Introduksjon til IN2010 – Algoritmer og datastrukturer

1. Litt om algoritmer En algoritme er en presis og endelig beskrivelse av hvordan et problem kan løses steg for steg. Det er en oppskrift som forteller hvordan input transformeres til output. Kjennetegn ved en algoritme • Presis og entydig: Hvert steg må være klart definert, uten tvetydighet. • Endelighet: Algoritmen må terminere etter et bestemt antall steg; den kan ikke kjøre evig. • Input: Kan ta null eller flere inputverdier. • Output: Må produsere et resultat som står i forhold til input. • Effektivitet: Skal bruke ressurser (tid og minne) på en fornuftig måte. Eksempel • Oppskrift for å lage en kopp te kan betraktes som en algoritme: 1. Kok vann. 2. Sett tepose i kopp. 3. Hell kokende vann over teposen. 4. Vent i 3–5 minutter. 5. Fjern tepose og server. Viktige poenger • Algoritmer er universelle: samme problem kan ha mange forskjellige algoritmer. • Effektive algoritmer løser problemet raskt og bruker lite ressurser. • Analyse av algoritmer handler om tidskompleksitet og plasskompleksitet.

2. Litt om datastrukturer En datastruktur er en organisert samling av data som følger en bestemt struktur, slik at data kan lagres, hentes og endres effektivt. Eksempler på datastrukturer • Lister / arrays: Sekvensiell lagring av elementer. Enkel tilgang via indeks. • Stack (stabel): LIFO-prinsipp (Last In, First Out). • Queue (kø): FIFO-prinsipp (First In, First Out). • Trær: Hierarkisk struktur, f.eks. binærtre, AVL-tre. • Grafer: Noder og kanter, brukes for nettverk og relasjoner. • Hashtabeller: Hurtig oppslag basert på nøkkel. Sammenheng mellom algoritmer og datastrukturer • Algoritmer trenger effektive datastrukturer for å være raske og enkle. ○ Eksempel: Sortering av data er enklere med arrays enn med en koblet liste. • Datastrukturer trenger gode algoritmer for å holde data konsistente og lette å bruke. ○ Eksempel: Å søke i et balansert binærtre er effektivt kun hvis algoritmen for innsetting holder treet balansert. God praksis • Velg datastruktur som gjør de mest brukte operasjonene raske. • Tenk på kompleksitet: hvor raskt kan man søke, legge til eller fjerne elementer? • Effektiv datastruktur kan redusere behovet for komplekse algoritmer, og omvendt.

Oppsummering: • Algoritmer er steg-for-steg-oppskrifter for å løse problemer. • Datastrukturer er måten vi organiserer og lagrer data på, og de er fundamentale for at algoritmer skal fungere effektivt. • God forståelse av begge konseptene er nøkkelen til effektiv programmering og problemløsning.

Perspektiv på algoritmer og programmering

1. Fokusområder Målet i kurset er å lære å: • Forstå algoritmer: Hvorfor fungerer de? Hvordan oppfører de seg? • Analysere algoritmer: Beregne effektivitet, tids- og plasskompleksitet. • Anvende algoritmer: Velge eller lage passende algoritmer for problemer. • Lage algoritmer: Bygge nye løsninger når eksisterende ikke passer.
2. Metoder for å bygge forståelse • Studer mange konkrete eksempler. • Implementer algoritmer i kode for å se hvordan de fungerer i praksis. • Analyser algoritmer for å forutsi hvor effektive de er i ulike situasjoner (stor vs liten input, beste/verste tilfelle).
3. Praktisk tilnærming • Når du løser problemer: finn først en algoritme som passer. • Hvis algoritmen ikke finnes, må du lage din egen: ○ Studer lignende problemer. ○ Del opp problemet eller løs en enklere versjon først. • Øvelse gjør mester: bruk online plattformer for å trene algoritmer: ○ Kattis ○ LeetCode ○ Project Euler ○ CSES Problemset
4. Tips for god progresjon • Vær aktiv i grupper og diskusjoner – lær av andre. • Skriv kode hver dag, selv små eksperimenter. • Lag egne oppgaver og løs dem – øv på kreativ problemløsning. • Gjør

algoritmelæring til en hobby, ikke bare en plikt.

