**Ressurseffektive programmer**

**Introduksjon**

* ***Kjøretid (execution time)****: Tiden det tar for programmet vårt å fullføre kjøringen.*
* ***Minne (memory)****: Den største mengden dataminne programmet bruker på noe tidspunkt.*
* ***Kjørbar filstørrelse (executable size)****: Diskplassen som kreves for å lagre programmets kompilerte kjørbare fil.*
* ***Diskplass****: Plassen som brukes av dataene programmet lagrer.*
* ***Nettverksbåndbredde****: Båndbredden som brukes når programmet overfører data til eller fra andre datamaskiner via et lokalt nettverk eller internett.*
* ***Elektrisk strøm****: Strømforbruket til datamaskinen når den kjører programmet vårt.*
* ***Utviklertid (developer time)****: En menneskelig ressurs i motsetning til en maskinressurs, og ofte den viktigste ressursen.*

Optimalisering innebærer å omstrukturere programmer for å redusere ressursbruk (f.eks. tid, minne, strøm) samtidig som korrekt beregning opprettholdes. Å oppnå optimalisering krever ofte avveininger, som å bruke mer minne for å redusere kjøretid eller investere mer utviklertid for å lage ressursøkonomiske løsninger. Korrekthet er kritisk, da et program må produsere nøyaktige resultater til tross for redusert ressursbruk.

I praksis er det umulig å optimalisere for alle faktorer samtidig, så prioriteringene avhenger av situasjonen. Utviklertid er spesielt viktig, da det er en menneskelig ressurs, i motsetning til maskinressurser som ofte er rikelige. Dermed er det ofte mer kritisk å optimalisere utviklertid enn maskinressurser, med mindre man arbeider med svært beregningstunge oppgaver.

Programmeringsspråk kan kategoriseres basert på hvordan de utføres:

1. **Interpreted languages**: En interpreter leser og utfører kildekoden direkte. Programmene krever at interpreter er installert på maskinen, og hver kombinasjon av operativsystem og prosessor trenger sin egen interpreter. Eksempler: Python, Lisp og JavaScript (tolkes av nettlesere).

A diagram of a process

Description automatically generated

1. **Kompilerte språk**: Bruker en kompilator til å omgjøre kildekode til maskinkode som prosessoren kan forstå. Resultatet er en binærfil eller kjørbar fil som kan kjøres uten andre programmer. Endres kildekoden, må den kompileres på nytt. Hver kombinasjon av operativsystem og prosessor krever en spesifikk kompilator. Eksempler: C, C++ og Rust.

A diagram of a process

Description automatically generated

1. **VM-språk**: Kombinerer egenskaper fra tolknings- og kompilasjonsspråk. Kildekoden kompileres til bytekode, som kjøres av en virtuell maskin (VM) som oversetter det til maskinkode under kjøring. Ulike operativsystemer og prosessorarkitekturer krever spesifikke kompilatorer og VM-er. Flere språk kan dele samme VM, for eksempel Java og Kotlin med Java Virtual Machine, eller C# og F# med .Net Runtime. Eksempler inkluderer Java, Kotlin, C# og F#.

A diagram of a computer hardware process

Description automatically generated

Programmeringsspråk kan teoretisk støtte flere kjøremetoder (interpreted, kompilert eller VM-basert), men å lage interpreter, kompilatorer eller VM-er er komplisert, så de fleste holder seg til én metode. Programmer i interpreted languages er tregere fordi parsing og analyse gjøres i sanntid. VM-språk reduserer dette med en separat kompilasjonsfase, men bruker fortsatt ekstra ressurser under kjøring. Kompilerte språk er vanligvis raskest, siden kjørbare filer går direkte på prosessoren, og noen (f.eks. C/C++) hopper over sikkerhetssjekker for bedre ytelse. Avanserte VM-er (f.eks. Java og C#) kan optimalisere programmer under kjøring og noen ganger overgå kompilerte språk i hastighet.

Her er noen optimaliseringer som kompilatoren kan utføre. Detaljert implementasjon dekkes i kompilatorkurs, men det er nyttig å vite om dem, fordi vi kan fokusere på mer lesbar kode i stedet for å optimalisere selv:

* **Constant folding**: Matematiske uttrykk med kun konstanter beregnes under kompilering og erstattes med resultatet (f.eks. 24 \* 60 \* 60 blir 86400).
* Strength reduction: Omskriving av matematiske uttrykk til raskere ekvivalenter (f.eks. 2 \* a til a + a).
* **Loop-unrolling**: Reduserer antall iterasjoner ved å kopiere løkkekoden flere ganger for å spare tid på sjekk av løkkebetingelser.
* **Common subexpression elimination**: Uttrykk som brukes flere ganger, beregnes én gang og gjenbrukes.
* **Inlining**: Kopierer små funksjoner direkte inn i kallene for å unngå kostnaden ved funksjonskall, til fordel for større filstørrelse.

