5: Sortering

Splitting av videoer

1. Om sortering. Hvorfor er sortering så viktig. Hva kan sorteres? Python sammenlikningsmetoder. Bok eksempel som kan sorteres.
2. Klassifisering av sorteringsalgoritmer
3. Innsettingssortering (Insertion Sort)
4. Shell Sort
5. Flettesortering (Merge Sort)
    a. Timsort (MergeSort variant, brukt av Python sin sort metode)
6. QuickSort: Presentasjon av algoritmen
7. QuickSort: Analyse og optimalisering
8. QuickSelect – variant av QuickSort som løser Selection problemet
9. Hvor gode kan sorteringsalgoritmer bli
10. Sortering i lineær tid: Tellesortering (Counting Sort)
11. Ytelsen til ulike programmeringsspråk – tas under Sortering siden jeg her har algoritmer og testrigger som kan brukes for å måle dette.

Introduksjon

Man har ofte et datasett som man ønsker sortert etter ulike kriterier. I ei liste med studenter ønsker man en gang å ha sortering etter navn, en annen gang etter studentnummer og en tredje gang etter studieprogram. Andre eksempler:

- I en telefonkatalog er det lett å finne nummeret til en person gitt navnet, men vanskelig å finne navnet til en person gitt nummeret.
- En del operasjoner blir mye enklere hvis lista er sortert!
    - Binærsøk vs. Sekvensielt søk (merk imidlertid at sortering i seg selv koster mer enn et sekvensielt søk)
    - RemoveAll på lister går fra O(m*n) til O(m + n). Vis algoritmen på ei sortert liste
    - RetainAll på lister blir tilsvarende RemoveAll
    - ContainsAll på lister blir også tilsvarende RemoveAll
- Man ønsker ofte data sortert i en eller annen orden, for eksempel alfabetisk
- Algoritme for å finne duplikater i ei liste. $O(n^2)$ for usortert liste, O(n) for sortert liste.

Derfor er sortering et mye studert problem i algoritme-faget.

Hva kan sorteres?

- Tall
- Strenger kan sorteres alfabetisk
- Objekter av klasser som implementerer «<» (mindre enn) operatoren og hvor den definerer en ordning på objektene (a < b = True medfører b < a = False, a < b og b < c medfører a < c, for et hvert par av objekter a og b så er enten a<b, a=b eller a>b).
- Objekter som kan sammenliknes av en eksplisitt Comparator (sammenlikner). Slike sammenliknere bruker man for å kunne sortere objekter av samme klasse etter ulike kriterier.
    - Standard Python komparator: Python sin innebygde list.sort() metode tar et key-argument, som skal være en funksjon med ett argument og som konstruerer en

sorteringsnøkkel for argumentet. Denne sorteringsnøkkelen må implementere «<» operatoren som nevnt over.

Python sammenlikningsmetoder

Ved å implementere disse spesialmetodene kan man bruke sammenlikningsoperatorene på objekter av egendefinerte klasser.

__eq__: == operatoren

__lt__: < operatoren

__gt__: > operatoren

__le__: <= operatoren

__ge__: >= operatoren

__ne__: != operatoren

Evaluering av sorteringsalgoritmer

Sorteringsalgoritmene vil bli evaluert på følgende kriterier:

- Kjøretid
- Minnebruk
- Stabilitet: Hvis du har flere objekter som har samme nøkkel (sammenlikningsattributt), men ellers er ulike, så vil en stabil sorteringsalgoritme ikke påvirke rekkefølgen på disse objektene. En ustabil sorteringsalgoritme kan godt påvirke rekkefølgen her. Eksempel: Person-klassen. Personer sammenliknes på navn, men har også andre egenskaper. Navn er derfor nøkkel til person.
- In-place sortering: En in-place sorteringsalgoritme modifiserer arrayen den sorterer underveis og trenger derfor ikke ekstra plass. Andre algoritmer sorterer inn i en ny array og trenger derfor ekstra plass underveis.
- Om sorteringsalgoritmen bruker sammenlikninger eller ikke. Algoritmer som bruker sammenlikninger virker på alle objekter som er sammenliknbare. Andre algoritmer virker bare på spesielle datatyper. Eksempel counting sort for heltall på slutten av dette temaet

Merk: Alle sorteringsalgoritmene vil bli vist og analysert basert på at de kjører på standard Python lister (array-lister). Sortering av lenkete lister er en frivillig tilleggsoppgave til øving 4 og vil bli gjennomgått i forbindelse med at jeg går gjennom øving 4.

