Algoritmer och datastrukturer

F11 - Sorteringsalgoritmer

Repetition

- Instickssortering (insertion sort)
  - Länkade strukturer
  - Arrayer
- Urvalssortering (selection sort)
  - Arrayer (i första hand)
- Shell-sortering (shell sort)
- Bubbelsortering (bubble sort)

Repetition

- Tidsåtgång för instickssortering, urvalssortering och bubbelsortering med avseende på jämförelser:
  - Genomsnittsfallet: $\boldsymbol{O}(n^2)$ och $\boldsymbol{\Omega}(n^2)$
  - Värsta fallet: $\boldsymbol{O}(n^2)$ och $\boldsymbol{\Omega}(n^2)$
- Urvalssortering har $O(n)$ för kopiering
- Teoretisk gräns för jämförelsebaserad sortering
  - $\boldsymbol{\Omega}(n*log(n))$ jämförelser i värsta fallet

Snabbare sortering?

- ”Divide and conquer”
  - Merge sort
  - Quicksort
- Heap sort

Jämförelse mellan olika sorteringsalgoritmer

Exempel på webben:

http://web.cs.wpi.edu/~cs2005/e00/handouts/sorting/sorting.html

Notera att det är Java, så webbläsaren måste tillåte att de körs, och det kan vara en säkerhetsrisk.

Är ”divide and conquer” alltid snabbare?

Nej…

- Sorteringsalgoritmers prestanda beror mycket på hur stor datamängden är
- Enkla sorteringar fungerar ofta snabbare...
  - **om** datamängden redan är *sorterad* eller *nästan sorterad*
  - **om** datamängden är *liten*

”Divide and conquer”

Grundfilosofi?

- Det är enklare att sortera en kort lista än en lång!
  - Att sortera en lista med noll eller ett element är busenkelt - den är ju redan sorterad!

En generisk D&C-sortering

Om listan har mer än ett element

- *Dela* listan i två delar (sublistor)
- *Sortera* varje sublista var för sig
- *Kombinera* de två sorterade sublistorna till en sorterad lista

För att vi ska tjäna tid är det viktigt att delningsfasen och kombineringsfasen inte tar för lång tid!

D&C: Delningsfasen

Hur gör vi för att *dela* listan i två delar?

- MS: Dela listan på hälften i varje steg
  - *Kan göras blixtsnabbt i en array*
  - *Kan göras på linjär tid i en länkad lista*
- QS: Dela upp i ”mindre” respektive ”större” element
  - *Kan göras på linjär tid*

D&C: Kombineringsfasen

Hur *kombinerar* vi de sorterade delarna?

- MS: Operationen *Merge* (”sammansmält”)
  - *Kan göras på linjär tid, men kräver en extra array*
- QS: [Mindre] + [Större]
  - *Kan göras mycket snabbt i en array*

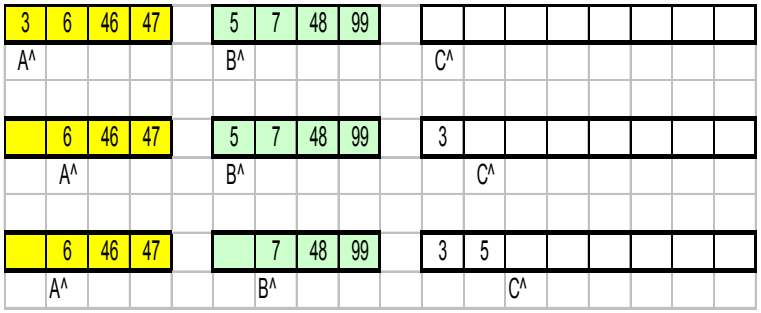
Merge sort (Array)

Fundamental operation: *Merge*

- Sammansmältning av två *sorterade* listor
- Kan göras i ett svep (linjär tid) genom indatamängden
  - Förutsättning: Utdata får placeras i en ny lista

Sammansmältning (merge)

Sammansmältning av två sorterade arrayer



Sammansmältning av två arrayer

Två inputarrayer (A, B)

- Två räknare (A^, B^)

En outputarray (C)

