Logaritmiske sorteringsalgoritmer

Logaritmisk sortering

- Rekursive og “splitt og hersk” metoder:
  - Deler verdiene i arrayen i to (helst) omtrent like store deler i henhold til et eller annet *delingskriterium*
  - Hver mindre del sorteres rekursivt på samme måte
  - Delene settes deretter sammen til ferdig sortert array
  - Er mer effektivt, oftest *O(n log(n))*
- To logaritmiske metoder i dette kurset:
  - Quicksort
  - Flettesortering

Effektivitet av logaritmiske sorteringsalgoritmer

- Hvis vi deler arrayen i to omtrent like store deler hver gang, blir det ca. log(n) nivåer med rekursive kall
- Hvis arbeidet som gjøre på hvert nivå *samlet* er $O(n)$, vil hele sorteringen bli $O(n \log(n))$
- Hvis oppdelingen er svært skjev (en stor og en svært liten del) kan vi få opp til n rekursive nivåer, og algoritmens effektivitet kan synke til $O(n^2)$

Quicksort

- Tony Hoare, 1960
- Algoritmen eg. laget for automatisk oversetting
- Oftest meget rask i praksis
- Virker for generelle sorteringsproblemer
- Krever svært lite ekstra hukommelse
- Gjennomsnittlig effektivitet er $O(n \log(n))$
- "Worst-case" er $O(n^2)$

Quicksort: Sortere array *A* av lengde *n*

Hvis $n > 1$:

- Velg et element *p* (partisjoneringselement) i *A*, f.eks. element nr. 0 eller nr. *n*/2 *
- Bytt om på elementene i *A* (partisjonér arrayen) slik at den deles i to deler:
  - Alle elementer som er mindre eller lik *p* står til venstre
  - Alle elementer som er større enn *p* står til høyre
  - Partisjoneringselementet *p* står mellom de to delene
- Sortér de to delene rekursivt med quicksort, hele arrayen er da ferdig sortert