See forelesningsnotater for kode og eksempler.

**Algoritmer & Datastrukturer**

**Algoritmer og problemer**

En algoritme er en detaljert, entydig beskrivelse av hvordan man løser et problem. Et problem beskriver hva inndataene ser ut som og hva ønsket utdata skal være. For eksempel kan sorteringsproblemet beskrives som: "gitt en liste med objekter der hvert par kan sammenlignes, omorganiser dem slik at hvert objekt i listen er større enn eller lik de foregående objektene." En spesifikk inndata for problemet, som en liste med fem elementer, kalles en probleminstans.

En algoritme må være strukturert slik at man på forhånd kan forutsi hvor lang tid det vil ta å løse et problem. Varigheten av en algoritme uttrykkes i abstrakte "operasjoner", der hver operasjon kan utføres på en konstant tid av en datamaskin.

Eksempler på primitive operasjoner:

* **Grunnleggende aritmetikk**: adding, subtrahering, multiplikasjon, divisjon eller sammenligning av to tall med fast størrelse. Dette dekker også operasjoner med enkelttegn på strenger (subtrahere eller sammenligne to tegn, eller legge til et heltall til et tegn). Fast størrelse betyr at hvert tall tar opp et forhåndsbestemt antall byte, vanligvis fire (32 biter) eller åtte (64 biter). Merk at heltall i Python ikke har fast størrelse, noe som betyr at verdiene kan være svært store, og operasjonene på dem kan potensielt ta lang tid — men hvis vi er sikre på at heltallene vi jobber med er begrenset i størrelse, kan en aritmetisk operasjon i Python fortsatt betraktes som en primitiv operasjon.
* **Lesing eller skriving av en variabel**: Et element i en liste/array eller en medlems-egenskap av et objekt.
* **Hopp til koden i if- eller else-grenen** (selv om selve betingelsen kanskje ikke er primitiv).
* **Hopp til toppen av en for-løkke eller while-løkke fra bunnen, og hopp ut av løkken når den er ferdig** (selv om beregningen av om løkken er ferdig kanskje ikke er primitiv).
* **Funksjonskall**: Så lenge alle parameterne er referanser eller verdier med fast størrelse. Returnere en referanse eller verdi med fast størrelse fra en funksjon.

Eksempler på ikke-primitive operasjoner:

* **Beregning av kvadratroten**
* **Legge til et element i en liste**
* **Sammenligne eller sammenkoble strenger**

For enhver algoritme kan vi finne en matematisk funksjon som beskriver hvor lang tid den vil bruke, basert på størrelsen på probleminstansen. Denne funksjonen kalles algoritmens kjøretidskompleksitet. Den beste investeringen i effektiviteten til et program er å velge gode algoritmer for de viktigste delene av programmet.

De fleste programmene vi skriver består av våre egne algoritmer, og derfor er det viktig å kontinuerlig analysere ytelsen til våre egne algoritmer. Hvis du trenger en spesifikk, velkjent algoritme, er det generelt ikke effektivt å implementere den selv, da mange algoritmer allerede finnes som biblioteker og er bedre testet.

**Asymptotisk notasjon**

Asymptotisk notasjon brukes til å analysere matematiske funksjoner, spesielt for å beskrive tidsbruken til algoritmer. Hensikten er å sammenligne veksten til funksjoner som beskriver kjøretiden til algoritmer, og forenkle dem ved å fokusere på de viktigste delene av funksjonene. Dette er viktig når vi jobber med store problemer, der veksten til algoritmer med store innganger er det mest relevante.

De tre hovedtypene asymptotisk notasjon er:

1. **Big-O (O)**: Beskriver øvre grense for en funksjon, som betyr at funksjonen ikke vokser raskere enn en annen funksjon.
2. **Big-Omega (Ω)**: Beskriver nedre grense, som betyr at funksjonen ikke vokser langsommere enn en annen funksjon.
3. **Big-Theta (Θ)**: Beskriver funksjoner som vokser med samme hastighet.