Abstrakte datatyper (ADT)

1. Hva er en abstrakt datatype? • En ADT beskriver hvordan data skal oppføre seg, ikke hvordan den er implementert. • Den spesifiserer: ○ Hvilke operasjoner som er tilgjengelige (f.eks. legg til, fjern, søk). ○ Forventet oppførsel (f.eks. FIFO for en kø, LIFO for en stabel). • Den skiller konseptet fra implementasjonen.
2. Relasjon mellom ADT og datastrukturer • En ADT kan ha flere konkrete implementasjoner: ○ Eksempel: en liste kan implementeres med: § Array (sekvensiell lagring) § Koblet liste (linkede elementer) • En konkret datastruktur realiserer ADT-en i kode. • I Java brukes interface for å definere ADT-er: interface Stack { void push(T element); T pop(); boolean isEmpty(); }
3. Hvorfor ADT er viktig i IN2010 • Kurset handler mye om å finne effektive implementasjoner for sentrale ADT-er. • Eksempler på sentrale ADT-er: ○ Stack (stabel) – LIFO ○ Queue (kø) – FIFO ○ List (liste) – sekvensiell tilgang ○ Priority Queue (prioritetskø) – elementer med prioritet ○ Map / Dictionary (nøkkel-verdi) – oppslag basert på nøkkel Viktige poenger • ADT gjør at vi kan fokusere på algoritmens logikk uten å bekymre oss for lavnivådetaljer i implementasjonen. • Effektive datastrukturer og algoritmer går hånd i hånd.

Stack (Stabel)

1. Definisjon • En stack er en LIFO (Last-In, First-Out) datastruktur. • Elementer legges alltid til på toppen, og tas alltid ut fra toppen. (tallerkener på hoteller).
2. Operasjoner Operasjon Beskrivelse push(x) Legg element x på toppen av stacken pop() Fjern og returner elementet som ligger på toppen peek() (ofte tilgjengelig) Se på elementet på toppen uten å fjerne det isEmpty() Sjekk om stacken er tom
3. Illustrasjon Stack-top x3 <- sist lagt inn x2 x1 Stack-bottom • push(x4) → x4 blir øverst • pop() → x4 fjernes, x3 blir ny topp
4. Implementasjon • Lenket liste: Hver node peker til neste node. Fordel: dynamisk størrelse, enkel push/pop. • Dynamisk array: Elementer lagres i array. Fordel: rask tilgang til topp-element, men må eventuelt utvide arrayen når kapasiteten overskrides.

Set (Mengde)

1. Definisjon • En abstrakt datatype for mengder, hvor rekkefølge og antall forekomster ikke spiller rolle. • Elementer forekommer maks én gang.
2. Operasjoner Operasjon Beskrivelse Insert(A, x) Sett inn element x i mengden A Remove(A, x) Fjern element x fra A $x \in A$ Sjekk om x er medlem av A Union(A,B) Slå sammen A og B → $A \cup B$ Intersection(A,B) Felles elementer → $A \cap B$ Difference(A,B) Elementer i A men ikke i B → $A \setminus B$ Difference(B,A) Elementer i B men ikke i A → $B \setminus A$
3. Implementasjon • HashSet: Rask tilgang basert på hashing, uordnet. • TreeSet: Ordnet (sortert) mengde, implementert med balanserte trær.
4. Variasjon: OrderedSet • Samme operasjoner som Set. • Elementer itereres i sortert rekkefølge.

Map (Ordbok / Dictionary)

1. Definisjon • Et Map assosierer hver nøkkel med én verdi. • Hver nøkkel er unik; verdien kan endres.
2. Operasjoner Operasjon Beskrivelse put(key, value) Sett inn nøkkel-verdi-par i map get(key) Hent verdien assosiert med nøkkelen remove(key) Fjern nøkkel-verdi-par basert på nøkkel

containsKey(key) Sjekk om nøkkelen finnes
3. Illustrasjon Key: ♣ ♢ ♠ ♡ Value: 0 1 2 3 • put(♣, 5) → verdien for ♣ endres til 5 • get(♠) → returnerer 2
4. Implementasjon • HashMap: Rask tilgang basert på hashing, uordnet. • TreeMap: Ordnet map, sortert etter nøkkel.
5. Variasjon: OrderedMap • Samme operasjoner som Map. • Nøklene itereres i sortert rekkefølge.