*: Lærebokens valg

Quicksort: Eksempel

Bruker *første* element til å partisjonere delarrayene

Nivå 1: 26 33 35 29 19 12 22

19 22 12 26 29 35 33

Nivå 2: 19 22 12 26 29 35 33

12 19 22 26 29 35 33

Nivå 3: 12 19 22 26 29 35 33

12 19 22 26 29 33 35

Ferdig: 12 19 22 26 29 33 35

Quicksort: Et eksempel til

Nivå 1: 65 57 81 92 43 31 26 75 13 10

26 57 10 13 43 31 65 75 92 81

Nivå 2: 26 57 10 13 43 31 65 75 92 81

10 13 26 57 43 31 65 75 92 81

Nivå 3: 10 13 26 57 43 31 65 75 92 81

10 13 26 43 31 57 65 75 81 92

Nivå 4: 10 13 26 43 31 57 65 75 81 92

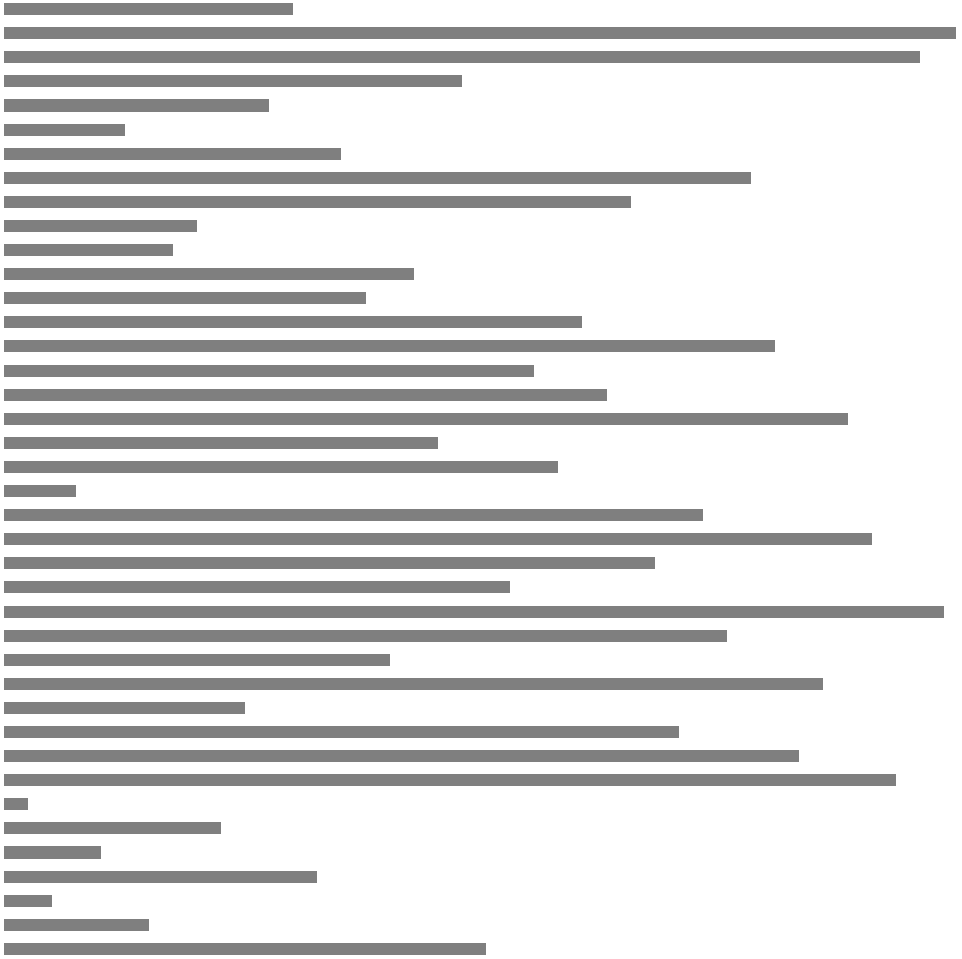
10 13 26 31 43 57 65 75 81 92

Ferdig: 10 13 26 31 43 57 65 75 81 92

Partisjoneringsalgoritmen

6 8 1 4 9 0 3 5 2 7	6 er partisjoneringselement
6 8 1 4 9 0 3 5 2 7	7 står riktig plassert
6 8 1 4 9 0 3 5 2 7	swap 2 og 8
6 2 1 4 9 0 3 5 8 7	1 står riktig plassert
6 2 1 4 9 0 3 5 8 7	4 står riktig plassert
6 2 1 4 9 0 3 5 8 7	swap 5 og 9
6 2 1 4 5 0 3 9 8 7	0 og 3 står riktig plassert
3 2 1 4 5 0 6 9 8 7	swap 6 og 3, ferdig

Quicksort: Animasjon



Effektivitet og implementasjon

- Implementeres med to rekursive kall, der parameterene er øvre og nedre indeks for arraysegmentet som skal sorteres
- Skiller ut partisjoneringen i en egen metode
- Er $O(n \log(n))$ i gjennomsnitt (random data)
- Hvis partisjoneringen gir mange skjeve oppdelinger (f.eks. ved nesten sorterte data) vil Quicksort dele opp arrayen ~n ganger, og algoritmen blir $O(n^2)$
- Se Java-koden

Effektivisering av Quicksort

- Forbedring av valg av partisjoneringselement, for å redusere muligheten for skjev oppdeling:
  - Sammenlign f.eks. elementene på indeksene 0, *n*/2 og *n* -1, og bruk verdien som er i midten
- Ikke la rekursjonen gå helt ned til lengde lik 1:
  - Bruk en enklere og "lettere" ikke-rekursiv metode, f.eks. innstikksortering, til å sortere korte segmenter av arrayen

Flettesortering (merge sort)

- John von Neumann(!), 1945
- Velegnet for “steinalderens” sekvensielle lagrings-media som magnettape, papirtape og hullkort
- Utmerket for sortering av lenkede lister (oppgave)
- Garanterer $O(n \log(n))$ effektivitet
- Standard implementasjon krever bruk av ekstra arrayer til å kopiere dataene frem og tilbake under sorteringen

Flettesortering av array *A* av lengde *n*

Hvis $n > 1$:

- Sortér nedre og øvre halvdel av *A* hver for seg, rekursivt med flettesortering
- *Flett* de to halvdelene sammen til en sortert array
- Hele arrayen er da ferdig sortert

