

6. Suchen

Lineare Suche, Binäre Suche [Ottman/Widmayer, Kap. 3.2, Cormen et al, Kap. 2: Problems 2.1-3,2.2-3,2.3-5]

Das Suchproblem

Gegeben

- Menge von Datensätzen.

Telefonverzeichnis, Wörterbuch, Symboltabelle

- Jeder Datensatz hat einen Schlüssel k .
- Schlüssel sind vergleichbar: eindeutige Antwort auf Frage $k_1 \leq k_2$ für Schlüssel k_1, k_2 .

Aufgabe: finde Datensatz nach Schlüssel k .

Suche in Array

Gegeben

- Array A mit n Elementen ($A[1], \dots, A[n]$).
- Schlüssel b

Gesucht: Index k , $1 \leq k \leq n$ mit $A[k] = b$ oder "nicht gefunden".

22	20	32	10	35	24	42	38	28	41
1	2	3	4	5	6	7	8	9	10

Lineare Suche

Durchlaufen des Arrays von $A[1]$ bis $A[n]$.

- **Bestenfalls** 1 Vergleich.
- **Schlimmstenfalls** n Vergleiche.

Suche im sortierten Array

Gegeben

- Sortiertes Array A mit n Elementen ($A[1], \dots, A[n]$) mit $A[1] \leq A[2] \leq \dots \leq A[n]$.
- Schlüssel b

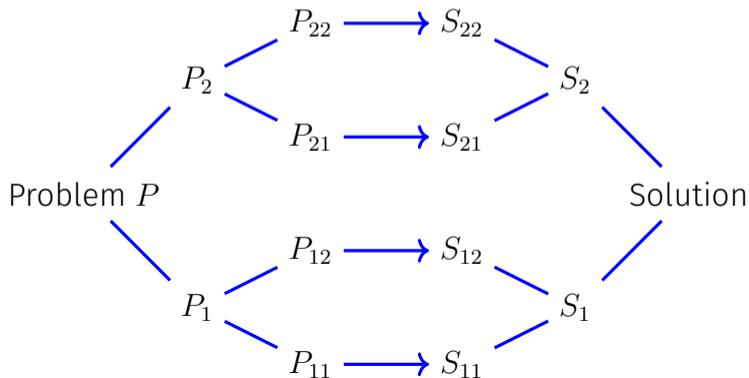
Gesucht: Index k , $1 \leq k \leq n$ mit $A[k] = b$ oder "nicht gefunden".

10	20	22	24	28	32	35	38	41	42
1	2	3	4	5	6	7	8	9	10

divide et impera

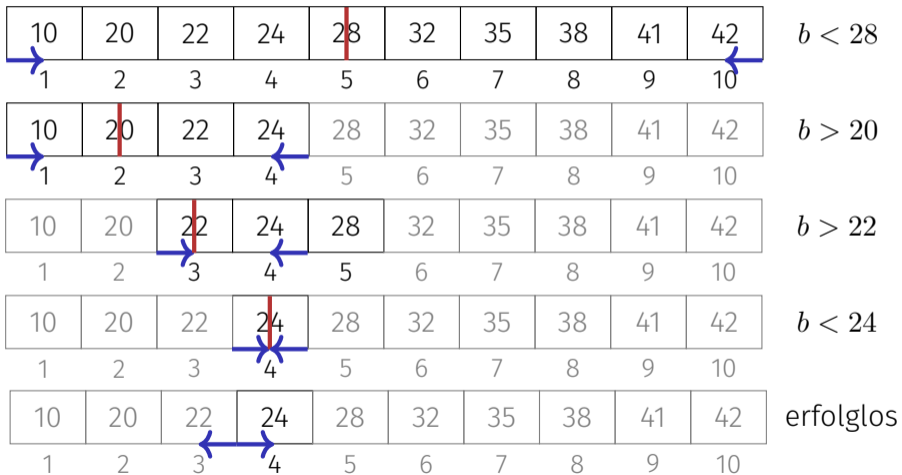
Teile und (be)herrsche (engl. divide and conquer)

Zerlege das Problem in Teilprobleme, deren Lösung zur vereinfachten Lösung des Gesamtproblems beitragen.



