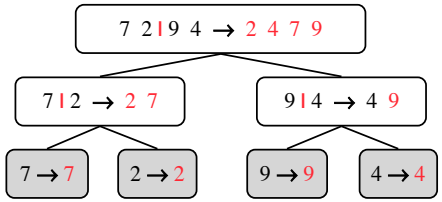
Sortering

Sortering ved fletning (merge-sort)



Del-og-hersk

Del-og-hersk er et generelt paradigme til algoritmedesign

Del: opdel input-data S i to disjunkte delmængder, S_1 og S_2

Løs (rekursivt): løs de delproblemer, der svarer til S_1 og S_2

Hersk: kombiner løsningerne for S_1 og S_2 til en løsning for S

Basistilfældet for rekursionen er delproblemer af størrelse 0 eller 1

Merge-sort er en sorteringsmetode baseret på del-og-hersk-paradigmet

Ligesom heap-sort
- benyttes et comparator-objekt
- har den $O(n \log n)$ køretid

I modsætning til heap-sort
- benyttes ingen hjælpe-prioritetskø
- tilgås data sekventielt (hensigtsmæssigt ved sortering på en disk)

Merge-sort

Merge-sort på en input-sekvens S med n elementer består at tre trin:

Del: opdel S i to sekvenser, S_1 og S_2, hver med cirka $n/2$ elementer

Løs: sorter rekursivt S_1 og S_2

Hersk: flet S_1 og S_2 til en sorteret sekvens

Algorithm *mergeSort*(**S, C**)
  Input sequence S with n elements, comparator C
  Output sequence S sorted according to C
  if $S.size() > 1$ **then**
    $(S_1, S_2) \leftarrow$ *partition*$(S, n/2)$
    mergeSort(S_1, C)
    mergeSort(S_2, C)
    $S \leftarrow$ *merge*(S_1, S_2)

Fletning af to sorterede sekvenser

Hersk-trinet i merge-sort består i at flette to sorterede sekvenser A and B til en sorteret sekvens S, der indeholder foreningsmængden af A og B

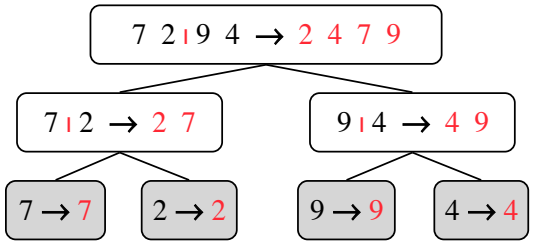
Fletning af to sorterede sekvenser implementeret ved hjælp af en hægtet liste tager $O(n)$ tid

Algorithm *merge*(*A*, *B*)
  Input sequences *A* and *B*
  Output sorted sequence of $A \cup B$

  $S \leftarrow$ empty sequence
  while $\neg A.isEmpty() \wedge \neg B.isEmpty()$ **do**
    if *A.first*().*element*() < *B.first*().*element*() **then**
      S.insertLast(*A.remove*(*A.first*()))
    else
      S.insertLast(*B.remove*(*B.first*()))
  while $\neg A.isEmpty()$ **do**
    S.insertLast(*A.remove*(*A.first*()))
  while $\neg B.isEmpty()$ **do**
    S.insertLast(*B.remove*(*B.first*()))
  return *S*

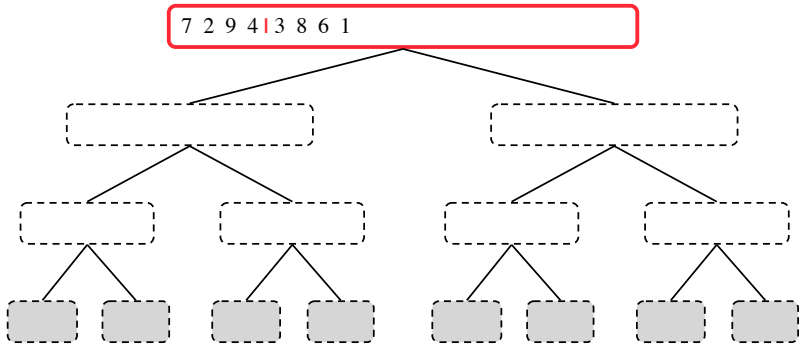
Merge-sort-træ

En udførelse af merge-sort kan anskueliggøres ved hjælp af et binært træ
  Hver knude repræsenterer et rekursivt kald af **mergeSort** og indeholder
    • en usorteret sekvens før sin opdeling
    • en sorteret sekvens efter udførelsen
  Roden er det første kald
  Bladene er delsekvenser af længde 0 eller 1