Eksempler

- Tilfeldige lister av heltall
- Bøker med __lt__ metode basert på tittel
- Kunne hatt større bokliste med sortering på utgivelsesår for å demonstrere stabile vs. ustabile sorteringsalgoritmer

Enkel algoritme: Insertion Sort (Innsettingssortering)

Dette er måten man ofte sortere hånda si i kortspill.

Algoritme:

- Start med å anta at det første elementet er sortert
- Sjekk det andre elementet mot det første og sett det inn før eller etter det første
- Sjekk det n-te elementet mot alle (n-1) elementene og sett det inn på riktig sted, flytt samtidig alle elementene som er høyere enn det n-te elementet ett hakk opp i lista.

- Skriv algoritmen
- Vis demonstrator
- Kjøretid $O(n^2)$ worst og average case. For hvert element må du flytte det i gjennomsnitt n/2 plasser bakover.
- Kjøretid O(n) best case: Lista er allerede sortert.

Fordeler

- Kort kode
- Bra lokalitet (Cache-systemet kan forutsi hvor den kommer til å gå, du får derfor god cache utnyttelse) Bruk norsk «hurtiglager» for cache?
- Rask på små datasett (har lavere konstantledd enn de fleste andre algoritmene)
- Rask på nesten sortert liste
- Stabil
- In-place

Ulemper

- Treg på store datasett

Vis algoritme for å sjekke at ei liste er sortert

- For hvert element unntatt det første i lista, sammenlikn elementet med det forrige elementet
    - Hvis det forrige elementet er større er ikke lista sortert, returner False
- Hvis den ikke returnerer False for noen av sammenlikningene er lista sortert, returner True.
- Kjøretid O(n).

Litt mer avansert: Shellsort

- Problemet med insertion sort er at den bare sammenlikner elementer som er ved siden av hverandre
- Kan gjøre det bedre ved å sammenlikne elementer som er langt unna
- Kan gjøre en insertion sort liknende algoritme hvor man bare sammenlikner elementer som er avstand k fra hverandre
- Hvis man gjør dette for en stor k, og deretter for en mindre k, er den fortsatt sortert på den store k etterpå (beviset for dette er komplisert så jeg tar det ikke)
- Shell's algoritme: Gjør en n/2 sort. Gjør deretter sorteringer på $k_n = k_{n-1}/2$ helt til du ender opp med å gjøre en insertion sort på en forhåpentligvis nesten sortert liste når k = 1.
- Kjøretid: $O(n^2)$ worst case (array hvor annethvert element er lite og annethvert element er stort), $O(n^{3/2})$ average case.
- Forbedring 1: hvis $k_n$er et partall, legg til 1. Hindrer worst case, gjør algoritmen $O(n^{3/2})$ worst og average case
- Forbedring 2: Del på 2,2 i stedet for på 2 og rund av til nærmeste heltall. Denne virker og har best kjøretid, men ingen har enda klart å bevise hvorfor!

- Shellsort er et eksempel på en algoritme som er basert på en enkel idé, men hvor det å analysere den er svært vanskelig.

Fordeler

- Kort kode, men ikke like kort som insertion sort
- Rask på datasett av middels størrelse (vis dette med konkrete eksempel – i min demonstrator fra 2016 så var shellsort raskere enn quicksort på et datasett med 40000 heltall. Du måtte opp i over 100000 heltall før mergesort og quicksort ble raskere.
- In-place

Ulemper

- Tregere enn de beste algoritmene på store datasett
- Bevisene for kjøretid er enten svært vanskelige eller ikke-eksisterende.
- Ustabil

Enkel splitt-og-hersk: Flettesortering (Merge Sort)

