

Til sidst omtales en vigtig delmængde af orienterede grafer, nemlig *orienterede grafer uden cykler* (DAGs). Sådanne grafer har mange praktiske anvendelser. For eksempel benyttes de til netværksplanlægning, en teknik til at foretage beregninger på planlægnings-

opgaver. Ved hjælp af netværksplanlægning kan man blandt andet bestemme varigheden af et projekt samt bestemme projektets ”kritiske” aktiviteter, d.v.s. de aktiviteter, for hvilke en forsinkelse vil bevirke, at hele projektet forsinkes.

Forud for disse beregninger foretages en såkaldt *topologisk sortering* af grafens knuder. Knuderne ordnes på en række, så alle orienterede kanter vender samme vej. Vi ser på to forskellige algoritmer til topologisk sortering (algoritmen fra lærebogen, samt en noget simplere).

Forelæsning 11: Parallelle algoritmer. Dynamisk programmering

De fleste computere i dag er variationer over den såkaldte *von Neumann model*: instruktioner og data er lagret i samme lager, og én processor henter instruktioner fra lageret og udfører dem en ad gangen.

Imidlertid vinder en række nye maskiner frem, som tillader udførelse af mange instruktioner samtidigt (i parallel). Disse maskiner stiller krav om kendskab til algoritmer, der kan udnytte en sådan parallelitet.

Forelæsningen giver en introduktion til parallelitet i programmel og maskinel. Det bemærkes, at det gennemgåede stof ikke indgår i pensum.

Indledningsvis gives et eksempel på en parallel algoritme, nemlig en algoritme til addition af to heltal. Java-koden for algoritmen gennemgås. Herefter skitseres problemerne ved parallelisering, problemer vedrørende kommunikation og synkronisering. Vi ser, hvorledes synkronisering kan opnås i Java ved hjælp af såkaldte *kritiske regioner*.

Efter en præsentation af Flynns klassifikation af maskinarkitekturer, gennemgås nogle specialbyggede maskiner. Vi ser på netværk til sortering og netværk til fletning. Parralleliteten i et netværk til fletning simuleres i Java ved brug af Javas trådbegreb.

Til sidst i denne del af forelæsningen vises, hvorledes matrixmultiplikation kan udføres i parallel på en såkaldt *systolisk maskine*. Også denne maskine simuleres i Java.

Forelæsningens anden del giver en introduktion til *dynamisk programmering*, en meget nyttig problemløsningsteknik. Hvor del-og-hersk-teknikken er en top-til-bund-teknik (et problem løses ved opdele det i uafhængige delproblemer, som så løses separat), er dynamisk programmering i princippet en bund-til-top-teknik (et stort problem løses ved at løse alle mindre (gerne overlappende) delproblemer, gemme deres løsning og benytte løsningerne til at løse større problemer).

Dynamisk programmering kan realiseres ved hjælp af såvel iteration som rekursion. Det er ofte simplest at benytte rekursion. Genberegning af løsningsresultater undgås i dette tilfælde ved at huske de beregnede resultater, f.eks. i en tabel.

Brugen af dynamisk programmering illustreres først gennem et simpelt eksempel, nemlig et Java-program til beregning af Fibonaccital. Derefter implementeres en algoritme til optimal veksling af mønter. Opgaven går ud på at veksle et beløb til mønter, således at antallet af mønter er minimalt.

I den udstrækning tiden tillader, vises en række andre eksempler på brug af dynamisk programmering (løsning af rygsækproblemet, konstruktion af optimale binære søgetræer, løsning af matrix-kædeprodukt-problemet).

Forelæsning 12: Udtømmende søgning. Problemkompleksitet

Denne forelæsning behandler to emner. Det første emne, *udtømmende søgning*, omhandler den opgave at foretage en systematisk gennemsøgning af alle potentielle løsninger af et problem.

Som udgangspunkt vælges et klassisk kombinatorisk optimeringsproblem, nemlig *Den rejsende sælgers problem*. Problemet går ud på at finde den korteste rejserute for en sælger, der skal besøge en række byer og vende tilbage til sit udgangspunkt.

Det konstateres, at problemet i princippet kan løses ved at generere samtlige rundture og derefter vælge den korteste blandt disse. Vi ser, at samtlige rundture kan bestemmes ved en simpel ændring af algoritmen til dybde-først-søgning i en graf. Vi konstruerer et komplet Java-program, der løser problemet. Dette program er dog ikke særligt effektivt, så vi undersøger forskellige muligheder for at beskære søgetræet (bl.a. ved at undgå symmetriske løsninger og ved brug af den såkaldte ”branch-and-bound-teknik”). Derved mindskes søgearbejdet, men ikke tilstrækkeligt til, at det i praksis bliver muligt at finde optimale løsninger, med mindre problemer er relativt små.

I mange situationer behøver man imidlertid ikke en optimal løsning - en rimelig kort tur kan være fuldt tilfredsstillende. Her kommer *approksimative* algoritmer ind i billedet. En approksimativ algoritme tilstræber at opnå en ”rimeligt god” løsning på kort tid, f.eks. ved brug af en eller anden ”heuristik” (tommelfingerregel). Nogle simple algoritmer til approksimativ løsning af den rejsende sælgers problem skitseres.

Baksporing er en generel teknik til systematisk generering af alle mulige løsninger til et kombinatorisk problem. Der præsenteres en skabelon, der kan benyttes til at implementere algoritmer, der benytter baksporing. Brugen af denne skabelon demonstreres ved en løsning af det såkaldte *8-dronninge-problem*.

Denne del af forelæsningen afsluttes med gennemgang af to algoritmer til generering af permutationer.

Anden del af forelæsningen beskæftiger sig med *problemkompleksitet*.

For en stor klasse af problemer er det påvist, at det ikke vil være muligt af udvikle effektive algoritmer (d.v.s. algoritmer, der løser problemet i polynomiel tid). Det er nyttigt at have et kendskab til disse problemer, fordi det vil være omsonst at forsøge at udvikle effektive algoritmer for dem.

Først gives en oversigt over sådanne svære problemer. Herefter introduceres mængden *P* af problemer, der kan løses i polynomiel tid på en sædvanlig, deterministisk maskine, samt mængden *NP* af problemer, der kan løses i polynomiel tid på en ikke-deterministisk maskine.

Problemer, der tilhører *NP*, men ikke *P*, er lette at løse for en ikke-deterministisk maskine, men, trods ihærdige forsøg, har ingen været i stand til at vise, at de kan løses effektivt på en konventionel maskine. Efterhånden mener alle dataloger, at det ikke kan lade sig gøre. Men det er aldrig blevet bevist (eller modbevist). Problemerne har den egenskab, at hvis blot et af dem kan løses effektivt, så kan de alle løses effektive. Problemerne fra denne klasse siges at være *NP-komplette*. Det viser sig, at et stort antal praktiske problemer tilhører denne klasse af problemer.

Til sidst i forelæsningen nævnes eksempler på problemer, som ikke kan løses algoritmisk (heriblandt det såkaldte *stopproblem*).