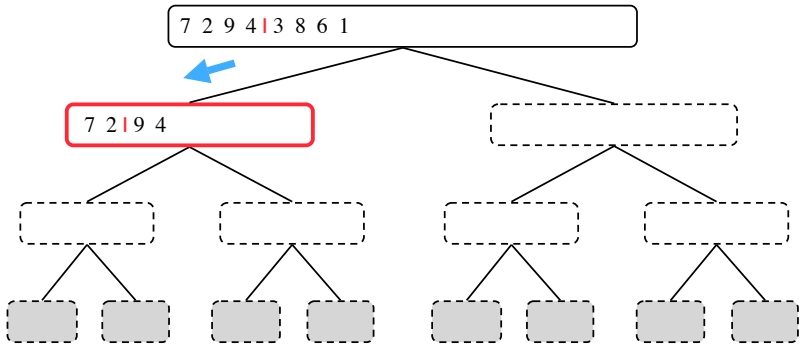
Eksempel på udførelse

Opdeling



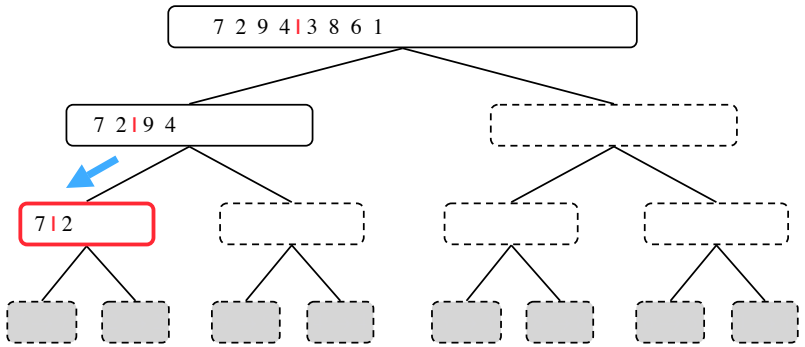
Eksempel på udførelse (fortsat)

Rekursivt kald, opdeling



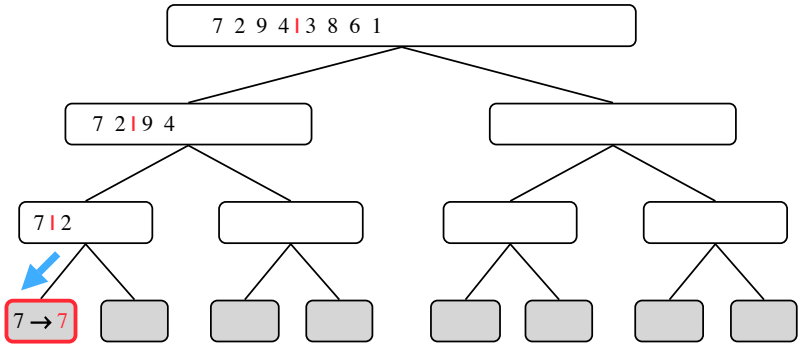
Eksempel på udførelse (fortsat)

Rekursivt kald, opdeling



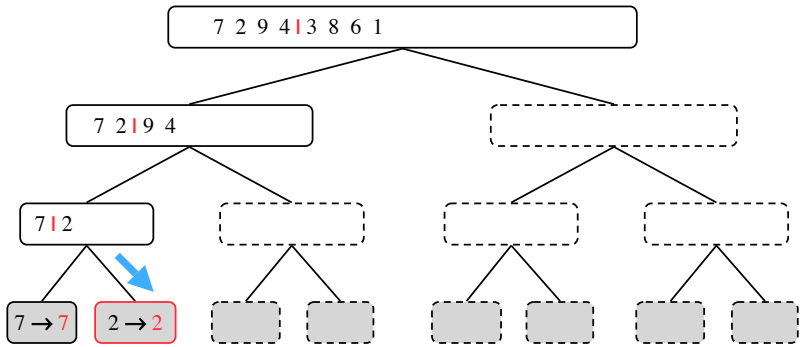
Eksempel på udførelse (fortsat)

Rekursivt kald, basistilfælde



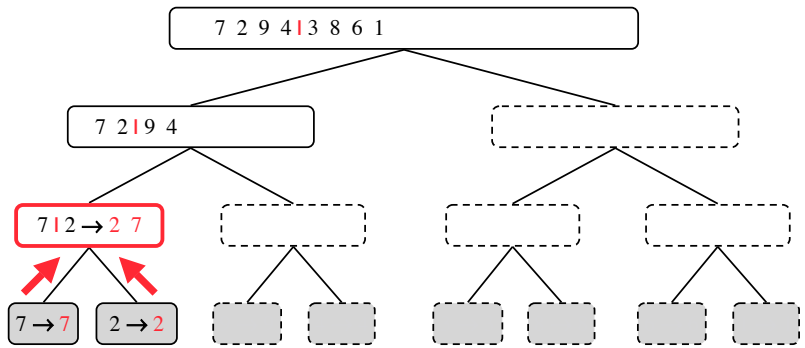
Eksempel på udførelse (fortsat)

Rekursivt kald, basistilfælde



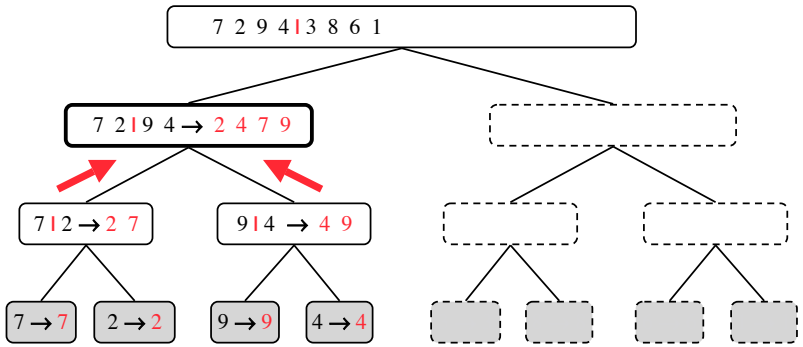
Eksempel på udførelse (fortsat)

