

Sortering i lineær tid

Nedre grænse for sammenligningsbaseret sortering

Nedre grænse for *alle* sorteringsalgoritmer. Kræver en præcis definition af sorteringsalgoritme.

Nedre grænse for sammenligningsbaseret sortering

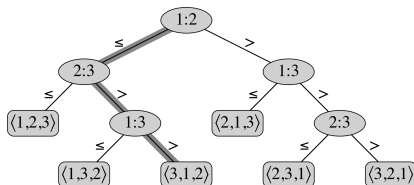
Nedre grænse for *alle* sorteringsalgoritmer. Kræver en præcis definition af sorteringsalgoritme.

Sammenligningbaseret: elementer kan sammenlignes med andre elementer, men ikke deltage i andre operationer.

- ▶ Grundlæggende handling: sammenligning af to elementer i input.
- ▶ Grundlæggende svar: opstilling som skal laves for at få sorteret orden.
- ▶ ID for elementer: deres oprindelige position (index) i input.

Nedre grænse for sammenligningsbaseret sortering

Præcis model som definerer begrebet “sammenligningsbaserede sorteringsalgoritmer”:

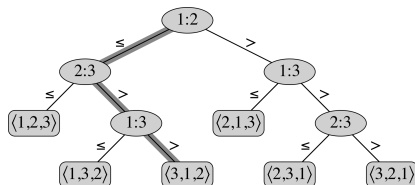


Labels for indre knuder: array-indeks for to input elementer der sammenlignes.

Labels for blade (svar når algoritmen stopper): hvilken opstilling der skal laves for at få sorteret orden (angivet med array-indeks for input elementer).

Nedre grænse for sammenligningsbaseret sortering

Præcis model som definerer begrebet “sammenligningsbaserede sorteringsalgoritmer”:



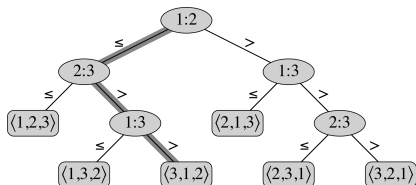
Labels for indre knuder: array-indeks for to input elementer der sammenlignes.

Labels for blade (svar når algoritmen stopper): hvilken opstilling der skal laves for at få sorteret orden (angivet med array-indeks for input elementer).

Worst-case køretid: længste rod-blad sti = træets højde.

Nedre grænse for sammenligningsbaseret sortering

Præcis model som definerer begrebet “sammenligningsbaserede sorteringsalgoritmer”:



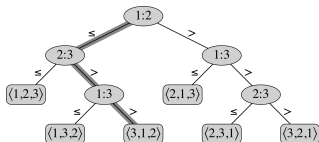
Labels for indre knuder: array-indeks for to input elementer der sammenlignes.

Labels for blade (svar når algoritmen stopper): hvilken opstilling der skal laves for at få sorteret orden (angivet med array-indeks for input elementer).

Worst-case køretid: længste rod-blad sti = træets højde.

Bemærk: Insertionsort, selectionsort, mergesort, quicksort, heapsort kan alle beskrives sådant (annotér elementer med deres index i input).

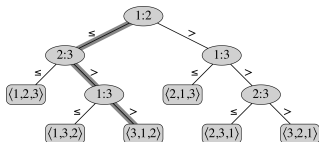
Sammenligningsbaseret sortering



For en fast samling af n elementer er der $n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot \dots \cdot n$ forskellige input (rækkefølger af elementer).

Hvis algoritmen (træet) skal kunne sortere alle disse, skal der være mindst $n!$ blade - ellers vil der være to forskellige input som leder til samme svar, og for det ene input må svaret være forkert.

Sammenligningsbaseret sortering

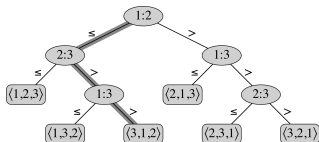


For en fast samling af n elementer er der $n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot \dots \cdot n$ forskellige input (rækkefølger af elementer).