Flettesortering: Eksempel

26 33 35 29 19 12 22

26 33 35 29 19 12 22 – oppdeling

26 33 35 29 19 12 22 – oppdeling

26 33 35 29 19 12 22 – bunn i rekursjonen

26 33 29 35 12 19 22 – fletting

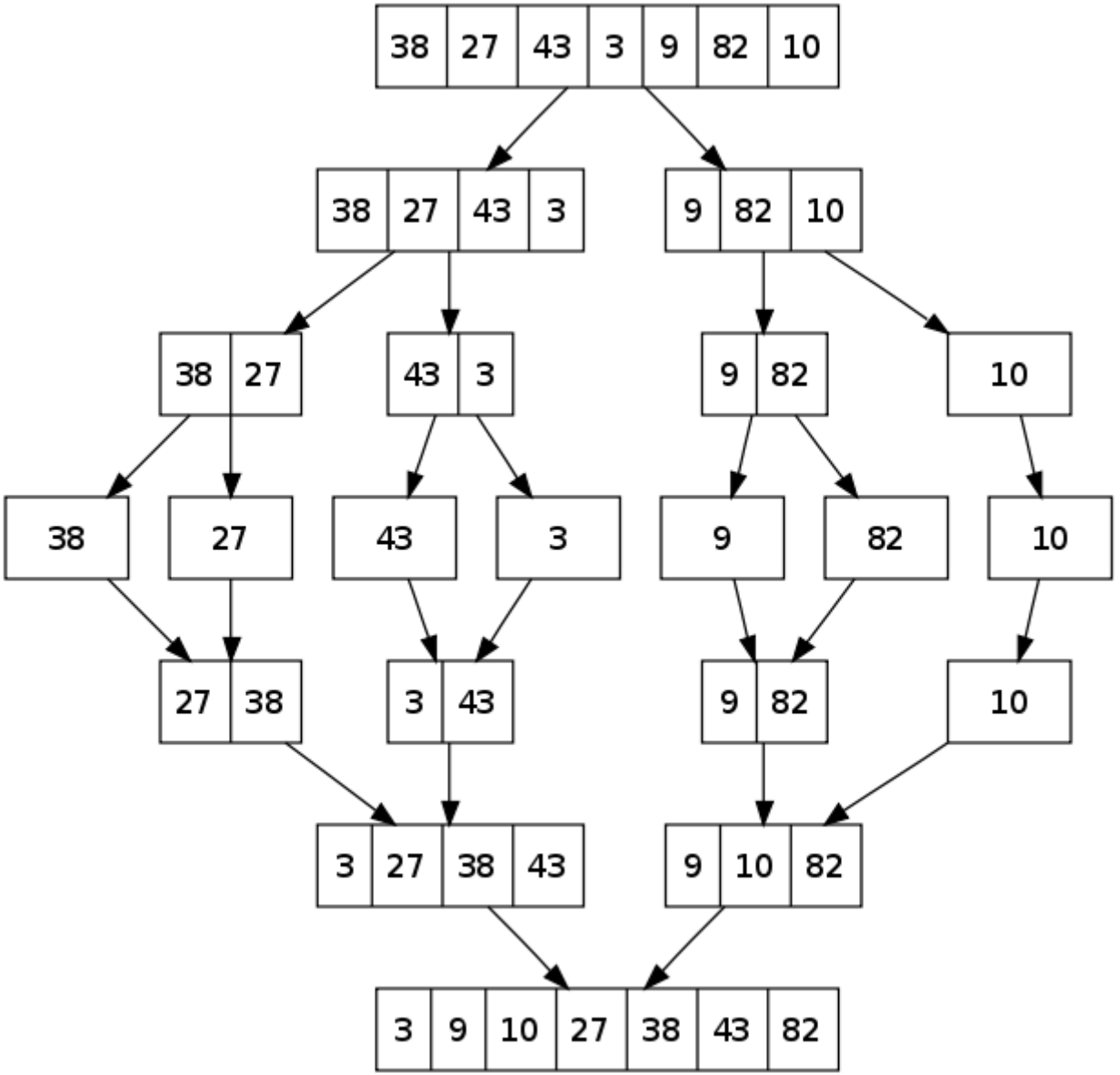
26 29 33 35 12 19 22 – fletting

12 19 22 26 29 33 35 – ferdig

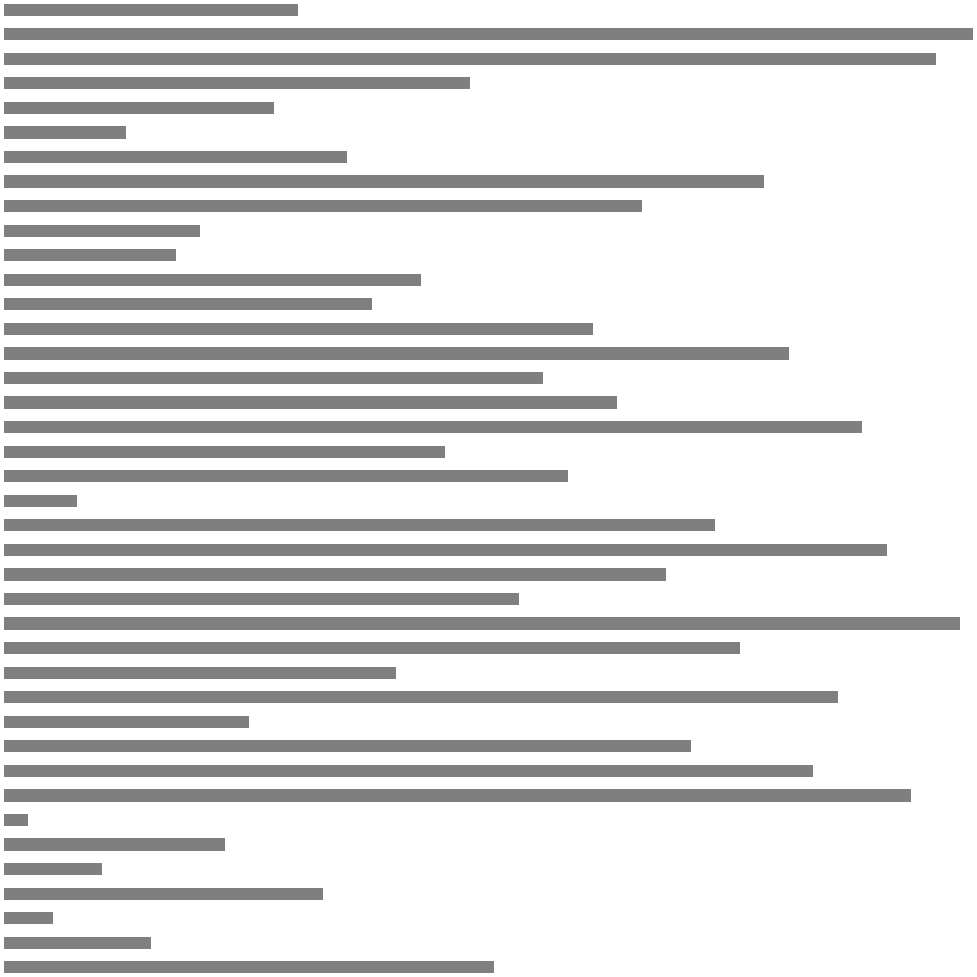
Fletting av to sorterte arraysegmenter

1 13 24 26	2 15 27 38	___ ___ ___ ___ ___ ___ ___ ___
1 13 24 26	2 15 27 38	1 ___ ___ ___ ___ ___ ___ ___
1 13 24 26	2 15 27 38	1 2 ___ ___ ___ ___ ___ ___
1 13 24 26	2 15 27 38	1 2 13 ___ ___ ___ ___ ___
1 13 24 26	2 15 27 38	1 2 13 15 ___ ___ ___ ___
1 13 24 26	2 15 27 38	1 2 13 15 24 ___ ___ ___
1 13 24 26	2 15 27 38	1 2 13 15 24 26 ___ ___
1 13 24 26	2 15 27 38	1 2 13 15 24 26 27 ___
1 13 24 26	2 15 27 38	1 2 13 15 24 26 27 38

Flettesortering: Kalltre for *n* = 7



Flettesortering: Animasjon



Effektivitet og implementasjon

- Implementeres med to rekursive kall, parametrene er øvre og nedre indeks for arraysegmentet som skal sorteres
- Flettingen gjøres ved å kopiere dataene over i en ekstra array og deretter flette de to halvdelene tilbake
- Programmering av flettingen krever litt "indeksfikling"
- Er alltid $O(n \log(n))$, men krever $O(n)$ ekstra hukommelse i tillegg til arrayen som sorteres*
- Se Java-koden

*: Flettesortering *kan* implementeres med konstant plassforbruk, komplisert algoritme

Quicksort vs. flettesortering

- Quicksort er *vesentlig* raskere i de aller fleste tilfeller
- Flettesortering må *alltid* flytte data mellom temporær og original array og blir derfor langsommere
- Flettesortering er alltid $O(n \log(n))$, men...
- Hvis det er viktig med garantert $O(n \log(n))$ oppførsel, er det bedre å bruke en "in-house" algoritme som ikke swapper så mye, som f.eks. heapsort
- Se testprogram for sorteringer