- En räknare (C^)

Räknarna ”pekar” från början på första elementet i respektive array

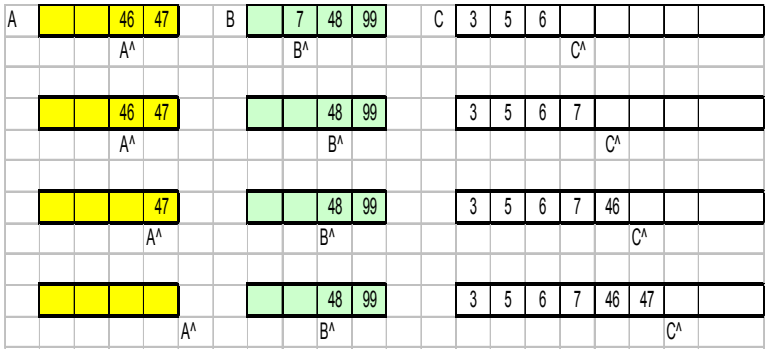
Sammansmältning av två arrayer (forts)

Algoritm för *Merge*:

- Kopiera det *mindre* av elementen A[A^] och B[B^] till C[C^]
- Öka motsvarande räknare ett steg
- När slutet av A eller B har nåtts, kopiera resten av den andra listan till C

Sammansmältning (forts)

Fortsättning på tidigare exempel



Merge sort av en array

Om listans längd är 0 eller 1

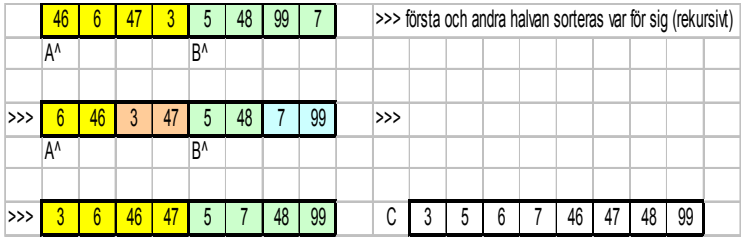
- … så är listan redan sorterad
  - ➔ returnera listan

Annars

- Dela arrayen i två halvor
- Sortera rekursivt *första* halvan
- Sortera rekursivt *andra* halvan
- Sammansmält (merge) de två sorterade halvorna
  - ➔ returnera resultatet

Merge Sort (forts)

Exempel



Merge sort av en länkad lista

Hur delar vi en länkad lista i två delar?

firstHalf := head;

secondHalf := ???

En **idé**: traversera listan med två referenser (*current* och *midpoint*)

current := head; midpoint := head; från början

Flytta fram *current* två steg för varje gång som *midpoint* flyttas fram ett steg

Merge sort av en länkad lista

Sätt *secondHalf* till noden efter *midpoint* (*secondHalf* := *midpoint.next*)

Sätt *midpoint.next* till **nil**

Merge sort (Länkad lista)

Om *list = null* eller *list.next = null*

… så är listan redan sorterad
➔ returnera listan

Annars

Dela listan (*firstHalf*, *secondHalf*) enligt ovan

Sortera rekursivt *firstHalf*

Sortera rekursivt *secondHalf*

Sammansmält *firstHalf* och *secondHalf*
➔ returnera resultatet

Sammansmältning av två länkade listor

- Kan göras iterativt genom att ändra om länkar i listnoderna
  - utlista := **null** från början
  - Så länge som ingen av de sorterade sublistorna är tom
    - … jämför de båda sublistornas huvuden
    - … och ta bort det huvud som är mindre och sätt in det i utlistan
  - Häng på den (eventuellt) kvarvarande icke-tomma listan på utlistan

Analys av Merge sort

- De rekursiva anropen kan illustreras med ett binärt *rekursionsträd*
  - Sortering av första halvan motsvarar vänster subträd
  - Sortering av andra halvan motsvarar höger subträd

Rekursionsträd för Merge sort

- Rekursionsträdet för *Merge sort* är alltid *välbalanserat*
- Varför?
  - Delningen skapar alltid två lika stora sublistor
- Slutsats?
  - Trädets höjd är *log(n)*
    - *n* = antal löv (= antal element i listan)