Pseudokode

1. Hva er pseudokode? • Pseudokode er en mellomting mellom naturlig språk og programmeringskode. • Hensikten er å formidle algoritmer på en måte som er lett å lese og forstå, men som ikke nødvendigvis kjører i et spesifikt språk. • Det er et verktøy for å: ○ Planlegge algoritmer ○ Diskutere løsninger ○ Forberede implementering i språk som Python, Java, C++, etc.
2. Notasjon i pseudokode • Aritmetiske uttrykk: a + b - c • Sammenligninger: a ≤ b, x = y • Tilordninger: i ← 0 (i får verdien 0) • Størrelse på datastrukturer: |A| = antall elementer i array eller liste A • While-løkker: while test do body • For-løkker: for i ← 0 to n–1 do body • Pseudokoden skal være lett å oversette til ekte programmeringsspråk.

Binærsøk – spesifikasjon

1. Formål • Algoritmen skal avgjøre om et element x finnes i et array A.
2. Input og output • Input: ○ Array A (sortert for binærsøk) ○ Element x (søkeverdi) • Output: ○ true dersom x ∈ A ○ false ellers
3. Krav til algoritmen • Algoritmen må terminere på et endelig antall steg. • Algoritmen må gi riktig svar for alle gyldige input. • Dette er en formell spesifikasjon, som angir hva algoritmen skal gjøre, men ikke hvordan den gjør det.

Lineært (rett-frem) søk – implementasjon

1. Algoritme ALGORITHM: RETT FREM SØK Input: Array A, element x Output: true hvis x ∈ A, ellers false Procedure Search(A, x) for i ← 0 to |A| – 1 do if A[i] = x then return true return false

2. Forklaring • Algoritmen går gjennom alle elementene i arrayet til den finner x. • Dersom x ikke finnes, sjekker den hele arrayet og returnerer false. • Bruker 0-indekserte arrayer (A[0] er første element).

3. Effektivitet • Verste tilfelle: x finnes ikke, eller ligger sist i arrayet. • Antall sammenligninger i verste tilfelle = |A|. • Dette er en lineær algoritme, med tidskompleksitet O(n).

4. Viktige poeng for eksamen • Pseudokode må alltid være presis, entydig og lett å følge. • Spesifikasjon og implementasjon er to forskjellige steg: først definerer vi hva algoritmen skal gjøre, deretter hvordan. • Lineært søk er enkelt, men ineffektivt for store datasett → her kommer binærsøk som effektiv variant når arrayet er sortert.

Binærsøk – spesifikasjon

1. Formål • Algoritmen skal avgjøre om et element x finnes i et ordnet array A.
2. Input og output • Input: ○ Array A, sortert i stigende rekkefølge (A[i] ≤ A[j] for alle i ≤ j) ○ Element x (søkeverdi) • Output: ○ true dersom x ∈ A ○ false ellers

3. Viktige poenger • Arrayet må være ordnet – det er forutsetningen som gjør binærsøk mulig. • Et rett-frem (lineært) søk vil også fungere, men det er mindre effektivt. • Ved å utnytte at arrayet er sortert, kan vi gjøre søket mye raskere, med færre sammenligninger.

Binærsøk – idé

1. Tenk som en ordliste • Metoden ligner på hvordan du slår opp et navn i en telefonbok eller ordbok: 1. Start i midten 2. Sjekk om elementet du søker etter er større eller mindre enn midten 3. Kast bort halvdelen av listen basert på sammenligningen 4. Fortsett på den gjenværende halvdelen • Utfordringen: formulere dette som entydige steg, slik at det kan oversettes til pseudokode eller programmeringskode.
2. Fordel • Hver gang vi deler listen i to, halveres antall kandidater. • Tidskompleksiteten blir O(log n) i stedet for O(n) som ved lineært søk.

Binærsøk – implementasjon Pseudokode ALGORITHM: BINÆRSØK Input: Ordnet array A, element x Output: true hvis x ∈ A, ellers false Procedure BinarySearch(A, x) low ← 0 high ← |A| – 1 while low ≤ high do i ← ⌊(low + high)/2⌋ // midtpunkt if A[i] = x then return true else if A[i] < x then low ← i + 1 // søk i høyre halvdel else high ← i – 1 // søk i venstre halvdel return false Forklaring • low og high definerer søkeintervallet i arrayet. • i er midtpunktet av intervallet, rundet ned (⌊ ⌋). • Sammenlign A[i] med x: ○ Hvis lik → true ○ Hvis mindre → kast venstre halvdel (low ← i + 1) ○ Hvis større → kast høyre halvdel (high ← i – 1) • Hvis intervallet krymper til ingenting (low > high), finnes ikke elementet → returner false. Effektivitet • Hver iterasjon halverer antall kandidater → logaritmisk tidskompleksitet: O(log n) • Meget effektivt for store datasett sammenlignet med lineært søk (O(n)).