Hvis algoritmen (træet) skal kunne sortere alle disse, skal der være mindst $n!$ blade - ellers vil der være to forskellige input som leder til samme svar, og for det ene input må svaret være forkert.

Et træ af højde h har højst 2^h blade (da det fulde træ af højde h har det).

Sammenligningsbaseret sortering



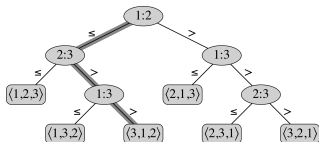
For en fast samling af n elementer er der $n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot \dots \cdot n$ forskellige input (rækkefølger af elementer).

Hvis algoritmen (træet) skal kunne sortere alle disse, skal der være mindst $n!$ blade - ellers vil der være to forskellige input som leder til samme svar, og for det ene input må svaret være forkert.

Et træ af højde h har højst 2^h blade (da det fulde træ af højde h har det).

$$2^h \geq \text{antal blade} \geq n!$$

Sammenligningsbaseret sortering



For en fast samling af n elementer er der $n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot \dots \cdot n$ forskellige input (rækkefølger af elementer).

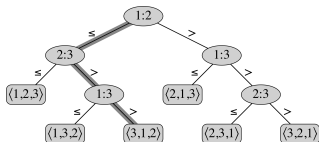
Hvis algoritmen (træet) skal kunne sortere alle disse, skal der være mindst $n!$ blade - ellers vil der være to forskellige input som leder til samme svar, og for det ene input må svaret være forkert.

Et træ af højde h har højst 2^h blade (da det fulde træ af højde h har det).

$$2^h \geq \text{antal blade} \geq n!$$

$$h \geq \log(n!) = \log(1 \cdot 2 \cdot 3 \cdot \dots \cdot n)$$

Sammenligningsbaseret sortering



For en fast samling af n elementer er der $n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot \dots \cdot n$ forskellige input (rækkefølger af elementer).

Hvis algoritmen (træet) skal kunne sortere alle disse, skal der være mindst $n!$ blade - ellers vil der være to forskellige input som leder til samme svar, og for det ene input må svaret være forkert.

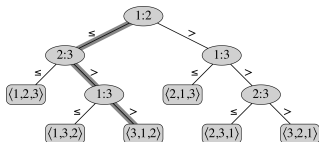
Et træ af højde h har højst 2^h blade (da det fulde træ af højde h har det).

$$2^h \geq \text{antal blade} \geq n!$$

$$h \geq \log(n!) = \log(1 \cdot 2 \cdot 3 \cdot \dots \cdot n)$$

$$= \log(1) + \log(2) + \dots + \log(n/2) + \dots + \log(n) \geq \frac{n}{2} \cdot \log\left(\frac{n}{2}\right) = \frac{n}{2}(\log(n) - 1)$$

Sammenligningsbaseret sortering



For en fast samling af n elementer er der $n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot \dots \cdot n$ forskellige input (rækkefølger af elementer).

Hvis algoritmen (træet) skal kunne sortere alle disse, skal der være mindst $n!$ blade - ellers vil der være to forskellige input som leder til samme svar, og for det ene input må svaret være forkert.

Et træ af højde h har højst 2^h blade (da det fulde træ af højde h har det).

$$2^h \geq \text{antal blade} \geq n!$$

$$h \geq \log(n!) = \log(1 \cdot 2 \cdot 3 \cdot \dots \cdot n)$$

$$= \log(1) + \log(2) + \dots + \log(n/2) \dots + \log(n) \geq \frac{n}{2} \cdot \log\left(\frac{n}{2}\right) = \frac{n}{2}(\log(n) - 1)$$

$$h = \Omega(n \log n)$$

Counting sort

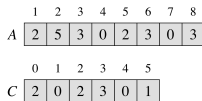
Elementer heltal: elementer kan bruges som array-indeks (≠ at bruge sammenligninger på elementer).