Fletning



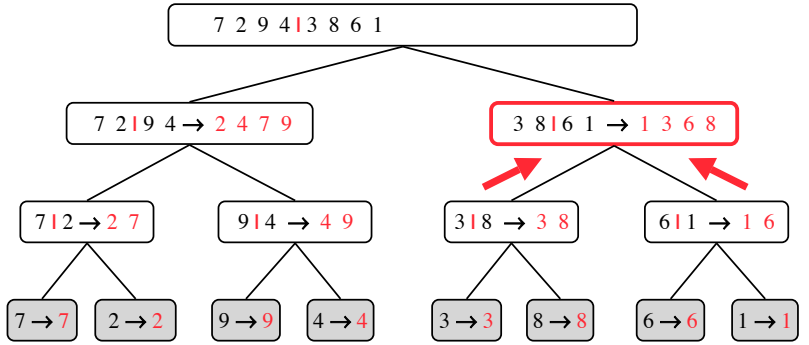
Eksempel på udførelse (fortsat)

Fletning



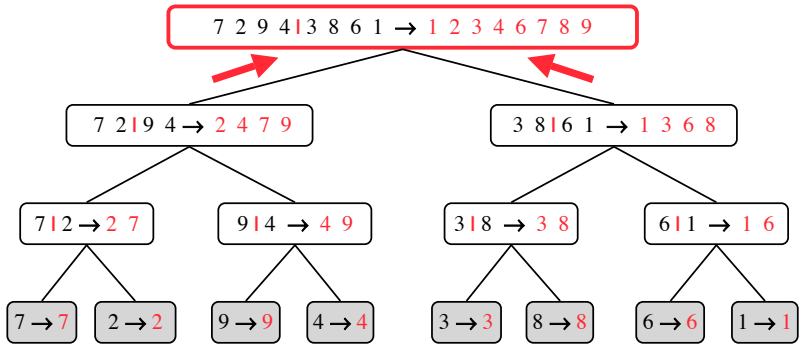
Eksempel på udførelse (fortsat)

Recursivt kald, …, fletning, fletning



Eksempel på udførelse (fortsat)

Fletning



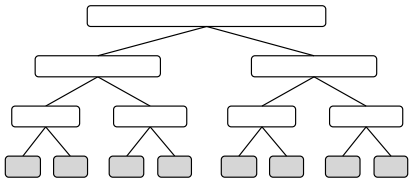
Analyse af merge-sort

Højden af merge-sort-træet er $O(\log n)$
- ved hvert rekursivt kald halveres sekvensen

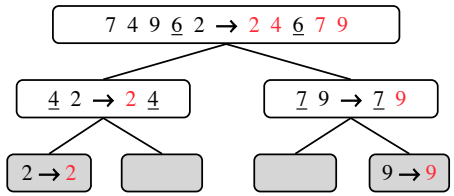
Den totale arbejdsmængde for knuderne med dybde i er $O(n)$
- vi opdeler og fletter 2^i sekvenser, som hver har længden $n/2^i$
- vi foretager 2^{i+1} rekursive kald

Derfor er den samlede køretid for merge-sort $O(n \log n)$

dybde	#sekv	længde
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



Quick-sort



Quick-sort

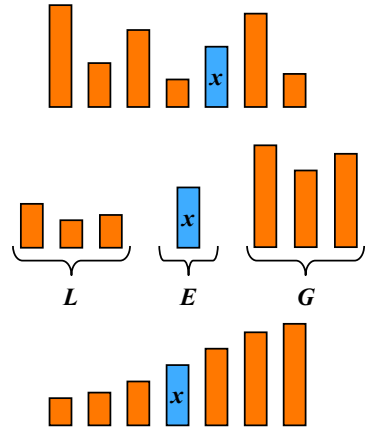
(Hoare, 1962)

Quick-sort er en randomiseret sorteringsalgoritme, der er baseret på del-og-hersk-paradigmet

Del: vælg et tilfældigt element x (kaldet **pivot**-elementet) og opdel S i
- L : elementer mindre end x
- E : elementer lig med x
- G : elementer større end x

Løs: sorter L og G

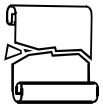
Hersk: sammenføj L, E og G



Pseudokode for quick-sort

```
Algorithm quickSort(S, C)
    Input sequence S with n
        elements, comparator C
    Output sequence S sorted
        according to C
    if S.size() > 1 then
        x ← element of S
        (L, E, G) ← partition(S, x)
        quickSort(L, C)
        quickSort(G, C)
        S ← join(L, E, G)
```


Opdeling



Vi opdeler input-sekvensen, som følger:
- Vi fjerner, et efter et, ethvert element y fra S
- Vi overfører y til L, E eller G, afhængigt af udfaldet af sammenligningen med pivot-elementet, x

Hver indsættelse og fjernelse sker enten i starten eller i slutningen af en sekvens og tager derfor $O(1)$ tid

Opdelingen i quick-sort tager derfor $O(n)$ tid

```
Algorithm partition(S, p)
    Input sequence S, position p of pivot
    Output subsequences L, E, G of the
        elements of S less than, equal to,
        or greater than the pivot, resp.
    L, E, G ← empty sequences
    x ← S.elementAtRank(p)
    while ¬S.isEmpty() do
        y ← S.remove(S.first())
        if y < x then
            L.insertLast(y)
        else if y = x then
            E.insertLast(y)
        else { y > x }
            G.insertLast(y)
    return L, E, G
```