Prøver Splitt og Hersk for å få en bedre algoritme enn ShellSort

- Bruker teknikken «splitt og hersk» for å lage en bedre algoritme
- Rekursjon hvor hvert ledd halverer størrelsen på problemet
- Kan kombinere løsninger på delproblemene i lineær tid.
- Algoritme for å slå sammen to sorterte lister til en:
    - Lag en referanse r1 til liste1 og en referanse r2 til liste2. Dette er begge indekser inn i den respektive lista, og starter begge på 0
    - Fortsett til en av indeksene når slutten av lista si
        - Sammenlikn liste1[r1] med liste2[r2].
            - Er liste1[r1] lavere eller lik, sett den inn i resultatet med append og øk r1 med 1
            - Ellers sett liste2[r2] inn i resultatet og øk r2 med 1.
    - Sett inn de siste elementene i den lista som ikke er ferdig satt inn enda i resultatet.
    - Analyse: while-loop som går gjennom begge listene en gang, kjøretid O(m + n).
- Algoritme for selve flettesorteringa
    - Er lista på 1 element er den allerede sortert. Returner lista uforandret. (basetilfelle)
    - Er lista på 2 elementer, bytt dem om de er i feil rekkefølge. Returner deretter lista. (basetilfelle)
    - Splitt lista i to halvparter, r1 og r2
    - Kall flettesortering rekursivt på r1 og r2
    - Bruk algoritmen for å slå sammen to sorterte lister på de nå sorterte delene r1 og r2.
- Analyser kjøretida til hele algoritmen
    - Vis at den kjører i Theta(n*log(n)) tid
    - Å sortere n grupper med 2-3 elementer i hver tar O(n) tid.
    - Å slå sammen to sorterte lister tar O(n) tid
    - Denne sammenslåingen må gjøres én gang for hvert rekursjonsnivå
    - Hvert rekursjonsnivå halverer størrelsen på listene
    - Det er derfor log(n) rekursjonsnivå
    - Ergo total kjøretid $\Theta(n*\log(n))$
    - Rekurrenslikning: $T(n) = 2*T(n/2) + n$, $T(1) = 1$.
    - Denne kan løses med master theorem. Den gir a=2, b=2 og k=1, b^k=2, a=b og derfor tilfelle 2, som gir O(n*log(n))

- Stort konstantledd i kjøretida da den kopierer arrayen for hver rekursjon
- Minnebruk: Implementasjonen fra timene har Θ(n*log(n)) minnebruk. En mer kompleks versjon som gjenbruker lister kan lages som har Θ(n) minnebruk. Flettesortering kan ikke implementeres med mindre minnebruk enn det.

Fordeler med flettesortering

- Har garantert Θ(n*log(n)) kjøretid, derfor alltid brukbart rask selv på store datasett og selv på det merkeligste datasett
- Enkel å parallellisere (send hver halvpart til sin egen maskin / prosessorkjerne for sortering)
- Stabil
- For delvis sorterte lister: Kan legge inn en «er denne delen allerede sortert?» test før man splitter lista og slik stoppe rekursjonen tidlig. Denne testen er O(n) og allerede vist. Siden fletting er O(n) allerede så øker ikke dette kjøretida i O-notasjon, men øker konstantleddet noe.

Ulemper

- Stor tidskonstant på grunn av kopiering av arrayen
- Trenger mer plass enn de andre på grunn av kopiering av arrayen
- Ikke in-place

Algoritmeteknikk: Skift algoritme når problemstørrelsen blir liten

Flettesortering har en bedre kjøretid i O-notasjon enn innsettingssortering, men har et høyere konstantledd. Flettesortering egner seg derfor for store datamengder, mens innsettingssortering egner seg for små. En måte å gjøre flettesortering raskere på er derfor å skifte til innsettingssortering når tabellene blir små nok. Erstatt basetilfellet i algoritmen med følgende:

- Hvis lengden på lista er mindre enn en terskelverdi, for eksempel 20, gjør innsettingssortering på den og returner den. Dette blir det nye basetilfellet.

