23
Sortering I.

Sorteringsproblemet.

Sorteringspragmatik.

Sortering ved fordeling.

Sortering ved udvælgelse.

Sortering ved indsættelse.

Shell-sort.

Sortering ved ombytning: boblesortering

PS1 -- Sortering I

Noter

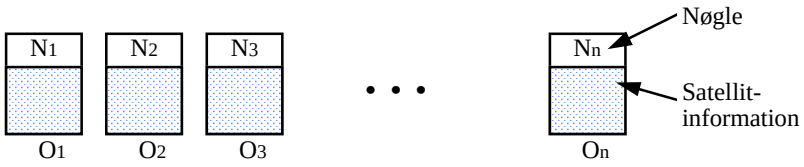
I denne forelæsning starter vi med at præsentere et antal simple sorteringsteknikker, og vi arbejder os gradvis frem mod mere avancerede algoritmer.

Sorteringsproblemet (1).

Total ordning

En total ordning på en mængde S opfylder

- for alle a og b i S gælder netop én af a < b, a = b eller b < a (trichotomi).
- hvis a < b og b < c så er a < c (transitivitet).



Målet med en sortering af objekterne $O_1 \ldots O_n$, med nøglerne hhv. $N_1 \ldots N_n$, hvorpå der er en total ordning, er at bestemme en *permutation* p af tallene 1 .. n således at

$$N_{p(1)} <= N_{p(2)} <= \ldots <= N_{p(n)}$$

PS1 -- Sortering I

Noter

Ovenfor har vi defineret, hvad vi forstår ved en *total ordning* af en mængde. En total ordning er også kendt som en *lineær ordning.*

Vi benytter symbolet "<" i infix notation (a < b) til at formulere, at a kommer forud for b i den ordning, vi fokuserer på i den givne mængde af objekter. På hel naturlig måde udtrykker a <= b at enten er a < b eller a = b.

Vi vil undertiden tillade os at sige, at Oi <= Oj, selv om ordningen egentlig er defineret på nøglerne Ni og Nj.

Angående permutationen p gælder, at det j-te element i den sorterede liste var på plads nummer p(j) i den oprindelige liste.

I kapitlet om søgning og søgetræer omtalte vi første gang *nøgler* og *satellit-information.*

Sorteringsproblemet (2).

Stabil sortering

Sortering kaldes *stabil* hvis to objekter med ens nøgler bevarer deres indbyrdes position i den sorterede liste:

$$i < j \quad \text{and} \quad N_{p(i)} = N_{p(j)} \quad \Rightarrow \quad p(i) < p(j)$$

Sorteringstilfælde:

- Bedste tilfælde: listen er sorteret på forhånd.
- Gennemsnits tilfælde: listen er tilfældig ordnet.
- Værste tilfælde: Afhænger af sorteringsteknikken.

Intern sortering:

Sortering af objekter i internt arbejdslager (RAM).

Ekstern sortering:

Sortering af objekter uden for det interne lager, f.eks. på sekventielle medier såsom magnetbånd.

Noter

Når vi diskuterer **intern sortering**, vil vi antage, at strukturen, der sorteres, er en *liste*. Lister kan enten repræsenteres som arrays eller som som kædede lister. Under alle omstændigheder opfatter vi en liste som en *abstrakt datatype* , hvorpå der findes et antal veldefinerede operationer til aflæsning og skrivning listens elementer. I array-baserede lister er sådanne operationer meget effektive (konstant køretid). Der er endvidere operationer som indsætter og sletter et element imellem to eksisterende elementer i listen. I array-baserede lister vil sådanne operationer kræve, at nogle af de eksisterende elementer skubbes én position. (Køretiden af disse er således lineær i længden af arrayet).

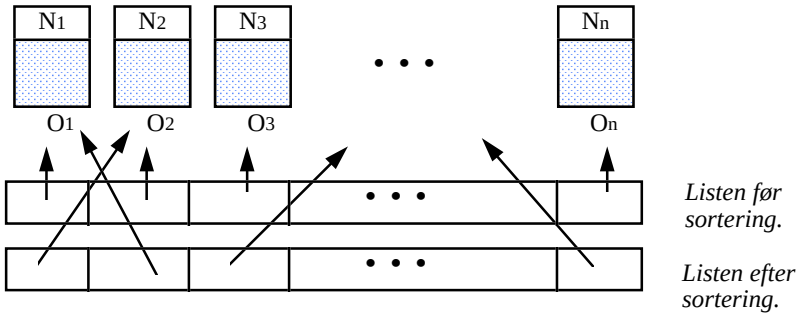
Ekstern sortering er emnet i den sidste af de tre forelæsninger om sortering. I forbindelse med ekstern sortering vil vi indskrænke mængden af operationer på lister til et absolut minimum, således at vi kun har sekventiel adgang til elementerne, "fra venstre mod højre". Vi taler nu om *sekvenser* i stedet for lister. Sekvenser modellerer f.eks. magnetbånd.