Quick-sort-træ

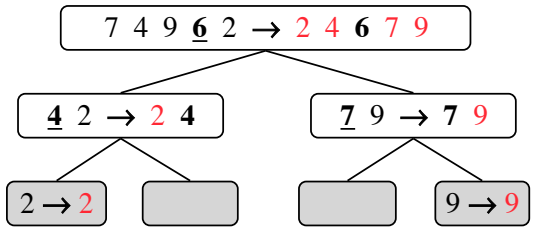
En udførelse af quick-sort kan anskueliggøres ved hjælp af et binært træ

Hver knude repræsenterer et rekursivt kald af **quickSort** og indeholder

- en usorteret sekvens før sin opdeling
- en sorteret sekvens efter udførelsen

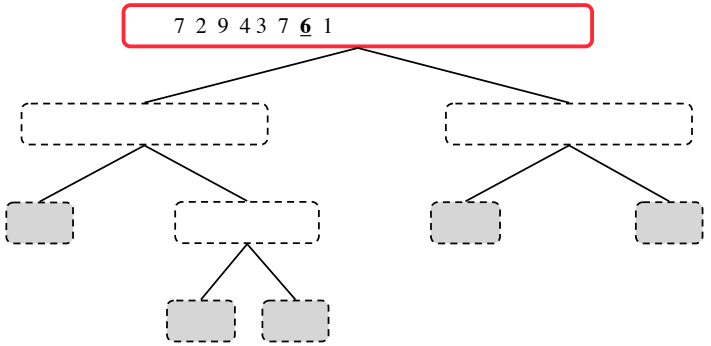
Roden er det første kald

Bladene er delsekvenser af længde 0 eller 1



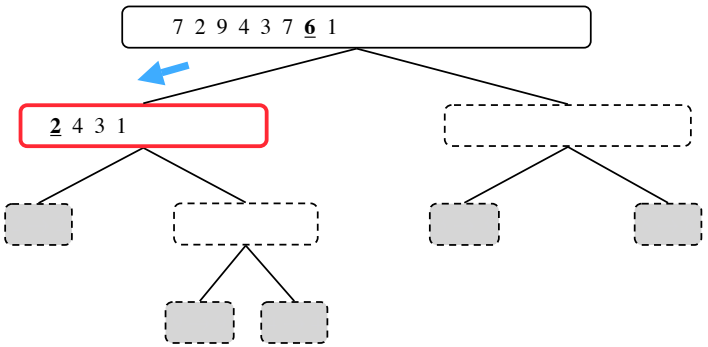
Eksempel på udførelse

Valg af pivot-element



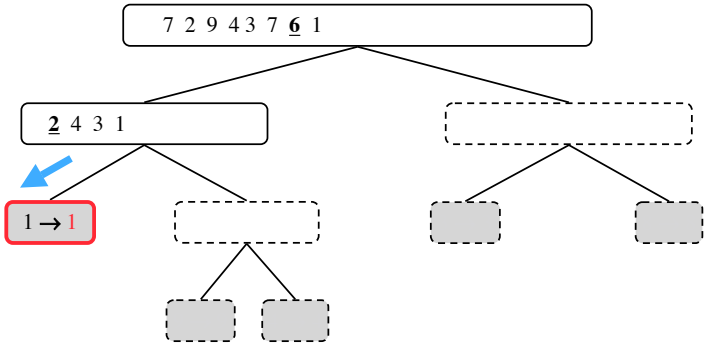
Eksempel på udførelse (fortsat)

Opdeling, rekursivt kald, valg af pivot-element



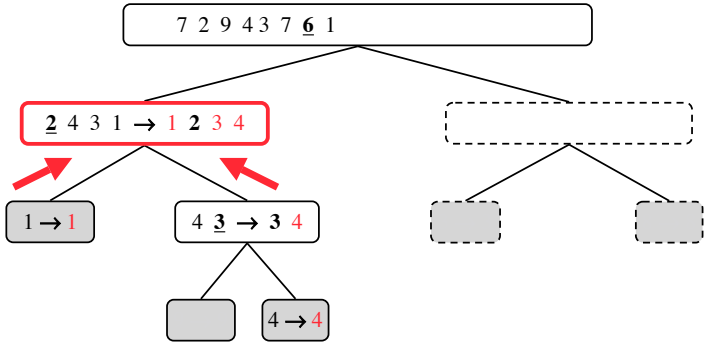
Eksempel på udførelse (fortsat)

Opdeling, rekursivt kald, basistilfælde



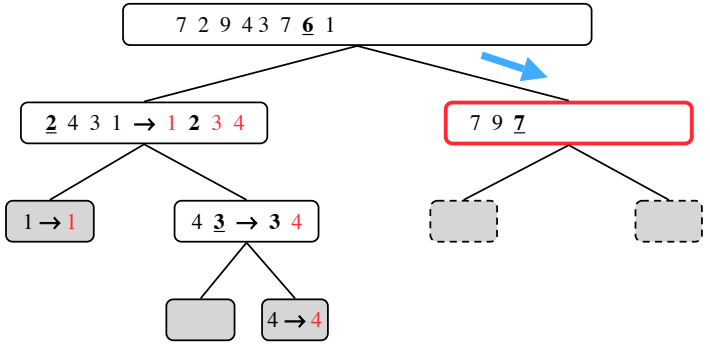
Eksempel på udførelse (fortsat)