Python sin standard sortering: Timsort, en variant av flettesortering

- Brukes av Python sin innebygde sort() metode for lister.
- Variant av flettesortering
- Går gjennom lista og ser etter del-lister som allerede er sortert enten i riktig eller motsatt rekkefølge
    - Et hvert par av elementer er sortert i enten riktig eller feil rekkefølge, så slike del-lister («runs») har minst to elementer.
- Har en terskel for disse del-listene
    - Mindre enn terskel: Bruker innsettingssortering på en serie nabo del-lister så lenge de totalt sett er kortere enn terskelverdien.
    - Høyere enn terskel: Bruker slaa_sammen_sorterte_lister fra flettesortering på to nabo-lister
- Slår sammen to og to del-lister i et hierarki flettesortering stil.
- Kjøretid Θ(n*log(n)) worst og average case, Θ(n) best case – en allerede sortert liste.
- Minnebruk O(n) worst case, O(1) best case
- Stabil
- Ikke in-place, men prøver å minimere mengden ekstra mine den allokerer gjennom å bare flette det som er nødvendig
- Bruker sammenlikninger

Mer komplisert splitt-og-hersk: Quicksort, Ofte den raskeste algoritmen

- Flettesortering sitt problem: kopiering av array
- Ønsker å sortere «in place» for å redusere kjøretida og minnebruken
- Bruker splitt og hersk teknikken, men på en annen måte enn flettesortering
- Heter quicksort siden den i gjennomsnitt og for et typisk datasett er den raskeste sorteringsalgoritmen
- Velger et «pivot element» og splitter lista basert på dette. Splitting av lista er O(n).
- Algoritme for in-place splitting
    - Gå gjennom lista fra starten og slutten i parallell. La S være startpeker og E være sluttpeker
    - Hvis begge er større enn pivot, flytt E bakover
    - Hvis begge er mindre enn pivot, flytt S framover
    - Hvis S er større enn pivot og E mindre, bytt verdiene de refererer til og flytt begge
    - Når de møtes, er lista splittet. Sett in pivot element der de møtes. Pivot element er nå på riktig plass i lista. Lagre de to pekerne og sorter hver halvpart for seg på samme måte.
- QuickSort algoritmen:
    - Er lista på 0 eller 1 element, er den sortert. Returner (Basetilfelle)
    - Er lista på 2 elementer, sjekk om det er i riktig eller feil rekkefølge. Bytt om på dem om de er i feil rekkefølge. Deretter returner (basetilfelle)
    - Splitt lista i to baser på pivot element
    - Sorter venstre del rekursivt med QuickSort
    - Sorter høyre del rekursivt med QuickSort
- Kjører i O(n*log(n)) tid best case
    - Beste splitt gir to like store lister, hver på halvparten av størrelsen.
    - Dette gir samme rekurrenslikning som flettesortering, og derfor samme kjøretid i O-notasjon
- Kjører i $O(n^2)$ i verste tilfelle.
    - Verste tilfelle splitter ut 1 element og (n-1) elementer hver gang
    - Rekurrenslikning T(n) = T(n-1) + n
    - Gjentatt substitusjon på denne gir T(n) = n + (n-1) + (n-2) + … + 1. Denne rekka er vist tidligere i faget og er $O(n^2)$
    - Derfor $O(n^2)$ kjøretid
- Kjører i O(n*log(n)) tid gjennomsnitt. Denne algoritmen er et unntak for regelen om at gjennomsnittlig tilfelle ofte likner verste tilfelle. For denne algoritmen likner gjennomsnittlig tilfelle beste tilfelle. Flere varianter av beviset:
    - Gjennomsnittsbetraktninger. Rekurrenslikningen T(n) = n + summen av sannsynligheter for hvor pivot elementet er.
    - Beste tilfelle halvparten, verste tilfelle andre halvpart gir O(n*log(n))
    - En splitting i 1/10n + 9/10n (som er ganske dårlig men ikke worst case) gir fortsatt O(n*log(n))
- Kan redusere sjansen for verste tilfelle med å velge pivot riktig
    - Naivt pivot valg (alltid velg første element) gir verste tilfelle for en array som allerede er sortert!
    - Sjekk første, midterste og siste element og velg det midterste av de tre som pivot.