Notasjonen er et verktøy for å analysere algoritmer og deres kjøretid, uavhengig av spesifikke maskinvare- og programvarefaktorer. Funksjonene beskrives ved hjelp av asymptotisk notasjon, og det er vanlig å bruke symbolene "O", "Ω" og "Θ" for å representere sett med funksjoner som vokser på samme måte.

A table with black text

Description automatically generated

Ingen av de generelle sorteringsalgoritmene (de uten spesifikke forutsetninger om elementene, som kan sammenlignes parvis) har et dårligere worst-case enn 𝑛 lg 𝑛. Det er umulig å oppnå et bedre worst-case enn dette. Vi deler sorteringsalgoritmene inn i tre grupper basert på deres kjøretid i verste fall:

1. **Kvadratisk tid (𝑛²)** – Algoritmer som innsats-sorting, seleksjonssortering og boblesortering, har to nestede løkker og en kjøretid på Θ(𝑛²).
2. **n log n-tid** – Algoritmer som quicksort, mergesort og heapsort tenker på problemet som et binært tre, og deres kjøretid er Θ(𝑛 log 𝑛), hvor dybden på treet er lg 𝑛. Quicksort kan få et dårlig worst-case med en ujevnt balansert trestruktur, men ved å stokke dataene på forhånd kan dette unngås.
3. **Lineær tid (𝑛)** – Algoritmer som bucket sort, counting sort og radix sort, forutsetter spesifikke egenskaper om dataene, og har enten én eller få løkker.

Når vi analyserer vår egen kode, kan vi bruke disse prinsippene for å vurdere kjøretiden basert på antall nestede løkker og operasjoner per iterasjon. Hvis datastrukturen er et balansert tre med 𝑛 noder, vil kjøretiden for å traversere treet være Θ(log 𝑛), og hvis algoritmen jobber på hver node i treet, vil kjøretiden være Θ(𝑛 log 𝑛).

Viktig mattematisk formel for kompleksitet:

**n + (n-1) + (n-2) + (n-3) +... + 1 + 0** 🡪 n \* (n+1) / 2 🡪 0.5n^2 + 0.5n, forenklet blir bare **n^2**

**Skjult kompleksitet; gjentatt strengsammenslåing**

Faktoren 𝑝 i formelen (som representerer operasjonene som utføres i den innerste loopen) kan være lur, spesielt når man bruker bibliotekfunksjoner, fordi én enkelt linje med kode kan ta mer enn en konstant mengde tid. Hvis du kaller en funksjon, er det viktig å se på kildekoden for å forstå dens kjøretid kompleksitet (eller lese bibliotekdokumentasjonen hvis kildekoden ikke er offentlig, og dersom dokumentasjonen ikke oppgir kjøretiden, kan man gjøre et kvalifisert gjetning basert på hva funksjonen gjør).

Selv innebygde funksjoner i språk kan ta lengre tid enn man skulle tro. Spesielt gjelder dette **strengsammenslåing** (å lime sammen to strenger), som er en lineær operasjon i språk der strenger er **immutable** (som Python og Java). I slike språk kan ikke strenger endres etter opprettelsen, så sammenslåing må utføres ved å lage en ny streng og kopiere innholdet fra de to originale strengene til den nye, noe som tar tid proporsjonal med summen av lengdene på de to originale strengene.

Gjentatt strengsammenslåing kan derfor føre til stor ineffektivitet. Hvis du har 𝑛 strenger med lengde 𝑘 hver, vil den første sammenslåingen ta 2𝑘 operasjoner, den andre sammenslåingen vil ta 3𝑘, den tredje vil ta 4𝑘, og den siste vil ta (𝑛 − 1)𝑘. Dette kalles **"painters algorithm"**, og summen av alle disse operasjonene blir Θ(𝑘𝑛²). I det ekstreme tilfellet hvor hver streng er én enkelt karakter, blir dette Θ(𝑛²), som er kvadratet av den totale inngangsstørrelsen.

**Amortisert kjøretid kompleksitet**

I forhold til strengsammenslåing, kan spørsmålet oppstå om det er bedre å legge til elementer i en liste. Det virker som om en liste også ville trenge en form for ekspansjonsalgoritme, som "painter’s algorithm". Svaret er at de innebygde listetypene i de fleste programmeringsspråk er **mutable**, og de håndterer minne på en måte som gjør at de kan tilby rask tilgang til elementene.

For at en liste skal kunne tilby konstant tid for tilfeldig tilgang (det vil si at lese/skrive et element ved indeks i tar samme tid uavhengig av i), må elementene være lagret konsekutivt i minnet. Når listen vokser og flere elementer legges til, er det vanligvis ikke nok plass rett ved siden av listen i minnet, og derfor må listen ekspanderes.