Rekursivt kald, …, basistilfælde, forening



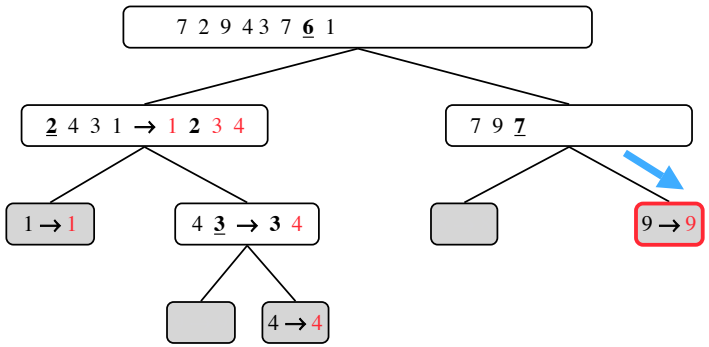
Eksempel på udførelse (fortsat)

Rekursivt kald, valg af pivot-element



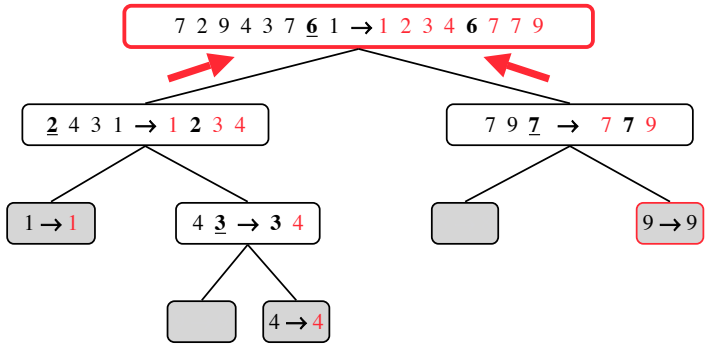
Eksempel på udførelse (fortsat)

Opdeling, …, rekursivt kald, basistilfælde



Eksempel på udførelse (fortsat)

Forening, forening



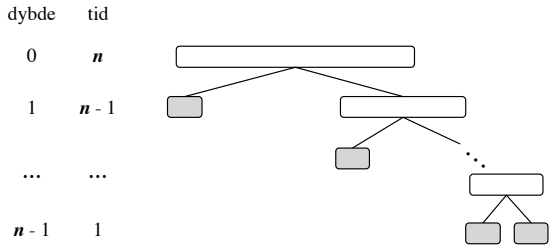
Køretid i værste tilfælde

Det værste tilfælde for quick-sort optræder, når pivot-elementet er det unikke minimum- eller maksimum-element. I dette tilfælde har enten L eller G længden $n - 1$, og den anden har længden 0

Køretiden er proportional med summen

$$n + (n - 1) + \ldots + 2 + 1$$

Den værste køretid for quick-sort er således $O(n^2)$



Forventet køretid

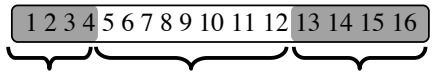
Betragt det rekursive kald af quick-sort for en sekvens af længde s

Godt kald: længderne af L og G er begge mindre end $3s/4$

Dårligt kald: enten har L eller G en længde større end $3s/4$

Godt kald **Dårligt kald**

Et kald er godt med sandsynligheden 1/2



Dårlige pivot-elementer **Gode pivot-elementer** **Dårlige pivot-elementer**

Forventet køretid (del 2)

Sandsynlighedsteoretisk faktum:
Det forventede antal møntkast, der skal til for at for få k kroner, er $2k$

For en knude med dybden i forventer vi, at

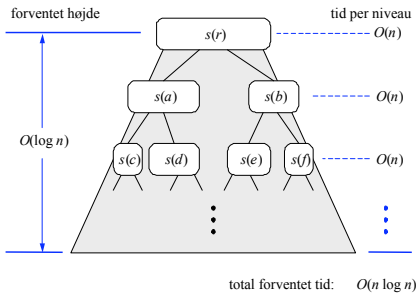
- $i/2$ forfædre er gode kald, hvilket betyder, at
- længden af input-sekvensen for det aktuelle kald er højst $(3/4)^{i/2}n$

Derfor har vi

- For en knude med dybden $2\log_{4/3} n$ har den forventede input-sekvens længden 1
- Den forventede højde af quick-sort-træet er $O(\log n)$

Den samlede arbejdsmængde for alle knuder med samme dybde er $O(n)$

Derfor er den forventede køretid for quick-sort $O(n \log n)$



Quick-sort på stedet

Quick-sort kan implementeres, så den der kører "på stedet" (in-place)

I opdelingstrinet benyttes erstatnings-operationer for at omordne input-sekvensens elementer således, at

- elementer mindre end pivot-elementet får rang mindre end h
- elementer lig med pivot-elementet får rang imellem h and k
- elementer større end pivot-elementet får rang større end k

De rekursive kald behandler

- elementer med rang mindre end h
- elementer med rang større end k

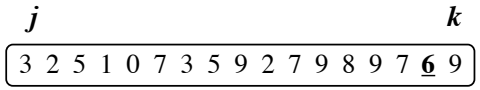
Algorithm *inPlaceQuickSort*(S, l, r)
  Input sequence S, ranks l and r
  Output sequence S with the elements of rank between l and r rearranged in increasing order
  if $l \geq r$ **then**
    return
  $i \leftarrow$ *a random integer between* l *and* r
  $x \leftarrow S.elemAtRank(i)$
  $(h, k) \leftarrow inPlacePartition(S, x)$
  $inPlaceQuickSort(S, l, h - 1)$
  $inPlaceQuickSort(S, k + 1, r)$