    - Randomisert: Velg tilfeldig element. Randomisering er en teknikk for å hindre verste tilfelle i en del algoritmer.
- Andre teknikker for å fjerne worst case
    - Får lett worst-case ytelse for liste med mange duplikater.
    - For dette tilfellet må man lage en versjon som trigger et bytte hvis begge verdiene (start-peker verdien og slutt-peker verdien) er lik nøkkelen.

Gjennomsnittsbetraktninger bevis

Her er gjennomsnittsbetraktninger beviset for at QuickSort kjører i O(n*log(n)) tid i gjennomsnittlig tilfelle:

Rekurrens T(n) = T(høyre) + T(venstre) + n

Kan si at T(høyre) og T(venstre) begge er et gjennomsnitt av kjøretida for alle mulighetene, så

T(høyre) = T(venstre) = (T(0) + T(1) + T(2) + … + T(n-1))/n

Setter dette inn i rekurrensen:

T(n) = 2*(T(0) + T(1) + T(2) + … + T(n-1))/n + n

Kan gange med n på begge sider

$n*T(n) = 2*(T(0) + T(1) + T(2) + … + T(n-1)) + n^2$

Kan skrive likningen for T(n-1):

$(n-1)*T(n-1) = 2*(T(0) + T(1) + T(2) + … + T(n-2)) + (n-1)^2$

Trekk den siste fra den nest siste:

$n*T(n) - (n-1)*T(n-1) = 2*T(n-1) + n^2 – (n-1)^2$. De andre leddene forsvinner.

$n^2 – (n-1)^2 = n^2 – n^2 + 2n – 1 = 2n-1$

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

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

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

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

$n*T(n) + = (n+1)*T(n-1) + 2*n – 1$

Har nå en likning som gir T(n) som funksjon av bare T(n-1). Bruker telescoping sum igjen men må endre likningen. Deler på n*(n+1) og fjerner det ubetydelige -1 leddet:

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

Skriv denne for n-1, n-2, osv.

Legg saman alle likningene

Fjern like ledd på høyre og venstre side.

T(n)/n+1 = 2*(1 + ½ + 1/3 + … + 1/n+1) – 5/2

Bruker at integralet av 1/n er log(n), så summen er O(log(n)).

T(n)/n+1 er O(log(n))

T(n) er O(n*log(n))

Fordeler med quicksort

- Den raskeste sorteringsalgoritmen for mange store datasett
- Sorterer in-place, som reduserer tidskonstanten og minnebruken

Ulemper

- Komplisert algoritme med mange fallgruver i implementering sammenliknet med de andre (eksempel: naivt pivot-valg)
- Treg worst-case
- Ustabil

Selection problemet og QuickSelect

- Selection problemet: Finn det n-te minste elementet i ei liste.
- Eksempel:
    - Finn det 10-ende minste elementet.
    - Finn medianen (k = n/2)
- Naiv algoritme tar O(n*k) tid:
    - Finn det minste elementet gjennom å gå gjennom lista og lagre denne verdien i forelopig_resultat
    - Gjør k ganger:
        - Gå gjennom lista og finn minste verdi som er høyere enn forelopig_resultat
        - Oppdater forelopig_resultat til den nye verdien
- Quickselect: Basert på quicksort men går bare ned den ene greina av rekursjonen. Tar O(n) tid.
- Analyse av quickselect:
    - Rekurrenslikning T(n) = T(n/2) + n best case. Kan bruke master theorem. A=1, b=2, k=1, b^k=2. Får tilfelle 3, som gir O(n) kjøretid
    - Et verre tilfelle vil være at du beholder 9/10 elementer i hvert splitt. Dette gir rekurrenslikningen T(n) = T(9n/10) + n. Master theorem gir a=1, b=10/9[1], k=1, b^k = 10/9. Får fortsatt tilfelle 3, som fortsatt gir O(n) kjøretid.