Divide and Conquer!

Suche $b = 23$.



Binärer Suchalgorithmus BSearch(A, l, r, b)

Input: Sortiertes Array A von n Schlüsseln. Schlüssel b . Bereichsgrenzen

$1 \leq l, r \leq n$ mit $l \leq r$ oder $l = r + 1$.

Output: Index $m \in [l, \dots, r + 1]$, so dass $A[i] \leq b$ für alle $l \leq i < m$ und

$A[i] \geq b$ für alle $m < i \leq r$.

$m \leftarrow \lfloor (l + r) / 2 \rfloor$

if $l > r$ **then** // erfolglose Suche

 | **return** l

else if $b = A[m]$ **then** // gefunden

 | **return** m

else if $b < A[m]$ **then** // Element liegt links

 | **return** BSearch($A, l, m - 1, b$)

else // $b > A[m]$: Element liegt rechts

 | **return** BSearch($A, m + 1, r, b$)

Analyse (schlechtester Fall)

Rekurrenz ($n = 2^k$)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

Teleskopieren:²

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c = \dots \\ &= T\left(\frac{n}{2^i}\right) + i \cdot c \\ &= T\left(\frac{n}{n}\right) + \log_2 n \cdot c = d + c \cdot \log_2 n \in \Theta(\log n) \end{aligned}$$

²Versuche eine geschlossene Form zu finden, indem die Rekurrenz, ausgehend von $T(n)$, wiederholt eingesetzt wird.

Resultat

Theorem 3

Der Algorithmus zur binären sortierten Suche benötigt $\Theta(\log n)$ Elementarschritte.

Iterativer binärer Suchalgorithmus

Input: Sortiertes Array A von n Schlüsseln. Schlüssel b .

Output: Index des gefundenen Elements. 0, wenn erfolglos.

$l \leftarrow 1; r \leftarrow n$

while $l \leq r$ **do**

$m \leftarrow \lfloor (l + r) / 2 \rfloor$

if $A[m] = b$ **then**

return m

else if $A[m] < b$ **then**

$l \leftarrow m + 1$

else

$r \leftarrow m - 1$

return *NotFound*;

7. Sortieren

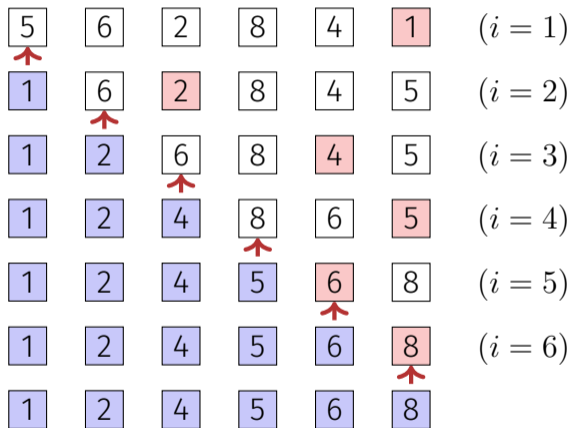
Einfache Sortierverfahren, Quicksort, Mergesort

Problemstellung

Eingabe: Ein Array $A = (A[1], \dots, A[n])$ der Länge n .

Ausgabe: Eine Permutation A' von A , die sortiert ist: $A'[i] \leq A'[j]$ für alle $1 \leq i \leq j \leq n$.

Sortieren durch Auswahl



- Auswahl des kleinsten Elementes durch Suche im unsortierten Teil $A[i..n]$ des Arrays.
- Tausche kleinstes Element an das erste Element des unsortierten Teiles.
- Unsortierter Teil wird ein Element kleiner ($i \rightarrow i + 1$). Wiederhole bis alles sortiert. ($i = n$)

Algorithmus: Sortieren durch Auswahl

Input: Array $A = (A[1], \dots, A[n])$, $n \geq 0$.