Om sorteringstilfælde: For nogle sorteringsalgoritmer er det værste tilfælde omvendt-sorterede lister. Dette er dog ikke altid tilfældet (se f.eks. quicksort). For mange sorteringsalgoritmer er det bedste tilfælde, som angivet ovenfor, at listen på forhånd er sorteret. Dette gælder dog heller ikke altid (se igen quicksort).

Sorteringspragmatik.

Det kan være uhensigtsmæssigt at skulle flytte rundt på hele objekter i en sorteringsproces.

Sortering ved brug af adressetabel:



Noter

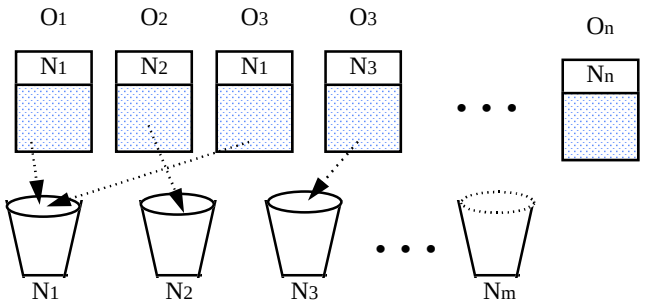
En adressetabel, som vist på denne slide, repræsenterer den tidligere omtalte permutation p. Indholdet af den j-te plades af adressetabllen efter sorteringsprocessen er afsluttet er således en reference til det p(j)'te objekt.

Sortering ved fordeling: Spandsortering.

Givet: L = (O_1 O_2 ... O_n) med nøglerne hhv. N_1 N_2 ... N_n.

Metode:

1. Der allokeres en tabel af køer med en indgang for hver af de m *forskellige* nøgler: T: array[N_1 .. N_m] of queue.
2. Objekterne gennemløbes, og O_i indsættes i køen T[N_i].
3. Køerne sættes sammen; resultatet er den sorterede liste.



Én "spand" for hver (forskellig) nøgle

Køretid:

O(n).

PS1 -- Sortering I

Noter

Den beskrevne form for 'sortering ved fordeling' kaldes for *spandsortering*, fordi hver kø kan opfattes som en "spand", hvor alle objekter med samme nøgle hældes i.

Kø-sammensætning kan defineres som en udvidelse af den algebraiske specifikation af køer, som vi har studeret tidligere (se slide "Algebraisk specifikation af køer" i kapitlet om arrays, lister, stakke og køer):

concat: Queue[X] x Queue[X] -> Queue[X]

concat(create(), Q) = Q

concat(enter(x, P), Q) = enter(x, concat(P, Q))

Mere præcist udtrykt end ovenfor danner vi nu køen

concat(T[n], concat(T[n-1], ... concat(T[2], T[1])...))

Hvis man udtrækker elementerne med kø-operationen front fra denne kø får man den resulterende, sorterede liste. Kødisciplinen på "spandene" medfører, at sorteringen bliver stabil.

Det er kun realistisk at overveje spandsortering, hvis nøglerne i objekterne udgør et mindre interval, såsom heltal fra 10 til 25.

I en praktisk realisering kan man blive nødt til at skulle allokere køer af passende størrelse for at kunne rumme det nødvendige antal objekter. Det kan derfor være nødvendigt at gennemløbe listen af objekter for at finde ud af, hvor mange objekter der findes med en given nøgle.

I forhold til andre sorteringsteknikker, som vi vil stifte bekendtskab med, er spandsortering meget lagerkrævende, idet der skal afsættes plads til køerne.

Køretiden O(n) er bemærkelsesværdig. Forklaringen diskuteres på næste slide (kendskabet til nøgler). Bemærk dog i denne sammenhæng, at sorteringsteknikken kræver ekstra lager.

Sortering ved fordeling: Radix-sortering.

Radix-sortering: Spandsortering i flere faser:

Spandsortering kan organiseres i flere faser ved at udnytte egenskaber i nøglernes indre struktur.