Løsningen er å **sette av ekstra minne** for å oppnå raskere operasjoner. Når listen når sitt maksimale kapasitet, dobles størrelsen på den underliggende tabellen (arrayen), og de gamle elementene kopieres over til den nye, større listen. Dette betyr at selv om enkelte operasjoner krever mye tid for å kopiere data, vil de fleste operasjoner være raske.

En viktig detalj er at antallet kopieringer følger et mønster: For hver ekspansjon av listen, kreves flere operasjoner for å kopiere eksisterende elementer. For eksempel, ved å legge til 𝑛 = 2𝑝 elementer, vil vi ha kopiert 20 + 21 + 22 + ... + 2𝑝−1 elementer, som gir en total kjøretid på 2𝑛 - 1 = Θ(𝑛).

Dette betyr at den **amortiserte kjøretiden** for å legge til ett enkelt element er Θ(1), selv om noen individuelle operasjoner kan være mye tregere. Når kjøretiden for en sekvens av operasjoner vurderes, der individuelle operasjoner kan ta forskjellig tid, refereres det til som **amortisert kompleksitet**.

I de fleste tilfeller er det greit at enkelte operasjoner tar lenger tid, så lenge total kjøretid er akseptabel. Imidlertid, i **harde sanntidsprogrammer** (som i maskinvarekontrollprogramvare, som biler eller medisinske instrumenter), kan denne variasjonen i kjøretid være uakseptabel. For slike programmer må man enten sette begrensninger på datamengden og forhåndsallokere minne for hver liste, eller bruke andre teknikker som **lenkede lister**, som kan vokse dynamisk med konstant forventet tid for hver append, men kan være risikabelt da minneallokering noen ganger kan være treg.

**Andre kjente algoritmer og datastrukturer**

Her er noen kjente algoritmer og datastrukturer som er nyttige i forskjellige problemstillinger:

1. **Balanserte binære søketre**:
   * Det vil bli forklart mer om dette i en kommende versjon av notatene.
2. \**Korteste vei fra en kilde (Bellman-Ford, Dijkstra, A*)\*\*:
   * For en vektet, rettet graf, er det ofte nødvendig å finne den korteste eller billigste veien fra en startnode til en målnode.
   * **Dijkstras algoritme** finner korteste vei på tidene Θ(𝐸 lg 𝑉) eller Θ(𝑉²), men kan ikke håndtere negative kantvekter.
   * **Bellman-Ford-algoritmen** kan håndtere negative vekter og finner korteste vei på Θ(𝐸𝑉) tid.
   * *A-algoritmen*\* er en forbedring av Dijkstra og er nyttig når vi bare er interessert i én målnode og kan gjøre informerte gjetninger om hvilke kanter som leder til målet.
3. **Minimum spanning trees (Prim, Kruskal)**:
   * Et minimum spanning tree (MST) forbinder alle noder med færrest kanter og lavest total vekt.
   * **Prim’s algoritme** ligner på Dijkstra, og vokser treet ved å velge den korteste kanten som ikke danner en syklus.
   * **Kruskal’s algoritme** sorterer alle kantene og legger dem til så lenge de ikke danner en syklus.
4. **Maksimalt flyt og bipartitt matching (Ford-Fulkerson, Edmonds-Karp)**:
   * Det vil bli forklart mer om dette i en kommende versjon av notatene.
5. **Prefiksfrie koder (Huffman-tre)**:
   * Huffman-treet lager prefiksfrie koder for tekst, som kan brukes til komprimering. Det garanterer at ingen kode er et prefiks til en annen, noe som gjør dekoding mulig.

**Grådige algoritmer**

Grådige algoritmer tar gjentatte ganger den beste løsningen tilgjengelig i øyeblikket. Eksempel: Ved å gi penger i mynter og sedler kan man bruke den største valutaen som er mindre enn eller lik beløpet.

Selv om grådige algoritmer er enkle å implementere, gir de ikke alltid den optimale løsningen. For eksempel, i tilfeller hvor mynter er 1, 4 og 5, vil en grådig algoritme gi en suboptimal løsning. Eksempler på grådige algoritmer som er optimale er **Dijkstra** og **Huffman-tre-algoritmen**, men Dijkstra bruker en prioritetskø for å optimalisere valget.

**Hovedpoeng:** Det finnes mange kjente algoritmer og datastrukturer som løser forskjellige problemer effektivt, men noen ganger kan grådige algoritmer være suboptimale hvis de ikke er utformet riktig for problemet.