Opdeling på stedet

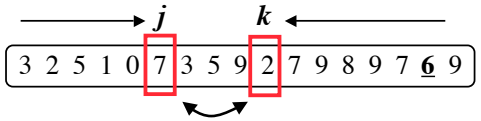
Benyt to indices, ***j*** og ***k***, til at opsplitte ***S*** til ***L*** og ***E*** ∪ ***G***

Gentag, indtil ***j*** og ***k*** krydser hinanden:

- Søg til højre, indtil der findes et element ≥ ***x***.
- Søg til venstre, indtil der mødes et element < ***x***
- Ombyt de to elementer på indeks ***j*** og ***k***



(pivot = 6)

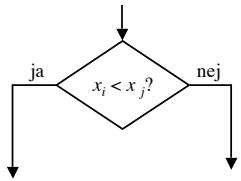


Sammenligningsbaseret sortering

Mange sorteringsalgoritmer er baseret på sammenligning
De sorterer ved at foretage sammenligninger imellem par af objekter

Eksempler: bubble-sort, selection-sort, insertion-sort, heap-sort, merge-sort, quick-sort, ...

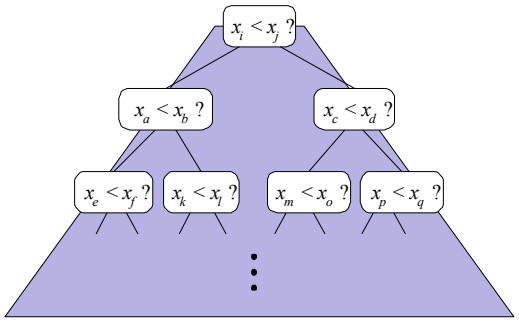
Lad os udlede en nedre grænse for køretiden for enhver algoritme, der bruger sammenligninger til at sortere n elementer $x_1, x_2, \ldots, x_n$



Tælling af sammenligninger

Lad os tælle sammenligningerne

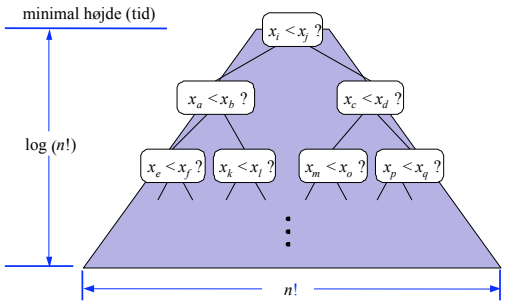
Enhver mulig udførelse af algoritmen svarer til en rod-til-blad-vej i et **beslutningstræ**



Højden af beslutningstræet

Højden af beslutningstræet er en nedre grænse for køretiden
Enhver mulig input-permutation må føre til hvert sit blad (output)
I modsat fald vil nogle input-permutationer, f.eks. ...4...5... og ...5...4..., få samme output, hvilket ville være forkert

Da der således er $n! = 1*2*\ldots*n$ blade, er højden mindst $\log(n!)$



Den nedre grænse

Enhver sammenligningsbaseret sorteringsalgoritme bruger mindst $\log(n!)$ tid i værste tilfælde

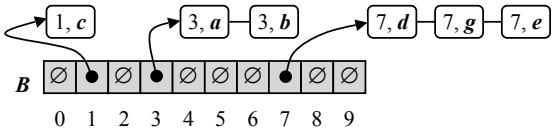
Derfor må enhver sådan algoritme have en køretid, der i værste tilfælde er mindst

$$\log(n!) \geq \log\left(\frac{n}{2}\right)^{\frac{n}{2}} = (n/2)\log(n/2).$$

mindst $n/2$ faktorer er $\geq n/2$

Enhver sammenligningsbaseret sorteringsalgoritme kører i $\Omega(n \log n)$ tid i værste tilfælde

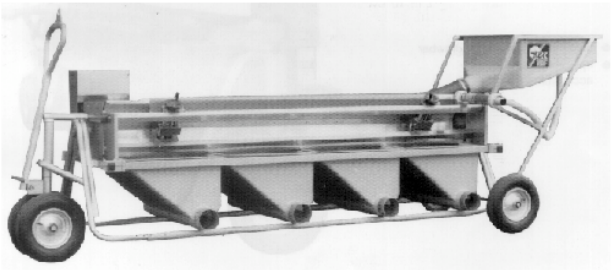
Bucket-sort og Radix-sort



Technology for Fish

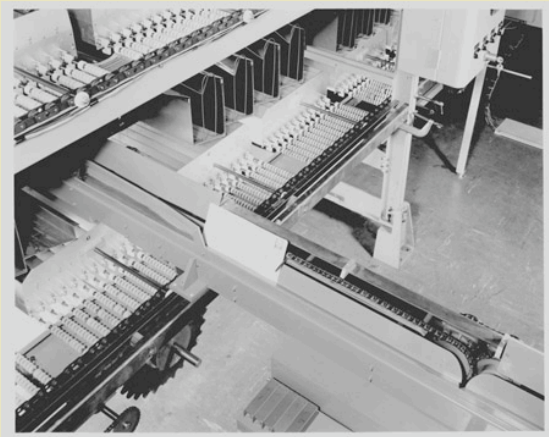
Fully Auto Sorting Machine

for large and medium-sized operations.