Eksempel 1:

Alle nøgler er 4-cifrede tal:
- Lav først spandsortering (i 100 spande) efter de to mindst betydende cifre.
- Lav dernæst spandsortering efter de to mest betydende cifre.

Eksempel 2:

Alle nøgler er strenge af længde 3 over alfabetet af (små) danske bogstaver:
- Lav først spandsortering (i 28 spande) over første tegn (c_1).
- Gentag for andet (c_2) og tredie tegn (c_3).

En streng: "c_3 c_2 c_1"

Ved 'sortering ved fordeling' kræves der et kendskab til nøglerne (nøglernes struktur) udover antagelsen om eksistens af den totale ordning.

Noter

Ved forelæsningen vil vi gennemføre et konkret eksempel på radix-sortering.

Opgave: Foreslå, hvordan man kan sortere et spil kort ved radix-sortering. Hvad kan man vælge som "spande"? Medbring gerne et spil kort og forsøg at gøre øvelsen så konkret som mulig.

Sortering ved udvælgelse (1).

Givet: L = (O_1 O_2 ... O_n)

Metode:

1. ***Udvælg*** objektet O_{min} med den mindste nøgle i L, udskriv O_{min}, og mærk dette objekt i listen som "uendelig stor".
2. Gentag trin 1 på L, hvorved det næstmindste element osv. findes.
3. Stop processen, når der er udvalgt n objekter.

Noter

Sortering ved udvælgelse er beskrevet intuitivt på denne slide. Det er naturligvis muligt at optimalisere denne sorteringsidé på forskellig vis, når vi skal lave en konkret sorteringsalgoritme for metoden.

For at slippe for at mærke allerede udvalgte objekter vil vi i omstående skitse til algoritme ombytte det mindste element med element 1, det næstmindste med element 2 osv.

Sortering ved udvælgelse, som beskrevet på denne og næste slide, afspejler en meget simpel, letforståelig, og ikke særlig effektiv sorteringsteknik. I den næste forelæsning, som handler om mere avancerede sorteringsteknikker, vil vi vende tilbage til "sortering ved udvælgelse" (Heapsort).

Sortering ved udvælgelse (2).

```
Selection-sort(L: Liste):
  let n be længden af L
  in for i := 1 to n -1 do
      udvælg det mindste element  L[min] i L[i..n];
      ombyt  L[min] og  L[i]
```

Alle objekter skal være tilstede, inden sorteringen kan begynde.

Køretid:

$O(n^2)$ i alle tre sorteringstilfælde.

Noter

Vi antager, at denne og efterfølgende sorteringsalgoritmer har en parameter, som er en liste, der ønskes sorteret.

Sorteringsalgoritmen reorganiserer listen ved mutation (sideeffekt). Alternativt kunne man programmere sorteringsalgoritmerne som funktioner, der returnerer en ny kopi (evt. en permutationstabel) af listen. Dette er mere lagerkrævende.

Principielt er det ikke nødvendigt at beslutte, om listen er baseret på arrays eller om den er sammenkædet. Vi skal blot forudsætte, at den abstrakte datatype Liste har et passende repertoire af operationer. I algoritmen på denne slide er ombytning en vigtig operation.

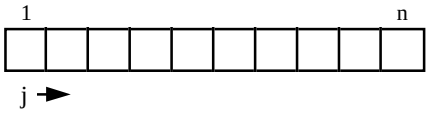
Reelt vil vi i kapitlerne Sortering I og Sortering II antage, at listerne, som sorteres, er baseret på arrays. *De angivne køretider er baseret på denne antagelse*. Årsagen til, at vi som regel vælger arrays er, at de nødvendige liste operationer til sortering (eksempelvis omyt) som regel er mere effektive på array-baserede lister end på kædede lister.

De tre sorteringstilfælde "bedste tilfælde", "gennemsnitstilfældet" og "værste tilfælde" giver nøjagtig de samme antal beregningstrin ved ovenstående sorteringsteknik.

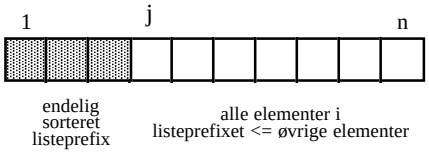
Opgave. Undersøg om den skitserede algoritme til sortering ved udvælgelse (ovenfor) er stabil. Om nødvendig, modificer algoritmen, så den bliver stabil.