Kjøretid for sortering

- Sorteringsalgoritmer som baserer seg på sammenlikninger har Ω(n*log(n)) worst-case kjøretid og kan ikke bli bedre enn dette.
- Bevis:
    - Antall mulige rekkefølger av ei liste 1, 2, … , n er n!
        - Første element kan være hvilket som helst av de n elementene i lista. Andre element kan være hvilket som helst element unntatt det du allerede valgte som første, og så videre. Du får n*(n-1)*(n-2)*…*1 = n! muligheter.
    - Kun 1 av disse er den sorterte lista. Antar at alle objektene er distinkte, en antakelse man kan gjøre siden dette er en worst-case analyse.
    - Å sortere med sammenlikninger er å leite gjennom alle de mulige rekkefølgene helt til man finner den riktige.

[1] b blir 10/9 siden Master Theorem sier T(n/b). For å få n*9/10 så må du dele på det motsatte, altså 10/9.

    - For hver sammenlikning deler man antall rekkefølger i to mengder: de mulige og de ikke lengre mulige. Den ene av de to må inneholde minst halvparten av elementene.
    - For hver sammenlikning kan man altså i verste tilfelle maksimalt halvere antall mulige permutasjoner. Du får en likning $n! <= 2^k$
    - Algoritmen må derfor i verste tilfelle gjøre Ω(log(n!)) sammenlikninger, som er Ω(n*log(n))
- For å bli bedre enn flettesortering og quicksort må man derfor vite mer om elementene enn at de er sammenliknbare.
- Eksempel: Counting sort for heltall

Sortering i lineær tid: Tellesortering (Counting Sort)

Enkel sorteringsalgoritme på lister av heltall hvor man vet verdiområdet (alle verdiene er mellom X og Y, for eksempel alder for mennesker som er fra 0 til rekorden på 122 år): Kan også brukes for objekter hvor det man sorterer etter er et heltall hvor man vet minimum og maksimumsverdi.

- Lag en array av tall med størrelse lik verdiområdet
- Gå gjennom opprinnelig liste. For hvert element i opprinnelig liste, øk verdien i arrayen med 1
- Gå igjennom arrayen en gang til eller lag ny array. Første celle i ny array er 0. Andre celle er lik første celle i ny array + første celle i opprinnelig array. Forsett slik oppover. Tanken er å lage indekser hvor første element med denne tallverdien skal settes inn.
- Lag ny liste med kapasitet lik opprinnelig liste
- Gå gjennom opprinnelig liste. Bruk tallverdien som indeks inn i ny array for å sjekke hvor elementet skal settes inn i ny liste, og øk tallverdien i cellen i ny array med 1.
- Kjøretid O(n + r) hvor r er størrelsen på arrayen som brukes. r for range dvs. antall verdier som arrayen som skal sorteres kan ta.

Fordeler

- Raskere enn quicksort for lister med tall med et begrenset verdiområde. Kjøretid O(n + r) hvor r er størrelsen på verdiområdet.
- Stabil
- Bruker ikke sammenlikninger

Ulemper

- Det som sorteres må være heltall eller ha en nøkkel som er et heltall
- Hvis antall mulige verdier er stort blir både plassbruk og kjøretid stor
- Bruker mer plass enn de andre algoritmene, O(n + r) ekstra plass.
    - Ikke in-place

Merk: Tellesortering er den eneste algoritmen uten sammenlikninger som er pensum i DAT200. Andre algoritmer som Radix Sort fikser noen av problemene til tellesortering, men har egne problemer. For Radix Sort er det noen som argumenterer for at den egentlig er O(n*log(n)).

Ytelse til ulike programmeringsspråk

Test: Flettesortering med tilfeldig liste med 150000 elementer:

Hjemme-PC, python vs Go

Python flettesortering: 1,41 sekunder

Go, rein flettesortering, implementert på int-er: 18 millisekunder (0,018 sekunder)

Merk at denne sammenlikningen er litt urettferdig siden Python implementasjonen er mye mer generell enn Go implementasjonen. Derfor testet jeg to Java implementasjoner for å se hvor stor effekt dette har.

Jobb-PC, Python vs. Java

Python flettesortering: 1,31 sekunder

Java, objekt-basert sortering: 92 millisekunder. Denne er like generell som Python sorteringen.

Java, int-basert sortering: 47 millisekunder. Denne er like spesifikk som Go sorteringen.