Letter Sorting Machine

Letter Sorter Conveyer Belt



Bucket-sort

Lad S være en sekvens af n (nøgle, element)-emner med heltalsnøgler i intervallet $[0, N-1]$

Bucket-sort bruger nøgler som indices i et hjælpe-array B af sekvenser (spande)

Fase 1: Tøm sekvensen S ved at flytte hvert emne (k, o) til sin spand $B[k]$

Fase 2: For $i = 0, \ldots, N-1$, flyt emnerne i spanden $B[i]$ til slutningen af sekvensen S

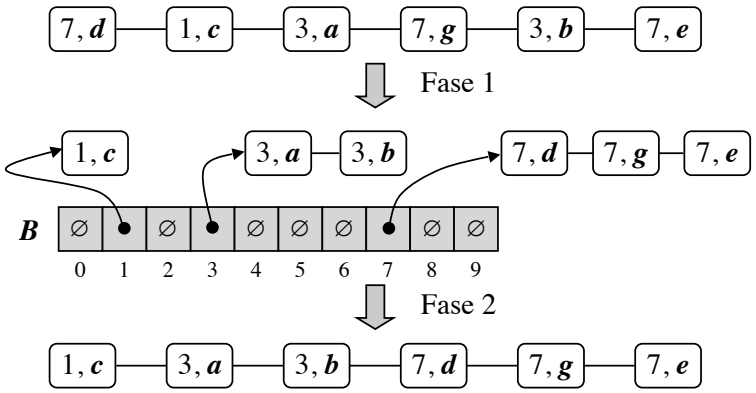
Analyse:

- Fase 1 tager $O(n)$ tid
- Fase 2 tager $O(n+N)$ tid
- Bucket-sort tager altså $O(n+N)$ tid

Algorithm *bucketSort*(S, N)
- **Input** sequence S of (key, element) items with keys in the range $[0, N-1]$
- **Output** sequence S sorted by increasing keys
- $B \leftarrow$ array of N empty sequences
- **while** $\neg S.isEmpty()$ **do**
  - $f \leftarrow S.first()$
  - $(k, o) \leftarrow S.remove(f)$
  - $B[k].insertLast((k, o))$
- **for** $i \leftarrow 0$ **to** $N-1$ **do**
  - **while** $\neg B[i].isEmpty()$ **do**
    - $f \leftarrow B[i].first()$
    - $(k, o) \leftarrow B[i].remove(f)$
    - $S.insertLast((k, o))$

Eksempel

Nøgleinterval [0, 9]



Egenskaber og udvidelser

Nøgle-type-egenskaben
- Nøglerne bruges som indices i et array og kan derfor ikke være vilkårlige objekter
- Der er intet comparator-objekt

Stabilitets-egenskaben
- Den indbyrdes rækkefølge for to elementer med samme nøgle bevares efter udførelsen af algoritmen

Udvidelser
- Heltalsnøgler i intervallet $[a, b]$
  Flyt emnet (k, o) til spanden $B[k-a]$
- Nøgler fra en mængde D af mulige tekststrenge, hvor D har en konstant størrelse (f.eks. navnene på Danmarks 100 største byer)
  Sorter D og beregn rangen $r(k)$ for enhver streng k fra D i den sorterede sekvens
  Flyt et emne (k, o) til spanden $B[r(k)]$

Leksikografisk orden

Et d-tupel er en sekvens af d nøgler $(k_1, k_2, \ldots, k_d)$
Nøgle k_i siges at være i den i'te dimension af tuplet

Eksempel: De Cartesiske koordinater for et punkt i rummet er et 3-tupel

Den **leksikografiske orden** for to d-tupler er defineret rekursivt som følger

$$(x_1, x_2, \ldots, x_d) < (y_1, y_2, \ldots, y_d)$$
$$\Leftrightarrow$$
$$x_1 < y_1 \vee (x_1 = y_1 \wedge (x_2, \ldots, x_d) < (y_2, \ldots, y_d))$$

D.v.s. Tuplerne sammenlignes i første dimension, så i anden, o.s.v.

Leksikografisk sortering

- Lad C_i være comparator-objektet, der sammenligner to tupler i deres i'te dimension
- Lad ***stableSort***(S, C) være en stabil sorteringsalgoritme, der bruger comparator-objektet C
- LexicographicSort sorterer en sekvens af d-tupler i leksikografisk orden ved d gange at udføre algoritmen ***stableSort***, en gang for hver dimension
- LexicographicSort kører i $O(dT(n))$ tid, hvor $T(n)$ er køretiden for ***stableSort***

Algorithm *lexicographicSort*(S)
  Input sequence S of d-tuples
  Output sequence S sorted in lexicographic order
  for $i \leftarrow d$ **downto** 1 **do**
    stableSort(S, C_i)

Eksempel:

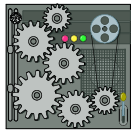
(7,4,6) (5,1,5) (2,4,6) (2,1,4) (3,2,4)

(2,1,4) (3,2,4) (5,1,5) (7,4,6) (2,4,6)

(2,1,4) (5,1,5) (3,2,4) (7,4,6) (2,4,6)

(2,1,4) (2,4,6) (3,2,4) (5,1,5) (7,4,6)

Radix-sort



- Radix-sort er en specialisering af lexicographicSort, som benytter bucketSort som den stabile sorteringsmetode i enhver dimension
- Radix-sort kan anvendes til tupler, hvor nøglerne i hver dimension i er heltal i intervallet $[0, N-1]$
- Radix-sort kører i $O(d(n+N))$ tid

Algorithm *radixSort*(S, N)
  Input sequence S of d-tuples such that $(0, \ldots, 0) \le (x_1, \ldots, x_d)$ and $(x_1, \ldots, x_d) \le (N-1, \ldots, N-1)$ for each tuple $(x_1, \ldots, x_d)$ in S
  Output sequence S sorted in lexicographic order
  for $i \leftarrow d$ **downto** 1 **do**
    bucketSort(S, N)