Sortering ved udvælgelse fortsættes i næste kapitel: Sortering ved udvælgelse -- Heapsort.

Løkkeinvariant i sortering ved udvælgelse.




```
Selection_sort(n: integer) is
local j, min: integer
do
  from j := 1
  invariant sorted(1, j - 1); interval_leq(1, j-1, j, n)
  until j = n
  loop
    -- udvælg det mindste element i intervallet [j, n]. Lad det have index min
    swap(j, min);
    j := j + 1
  end
ensure sorted(1,n)
end
```



PS1 -- Sortering I

Noter

Vi forestiller os her, at Selection_sort er en operation i en klasse, såsom ARRAY. Parameteren n angiver den øvre grænse i det interval, som skal sorteres. Den nedre grænse er 1.

Her og på efterfølgende slides med løkkeinvarianter for sorteringsalgoritmer, vil følgende definitioner på assertions blive benyttet:

sorted(i, j) betyder

item(i) <= item(i+1) <= .... <= item(j).

interval_leq(i, j, k, l) betyder:

∀ x in [i , j], ∀ y in [k , l]: item(x) <= item(y).

Hvis et interval er tomt, er ovenstående betingelser altid opfyldt.

Læg mærke til hvordan postbetingelsen kan udledes af løkkeinvarianten og termineringsbetingelsen. Når løkken terminerer giver invarianten direkte at

sorted(1, n - 1); interval_leq(1, n-1, n, n).

Altså er de første n-1 elementer sorterede, og det sidste er større eller lig med de n-1 første. Dette giver naturligvis alt i alt sorted(1,n).

Sortering ved indsættelse (1).

Givet: L = (O_1 O_2 ... O_n)

Metode:

Antagelse: Listeprefixet af længde k-1 er sorteret:
($O_{p(1)}$ $O_{p(2)}$ ... $O_{p(k-1)}$) .

1. ***Indsæt*** O_k i det sorterede listeprefix, således at de første k objekter nu er sorterede.

2. Gentag trin 1 for k løbende fra 2 indtil n.

Noter

Sortering ved indsættelse (2).

```
Insertion-sort(L: Liste):
  for j := 2 to længden af L   do
    if not(element(j-1) <=  element(j))
    then slet element j  i L ;
          insert-in-place(1, j - 1, element(j) )

Insert-in-place(min, max: integer; el: Liste-element):
    k := max;
    while k >= min and element(k) >  el
    do k := k - 1;
    indsæt  el efter det k-te element.
```

Objekterne, som skal sorteres, kan genereres undervejs i sorteringsprocessen.

Formuleringen af 'Insertion-sort' er stabil.

Køretid:
Bedste tilfælde: O(n).
Gennemsnits- og værste tilfælde: $O(n^2)$

PS1 -- Sortering I

Noter

I ovenstående algoritmeskitse af sortering ved indsættelse kan man opfatte Insert-in-place som en lokal procedure til Insertion-sort.

Det j-te element af listen L betegnes ovenfor med element(j). Vi kunne alternativt have valgt at skrive L[j] eller L.item(j). Idet formålet er at sætte fokus på algoritme-ideen snarere end en konkret formulering i et programmeringssprog af "sortering ved indsættelse" kan vi leve med denne notation.

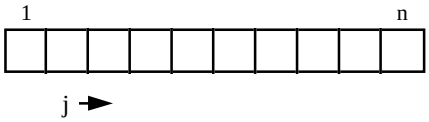
I proceduren insert-in-place kan k blive min - 1 (hvilket er 0 i vor konkrete anvendelse). Indsættelse efter det 0-te element betyder "indsættelse på den første plads i listen".

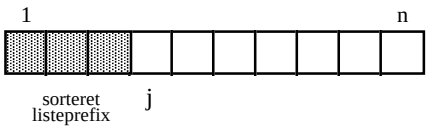
Ved forelæsningen vil vi argumentere for den detaljerede kompleksitet i bedste og værste tilfælde.

Opgave: Programmer ovenstående sorteringsalgoritme i Eiffel. Formuler løkke-invariant samt pre- og postbetingelser på proceduren. Overvej i hvilken klasse sorterings-proceduren skal placeres. Overvej hvilke krav man skal stille til typen T i Liste[T]. Man kan konkret overveje at specialisere en eksisterende klasse til at omfatte en sorterings-operation. Man kan også overveje at indplacere sorterings-operationen i en abstrakt klasse for arrays og/eller lister. Diskuter!