Output: Sortiertes Array A

for $i \leftarrow 1$ **to** $n - 1$ **do**

$p \leftarrow i$

for $j \leftarrow i + 1$ **to** n **do**

if $A[j] < A[p]$ **then**

$p \leftarrow j$;

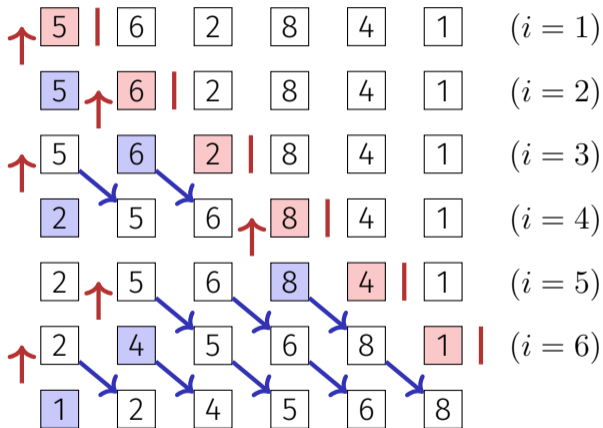
 swap($A[i], A[p]$)

Analyse

Anzahl Vergleiche im schlechtesten Fall: $\Theta(n^2)$.

Anzahl Vertauschungen im schlechtesten Fall: $n - 1 = \Theta(n)$

Sortieren durch Einfügen



- Iteratives Vorgehen:
 $i = 1 \dots n$
- Einfügeposition für Element i bestimmen.
- Element i einfügen, ggfs. Verschiebung nötig.

Sortieren durch Einfügen

Welchen Nachteil hat der Algorithmus im Vergleich zum Sortieren durch Auswahl?

Im schlechtesten Fall viele Elementverschiebungen.

Welchen Vorteil hat der Algorithmus im Vergleich zum Sortieren durch Auswahl?

Der Suchbereich (Einfügebereich) ist bereits sortiert. Konsequenz: binäre Suche möglich.

Algorithmus: Sortieren durch Einfügen

Input: Array $A = (A[1], \dots, A[n])$, $n \geq 0$.

Output: Sortiertes Array A

for $i \leftarrow 2$ **to** n **do**

$x \leftarrow A[i]$

$p \leftarrow \text{BinarySearch}(A, 1, i - 1, x)$; // Kleinstes $p \in [1, i]$ mit $A[p] \geq x$

for $j \leftarrow i - 1$ **downto** p **do**

$A[j + 1] \leftarrow A[j]$

$A[p] \leftarrow x$

7.1 Mergesort

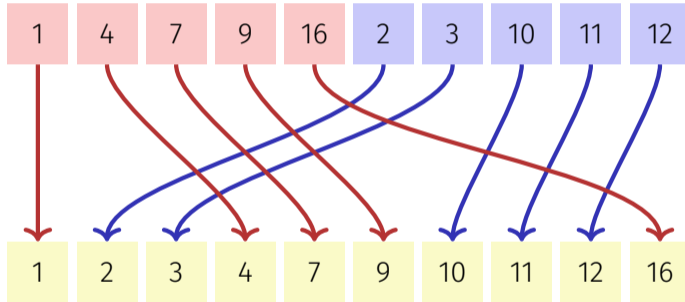
[Ottman/Widmayer, Kap. 2.4, Cormen et al, Kap. 2.3],

Mergesort (Sortieren durch Verschmelzen)

Divide and Conquer!

- Annahme: Zwei Hälften eines Arrays A bereits sortiert.
- Folgerung: Minimum von A kann mit 2 Vergleichen ermittelt werden.
- Iterativ: Füge die beiden vorsortierten Hälften von A zusammen in $\mathcal{O}(n)$.