Radix-sort af binære heltal

- Betragt en sekvens af n b-bit tal

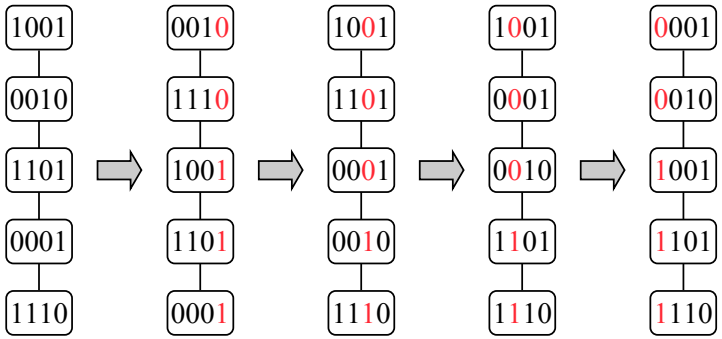
  $x = x_{b-1} \ldots x_1 x_0$

- Vi repræsenterer hvert element som et b-tupel af heltal fra intervallet [0, 1] og anvender radix-sort med $N = 2$
- Denne anvendelse af radix-sort kører i $O(bn)$ tid
- For eksempel kan vi sortere en sekvens af 32-bit heltal i lineær tid

Algorithm *binaryRadixSort*(S)
  Input sequence S of b-bit integers
  Output sequence S sorted
  replace each element x of S with the item $(0, x)$
  for $i \leftarrow 0$ **to** $b-1$ **do**
    replace the key k of each item (k, x) of S with bit x_i of x
    bucketSort$(S, 2)$

Eksempel

Sortering af en sekvens af 4-bit heltal



Udvælgelse

Udvælgelsesproblemet

Lad der være givet et heltal k og n elementer $x_1, x_2, \ldots, x_n$, der opfylder en total ordningsrelation. Find det k'te mindste element i denne mængde.

Vi kan selvfølgelig sortere mængden i $O(n \log n)$ tid, og derefter indicere det k'te element.

k=3

7 4 9 6 2 → 2 4 **6** 7 9

Kan vi løse problemet hurtigere?

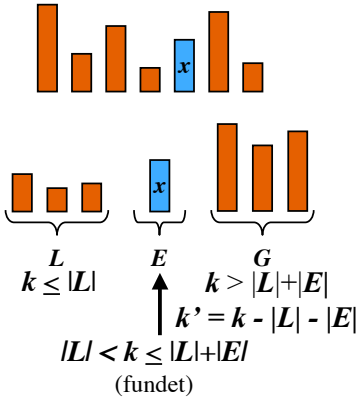
Quick-select

Quick-select er en **randomiseret** udvælgelses-algoritme

Afskær: vælg et tilfældigt element ***x*** (**pivot**-elementet) og opdel ***S*** i

- ***L :*** elementer mindre end ***x***
- ***E :*** elementer lig med ***x***
- ***G :*** elementer større en ***x***

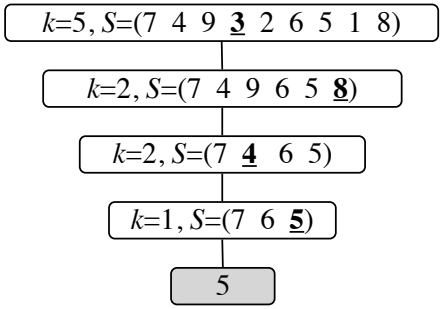
Søg: afhængigt af k, søg svaret i ***E,*** eller søg rekursivt i enten ***L*** eller ***G***



Visualisering af quick-select

En udførelse af quick-select kan anskueliggøres ved hælp af en rekursionsvej

Hver knude repræsenterer et rekursivt kald af quick-select og indeholder k og den resterende sekvens



Forventet køretid

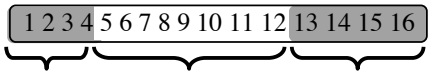
Betragt det rekursive kald af quick-select for en sekvens af længde ***s***

Godt kald: længderne af ***L*** og ***G*** er begge mindre end $3s/4$

Dårligt kald: enten ***L*** eller ***G*** has har en længde større end eller lig med $3s/4$

Godt kald **Dårligt kald**

Et kald er godt med sandsynligheden 1/2



Dårlige pivot-elementer **Gode pivot-elementer** **Dårlige pivot-elementer**

Forventet køretid (del 2)

Sandsynlighedsteoretisk faktum 1: Det forventede antal møntkast, der skal til for at få krone, er 2

Sandsynlighedsteoretisk faktum 2: Forventning er en lineær funktion
$E(X + Y) = E(X) + E(Y)$
$E(cX) = cE(X)$

Lad $T(n)$ betegne den *forventede* køretid for quick-select

Ved faktum 2:
$T(n) \leq T(3n/4) + bn$*(forventet antal kald inden et godt kald),
hvor $b > 0$ er en konstant

Ved faktum 1:
$T(n) \leq T(3n/4) + 2bn$

$T(n)$ er en opad begrænset af en kvotientrække:
$T(n) \leq 2bn + 2b(3/4)n + 2b(3/4)^2n + 2b(3/4)^3n + \ldots$

Da kvotienten er mindre end 1, konvergerer rækken.
Så $T(n)$ er $O(n)$