Analys av Merge sort (forts)

- Jämförelser görs bara vid *merge*
- På varje nivå i rekursionsträdet deltar varje nod i en sammansmältning
  - Alltså: Upp till *n* jämförelser per nivå

Analys av Merge sort (forts)

log(n) nivåer
➔ Totalt *n*log(n)* jämförelser i värsta fallet

Alltså: ***O****(n*log(n))* i värsta fallet
- Nära den ”ideala” algoritmen!
  - Möjligt att göra vissa ytterligare optimeringar

Quick sort

Inget *Merge*-steg behövs
- I stället flyttas elementen i samband med att listan delas upp i två delar

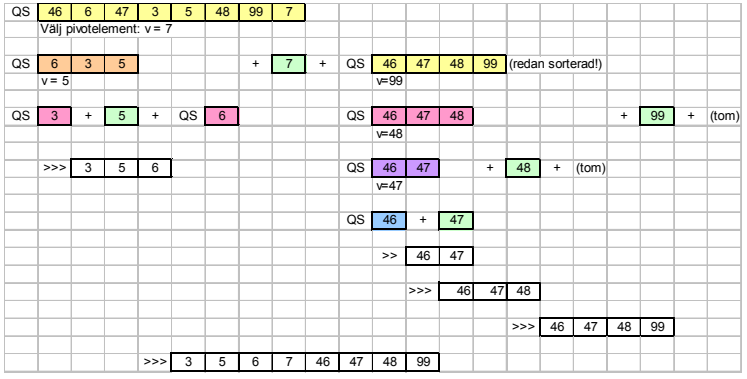
Quick sort

Om *length(L) = 0 eller 1*
- … så är L redan sorterad
  ➔ returnera L

Annars
- Välj ett ”pivotelement” *v* i L
- Partitionera *L-{v}* i två delar
  - L_1 = *{alla element i L som är **mindre** än v}*
  - L_2 = *{alla element i L som är **större** än v}*
- Returnera *QuickSort(L_1)+{v}+QuickSort(L_2)*

Quick sort: Ett exempel

QuickSort(46,6,47,3,5,48,99,7)



Val av pivotelement

Vi ser i exemplet att valet av pivotelement har stor betydelse:

- Högra sublistan sorteras trots att den redan är helt sorterad
  - Med v=47 (eller 48) hade sorteringen gått snabbare
- Vänstra sublistan sorteras snabbt!

Viktig fråga: Hur bör pivotelementet väljas?

Val av pivotelement (forts)

Metod 1: *Första* eller *sista* elementet

- Enkel metod… men ett dåligt val om listan är sorterad eller nästan sorterad
  - Se exemplet!

Metod 2: *Mittersta* elementet

Metod 3: *Slumpmässigt* element

Metod 4: *”Median of three”*

- Välj **medianen** av *första*, *sista* och *mittersta* elementet

Analys av Quick sort

Tidsåtgången beror på...

- Indatamängdens utseende
- Valet av pivotelement
  - Ett bra val av pivotelement (metod 4) kan minska exekveringstiden ordentligt
  - Vanlig insticksortering är oftast snabbare om n<20

Tidsåtgång för Quick sort

Genomsnittsfallet: $\boldsymbol{O}(n*log(n))$

Värsta fallet: $\boldsymbol{\Omega}(n^2)$ och $\boldsymbol{O}(n^2)$

- ... vid konsekvent dåliga val av pivotelement
  - Exempel: Om första elementet i sublistan väljs som pivotelement, och listan redan är sorterad, så kommer den vänstra sublistan alltid att vara tom efter partitioneringen
    - Mycket ineffektivt! Vi får ett rekursionsträd som är *mycket obalanserat*, vilket gör att djupet av trädet närmar sig *n*.

MergeSort eller QuickSort?

- Värsta fallet är bättre för MS än för QS!
- Trots det används QS för det mesta
  - MS kräver oftast mer minne
  - Värsta fallet för QS är ovanligt om algoritmen använder en bra strategi för valet av pivotelement