Merge



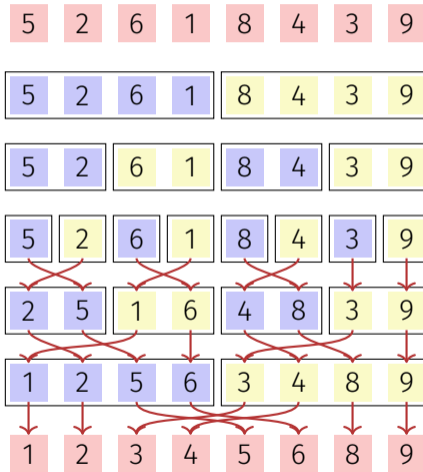
Algorithmus Merge(A, l, m, r)

Input: Array A der Länge n , Indizes $1 \leq l \leq m \leq r \leq n$.
 $A[l, \dots, m]$, $A[m + 1, \dots, r]$ sortiert

Output: $A[l, \dots, r]$ sortiert

```
1  $B \leftarrow$  new Array( $r - l + 1$ )
2  $i \leftarrow l$ ;  $j \leftarrow m + 1$ ;  $k \leftarrow 1$ 
3 while  $i \leq m$  and  $j \leq r$  do
4   if  $A[i] \leq A[j]$  then  $B[k] \leftarrow A[i]$ ;  $i \leftarrow i + 1$ 
5   else  $B[k] \leftarrow A[j]$ ;  $j \leftarrow j + 1$ 
6    $k \leftarrow k + 1$ ;
7 while  $i \leq m$  do  $B[k] \leftarrow A[i]$ ;  $i \leftarrow i + 1$ ;  $k \leftarrow k + 1$ 
8 while  $j \leq r$  do  $B[k] \leftarrow A[j]$ ;  $j \leftarrow j + 1$ ;  $k \leftarrow k + 1$ 
9 for  $k \leftarrow l$  to  $r$  do  $A[k] \leftarrow B[k - l + 1]$ 
```

Mergesort



Split

Split

Split

Merge

Merge

Merge

Algorithmus (Rekursives 2-Wege) Mergesort(A, l, r)

Input: Array A der Länge n . $1 \leq l \leq r \leq n$

Output: $A[l, \dots, r]$ sortiert.

if $l < r$ **then**

```
 $m \leftarrow \lfloor (l + r) / 2 \rfloor$  // Mittlere Position  
Mergesort( $A, l, m$ ) // Sortiere vordere Hälfte  
Mergesort( $A, m + 1, r$ ) // Sortiere hintere Hälfte  
Merge( $A, l, m, r$ ) // Verschmelzen der Teilfolgen
```

Analyse

Rekursionsgleichung für die Anzahl Vergleiche und Schlüsselbewegungen:

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(n) \in \Theta(n \log n)$$

Herleitung für $n = 2^k$

Sei $n = 2^k$, $k > 0$. Rekurrenz

$$T(n) = \begin{cases} d & \text{falls } n = 1 \\ 2T(n/2) + cn & \text{falls } n > 1 \end{cases}$$

Teleskopieren

$$\begin{aligned} T(n) &= 2T(n/2) + cn = 2(2T(n/4) + cn/2) + cn \\ &= 2(2(T(n/8) + cn/4) + cn/2) + cn = \dots \\ &= 2(2(\dots(2(2T(n/2^k) + cn/2^{k-1})\dots) + cn/2^2) + cn/2^1) + cn \\ &= 2^k T(1) + \underbrace{2^{k-1}cn/2^{k-1} + 2^{k-2}cn/2^{k-2} + \dots + 2^{k-k}cn/2^{k-k}}_{k\text{Terme}} \\ &= nd + cnk = nd + cn \log_2 n \in \Theta(n \log n). \end{aligned}$$

7.2 Quicksort

[Ottman/Widmayer, Kap. 2.2, Cormen et al, Kap. 7]

Quicksort

Was ist der Nachteil von Mergesort?

Benötigt zusätzlich $\Theta(n)$ Speicherplatz für das Verschmelzen.

Wie könnte man das Verschmelzen einsparen?

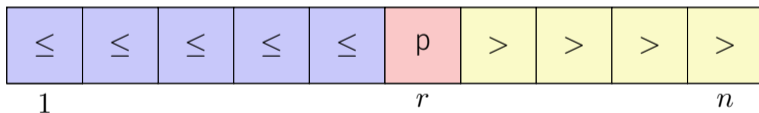
Sorge dafür, dass jedes Element im linken Teil kleiner ist als im rechten Teil.

Wie?

Pivotieren und Aufteilen!