Introduksjon til kjøretidskompleksitet

1. Hastighet vs. effektivitet • Å si at et program er “raskt” er ikke presist, fordi hastigheten påvirkes av maskinen programmet kjører på. ○ Et program kan kjøre raskt på en ny datamaskin, men tregt på en eldre maskin. • For å kunne sammenligne algoritmer uavhengig av maskinvare, snakker vi om effektivitet, ikke ren hastighet.
2. Effektivitet • En algoritme er effektiv hvis den ikke gjør unødvendig arbeid. • Effektivitet vurderes ut fra antall steg algoritmen bruker i forhold til størrelsen på input. • Effektivitet er et egenskap ved algoritmen – ikke maskinen den kjører på.
3. Ytelse • Ytelse handler om hvor raskt arbeidet faktisk utføres på en konkret maskin. • En algoritme kan være effektiv, men ytelsen vil variere basert på: ○ Maskinvaren (CPU, RAM, lagring) ○ Kompilatoren/interpreter ○ Driftsforhold (f.eks. temperatur, samtidige prosesser) • I IN2010 fokuserer vi primært på effektivitet, fordi en effektiv algoritme gir god ytelse uansett maskin.

Kompleksitet

1. Ressurser • Kompleksitet handler om hvor mye ressurser algoritmen bruker. • Viktigste ressurser: 1. Tid – antall steg algoritmen trenger 2. Minne – hvor mange elementer som lagres i datastrukturer
2. Kjøretidskompleksitet • Kjøretidskompleksitet måler hvor mange steg som trengs for å fullføre algoritmen, som en funksjon av inputstørrelsen n. • Dette gir en maskinuavhengig vurdering av effektivitet.
3. Minnekompleksitet • Måler hvor mye plass som brukes, for eksempel: ○ Antall elementer i lister, trær, tabeller ○ Ekstra minne for rekursive kall eller buffer

Størrelse på input • Input kan variere i størrelse: ○ Kort eller lang binærstreng ○ Liste med få eller mange noder ○ Array med få eller mange elementer ○ Mengde med få eller mange elementer ○ Tall med lav eller høy verdi • Vi angir inputstørrelse med variabelen n ○ n representerer enheter relevant for algoritmen, f.eks. antall elementer i et array.

Telle steg Primitive steg • Vi vurderer disse operasjonene som én tidsenhet hver: ○ Tilordninger: x ← 3 ○ Aritmetiske operasjoner: a + b, a - b ○ Sammenligninger: a < b, a = b ○ Aksessering i array: A[i] ○ Returnering: return a Løkker og funksjonskall • While-løkker: kostnaden = test + kropp _ antall iterasjoner • For-løkker: tilsvarende, kostnad = summen av steg i kroppen _ antall iterasjoner • Prosedyre-/metodekall: arver kostnaden fra funksjonen som kalles Formål • Ved å telle primitive steg kan vi modellere algoritmens kjøretid som en funksjon av n. • Dette gir grunnlaget for å sammenligne algoritmer objektivt, uavhengig av maskin.

En enkel omskrivning: for → while For-løkke 1 Procedure Search(A, x) 2 for i ← 0 to |A|–1 do 3 if A[i] = x then 4 return true 5 return false Omskrevet til while-løkke 1 Procedure Search(A, x) 2 i ← 0 3 while i < |A| do 4 if A[i] = x then 5 return true 6 i ← i + 1 7 return false • Funksjonaliteten er identisk. • Forskjellen er kun i syntaks: for-løkke har intern iterasjonskontroll, mens while-løkke gjør det eksplisitt med i ← i + 1. • Begge kan analyseres på samme måte for kjøretid.

Telle steg: eksempel Anta: • n = |A| (størrelsen på arrayet) • Elementet x er ikke i arrayet (verste tilfelle) Analyse av primitive steg Operasjon Antall steg per iterasjon Forklaring Tilordning i ← 0 1 Før løkken starter Test `i < A` If-test A[i] = x n 1 sammenligning + 1 aksessering per iterasjon = 2 steg Inkrement i ← i + 1 n 1 addisjon + 1 tilordning = 2 steg Return return false 1 Når løkken avsluttes Samlet antall steg • Formelt: $1+(n+1)+n \cdot 4+1=5n+3$ • Her teller vi alle primitive steg som tilordninger, aksesseringer, sammenligninger og returnering.