Counting sort: Sorterer n heltal af størrelse mellem 0 og k (inkl.).

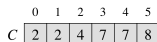
Inputarray: A (længde n)

Outputarray: B (længde n)

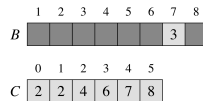
Array af tællere for hver mulig elementværdi: C (længde $k + 1$)



(a)



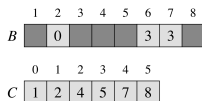
(b)



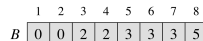
(c)



(d)



(e)



(f)

Counting sort

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3
C	2	0	2	3	0	1		

(a)

	0	1	2	3	4	5
C	2	2	4	7	7	8

(b)

	1	2	3	4	5	6	7	8
B								3
C	0	1	2	3	4	5		

(c)

	1	2	3	4	5	6	7	8
B	0							3
C	1	2	4	6	7	8		

(d)

	1	2	3	4	5	6	7	8
B	0				3	3		
C	1	2	4	5	7	8		

(e)

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

(f)

COUNTING-SORT(A, B, k)

for $i = 0$ **to** k

$C[i] = 0$

for $j = 1$ **to** $A.length$

$C[A[j]] ++$

for $i = 1$ **to** k

$C[i] = C[i] + C[i - 1]$

for $j = A.length$ **downto** 1

$B[C[A[j]]] = A[j]$

$C[A[j]] --$

Tid: $O(n + k)$

Bemærk: stabil (da sidste løkke løber baglæns gennem både A og B), dvs at elementer med ens værdier beholder deres indbyrdes plads.

Radix sort

Radix sort: Sorterer n heltal alle med d cifre i base (radix) k .

(dvs. cifrene er heltal i $\{0, 1, 2, \dots, k - 1\}$)

På figuren nedenfor er der 7 heltal med 3 cifre i base 10.

`RADIX-SORT(A, d)`

`for $i = 1$ to d`

`use a stable sort to sort A on digit i from right`

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Tid: $O(d(n + k))$ hvis der bruges Counting Sort i **for**-løkken.

Korrekthed:

Efter i 'te iteration af **for**-løkken er A sorteret hvis man kun kigger på de i cifre mest til højre.

Radix sort

Eksempel: 32-bits heltal

11011001 10011000 01101000 10110101

Countingsort sorterer disse i tid $O(n + 2^{32})$

Dette er $O(n)$ hvis $n \geq 2^{32} = 4.294.967.296$

Radix sort

Eksempel: 32-bits heltal

11011001 10011000 01101000 10110101

Countingsort sorterer disse i tid $O(n + 2^{32})$

Dette er $O(n)$ hvis $n \geq 2^{32} = 4.294.967.296$

Se som 2-cifrede tal i base 2^{16} (bemærk: sorteret orden er den samme)

11011001 10011000 01101000 10110101

Radixsort sorterer disse i tid $O(2(n + 2^{16}))$

Dette er $O(n)$ hvis $n \geq 2^{16} = 65.536$

Radix sort

Eksempel: 32-bits heltal

11011001 10011000 01101000 10110101

Countingsort sorterer disse i tid $O(n + 2^{32})$

Dette er $O(n)$ hvis $n \geq 2^{32} = 4.294.967.296$

Se som 2-cifrede tal i base 2^{16} (bemærk: sorteret orden er den samme)

11011001 10011000 01101000 10110101

Radixsort sorterer disse i tid $O(2(n + 2^{16}))$

Dette er $O(n)$ hvis $n \geq 2^{16} = 65.536$

Se som 4-cifrede tal i base 2^8 (bemærk: sorteret orden er den samme)

11011001 10011000 01101000 10110101

Radixsort sorterer disse i tid $O(4(n + 2^8))$

Dette er $O(n)$ hvis $n \geq 2^8 = 256$