Pivotieren

1. Wähle ein (beliebiges) Element p als Pivotelement
2. Teile A in zwei Teile auf: einen Teil L der Elemente mit $A[i] \leq p$ und einen Teil R der Elemente mit $A[i] > p$.
3. Quicksort: Rekursion auf Teilen L und R



Algorithmus Partition(A, l, r, p)

Input: Array A , welches den Pivot p in $A[l, \dots, r]$ mindestens einmal enthält.

Output: Array A partitioniert in $A[l, \dots, r]$ um p . Rückgabe der Position von p .

while $l \leq r$ **do**

while $A[l] < p$ **do**

$l \leftarrow l + 1$

while $A[r] > p$ **do**

$r \leftarrow r - 1$

 swap($A[l], A[r]$)

if $A[l] = A[r]$ **then**

$l \leftarrow l + 1$

return $l-1$

Algorithmus Quicksort(A, l, r)

Input: Array A der Länge n . $1 \leq l \leq r \leq n$.

Output: Array A , sortiert in $A[l, \dots, r]$.

if $l < r$ **then**

 Wähle Pivot $p \in A[l, \dots, r]$

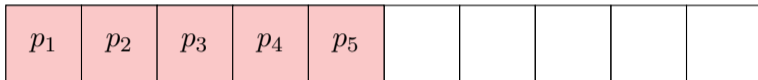
$k \leftarrow \text{Partition}(A, l, r, p)$

 Quicksort($A, l, k - 1$)

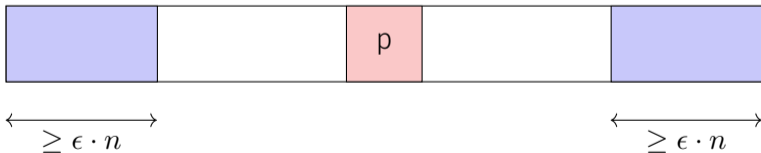
 Quicksort($A, k + 1, r$)

Wahl des Pivots

Das Minimum ist ein schlechter Pivot: worst Case $\Theta(n^2)$

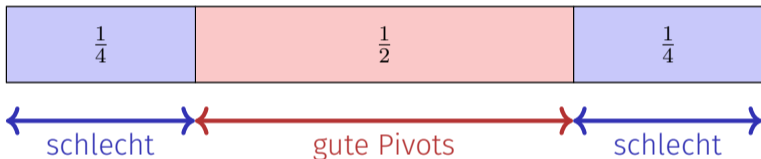


Ein guter Pivot hat linear viele Elemente auf beiden Seiten.



Wahl des Pivots?

Der Zufall hilft uns (Tony Hoare, 1961). Wähle in jedem Schritt einen zufälligen Pivot.



Wahrscheinlichkeit für guten Pivot nach einem Versuch: $\frac{1}{2} =: \rho$.

Wahrscheinlichkeit für guten Pivot nach k Versuchen: $(1 - \rho)^{k-1} \cdot \rho$.

Erwartete Anzahl Versuche³: $1/\rho = 2$

³Erwartungswert der geometrischen Verteilung:

Quicksort (willkürlicher Pivot)

2 4 5 6 8 3 7 9 1

2 1 3 6 8 5 7 9 4

1 2 3 4 5 8 7 9 6

1 2 3 4 5 6 7 9 8

1 2 3 4 5 6 7 8 9

1 2 3 4 5 6 7 8 9

Analyse: Anzahl Vergleiche

Schlechtester Fall. Pivotelement = Minimum oder Maximum; Anzahl Vergleiche:

$$T(n) = T(n - 1) + c \cdot n, T(1) = 0 \quad \Rightarrow \quad T(n) \in \Theta(n^2)$$

Analyse (Randomisiertes Quicksort)

Theorem 4

Im Mittel benötigt randomisiertes Quicksort $\mathcal{O}(n \cdot \log n)$ Vergleiche.

(ohne Beweis.)

Praktische Anmerkungen

- Für den Pivot wird in der Praxis oft der Median von drei Elementen genommen. Beispiel: $\text{Median3}(A[l], A[r], A[\lfloor l + r/2 \rfloor])$.