Verste tilfelle • Verste tilfelle = maksimal kjøretid algoritmen kan bruke. • Ofte brukes dette i analyser fordi: ○ Gir sikkerhet for at algoritmen aldri bruker mer enn et visst antall steg. ○ En gjennomsnittsanalyse krever sannsynlighetsfordeling over alle mulige input, som ofte er komplisert eller upraktisk. • Spørsmålet vi ønsker å besvare: ○ Hvordan skalerer algoritmen når input-størrelsen n blir veldig stor?

Store O-notasjon • Big O er et verktøy for å beskrive kjøretidens vekst når n blir stor. • Vi ser bort fra: ○ Konstanter ○ Ledd med lavere orden Eksempel • Antall steg: 5n + 3 • I Big O: $O(5n+3)=O(n)$ • Intuisjon: ○ Konstantfaktor (5) og små ledd (+3) betyr lite når n blir stort ○ Big O bevarer hovedtrenden: algoritmens kjøretid vokser lineært med n. Fordeler med Big O • Lar oss sammenligne algoritmer uavhengig av maskin og implementasjonsdetaljer. • Viser hvordan algoritmen skalerer med store datasett. • F.eks.: ○ Lineært søk: O(n) ○ Binærsøk i sortert array: O(log n)

Foretrekke algoritmer med lavere kjøretidskompleksitet • Når vi skal velge algoritmer for et problem, vil vi som regel foretrekke algoritmer som vokser langsommere når input-størrelsen n øker. • Lavere kompleksitet betyr at algoritmen blir mer skalerbar og kan håndtere større datasett effektivt. • Selv om en algoritme kan være “rask” på små datasett, kan forskjellen bli enorm på store datasett.

Vanlige uttrykk for kjøretidskompleksitet Notasjon Betegnelse Forklaring O(1) Konstant tid Antall steg er uavhengig av input-størrelse O(log n) Logaritmisk tid Antall steg vokser logaritmisk med n (f.eks. binærsøk) O(n) Lineær tid Antall steg vokser proporsjonalt med n (f.eks. lineært søk) O(n·log n) Lineæritmisk tid Vanlig for effektive sorteringsalgoritmer som mergesort $O(n^2)$ Kvadratisk tid Antall steg vokser med kvadratet av n (f.eks. boblesortering) O(n^k) Polynomiell tid Generell polynomvekst, ofte

ineffektivt for store n når k>2 O(2^n) Eksponensiell tid Dobler antall steg for hver økning av n med 1, ekstremt ineffektivt Intuisjon: Jo “flattere” kurven, desto bedre skalerer algoritmen.

Kodeeksempler med Big O

1. Konstant tid – O(1) 1 Procedure Constant(n) 2 return n * 3 • Utfører én operasjon, uansett hvor stor n er.
2. Logaritmisk tid – O(log n) 1 Procedure Log(n) 2 i ← n 3 while i > 0 do 4 Constant(i) 5 i ← ⌊i / 2⌋ • i halveres hver iterasjon → antall iterasjoner ~ $\log_2(n)$. • Eksempel: binærsøk.
3. Lineær tid – O(n) 1 Procedure Linear(n) 2 for i ← 0 to n-1 do 3 Constant(i) • Går gjennom alle n elementer én gang. • Eksempel: lineært søk i array.
4. Lineæritmisk tid – O(n·log n) 1 Procedure Linearithmic(n) 2 for i ← 0 to n-1 do 3 Log(n) • Kombinerer lineær og logaritmisk kompleksitet. • Typisk for effektive sorteringsalgoritmer som mergesort og heapsort.
5. Polynomiell tid – O(n^k) 1 Procedure Polynomial(n) 2 for i1 ← 0 to n-1 do 3 for i2 ← 0 to n-1 do ... 5 for ik ← 0 to n-1 do 6 Constant(i) • Generell struktur med k nested løkker → kompleksitet O(n^k). • Kan bli svært ineffektiv hvis k er stor og n vokser.
6. Kvadratisk tid – O(n^2) 1 Procedure Quadratic(n) 2 for i ← 0 to n-1 do 3 for j ← 0 to n-1 do 4 Constant(i) • Spesialtilfelle av polynom: k=2. • Typisk for enkle “nested loop”-algoritmer som boblesortering.
7. Eksponensiell tid – O(2^n) 1 Procedure Exponential(n) 2 if n = 0 then 3 return 1 4 a ← Exponential(n-1) 5 b ← Exponential(n-1) 6 return a + b • Antall kall dobles for hver økning av n → svært raskt voksende. • Eksempel: brute-force algoritmer for kombinatoriske problemer (f.eks. power set, Travelling Salesman).