Løkkeinvariant i sortering ved indsættelse.

```
Insertion_sort (n: integer) is
local j: integer
do
  from j := 2
  invariant sorted (1, j-1)
  until j = n + 1
  loop
    -- indsæt item(j) korrekt i listeprefixet [1, j]
    j := j + 1
  end
ensure sorted(1,n)
end
```





PS1 -- Sortering I

Noter

Vi forestiller os også her, at sorteringsoperationen findes i en klasse, såsom ARRAY. Parameteren n angiver den øvre grænse i det interval, som skal sorteres. Den nedre grænse er 1.

Løkkeinvarianten giver direkte postbetingelsen når løkken terminerer.

Shell Sort.

Givet: L = (O_1 O_2 ... O_n)

Metode:

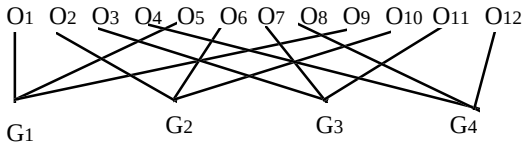
Givet listen $g_1, g_2, ... g_m$ af "gab" hvor

- $g_i > g_{i+1}$ (for $1 <= i < m$)
- $g_m = 1$

g_i -sort:

Sorter g_i grupper af objekter, som indbyrdes har afstand g_i.

Foretag g_1-sort, g_2-sort, ... g_m-sort på listen L.



Grupper i 4-sort:
4 grupper af objekter
med indbyrdes afstand 4

Køretid (worst-case):
$O(n^2)$ --
$O(n^{1.33})$
afhængig af gab-listen

PS1 -- Sortering I

Noter

Formuleringen af Shell-sort er her mere generel end i Weiss. Weiss viser et program for Shell-sort, hvor g1 er n div 2, g2 = g1 div 2, ...

Det er klart, at vilkårlige gab resulterer i en sorteret liste, blot vi sikrer at det sidste gab er 1.

Hver enkelt gruppe i gi-sort kan sorteres ved indsættelse, ved brug af algoritmen fra den forrige slide.

Det er mindre klart, at der er en gevinst i Shell-sort i forhold til f.eks. sortering ved simpel indsættelse, fra de to forrige slides. I sortering ved simpel indsættelse flytter vi elementer over små afstande. I Shell-sort, derimod, starter vi med at flytte elementer over store afstande, og derved opnår vi de sorterede grupper. Senere aktiveringer (gi-sort for større i) har et lettere og lettere arbejde, idet listen af objekter "bliver mere og mere sorteret".

I disse noter nøjes vi med at skitsere ideen i Shell-sort, som gjort på denne slide. Vi henviser til Weiss, som viser detaljerede Pascal procedurer for denne sorteringsteknik.

Nederst på denne slide vises, hvordan 12 objekter deles op i 4 grupper af hver 3 elementer. Dette er gruppeopdelingen i en 4-sort. Man kan naturligvis ikke sikre, at alle grupper har lige mange objekter. Hvis der havde været 11 objekter ovenfor, ville gruppen G4 kun have indeholdt 2 elementer. Dette berører dog ikke princippet i sorteringen.

Tidskompleksiteten af shell-sort er vanskelig at bestemme. Der henvises til Weiss for detaljer om dette.

Sortering ved ombytning (1).
Boblesortering.

Givet: L = (O_1 O_2 ... O_n) med nøglerne hhv. K_1 ...K_n.

Metode:

1. Sammenlign parvis O_1 og O_2, O_2 og O_3, osv.
   Ombyt et par hvis $K_i > K_{i+1}$.

   Det sidste element er nu det største element.

2. Gentag ombytningsprocessen fra trin 1 på mindre og mindre listeprefixer, indtil prefixet bliver tomt.

Boblesortering kan helt naturlig udformes, således at teknikken er stabil.

Noter

Når vi taler om sortering ved ombytning, er det naturligt at se på den kendte, simple sorteringsteknik som kaldes boblesortering.

Opgave: Analyser boblesortering i både det bedste tilfælde (listen er allerede sorteret) og i det værste tilfælde (listen er initialt omvendt sorteret). I analysen skal der holdes regnskab med antal sammenligninger og antal ombytninger i sorteringsprocessen.

Som grundlag for analysen skal man formulere en algoritme(skitse) af boblesortering. Overvej i denne forbindelse, hvornår sorteringsprocessen kan afbrydes.

Sortering ved ombytning fortsættes i næste kapitel: quicksort.