Pointer

= Zeiger = Verweise = Datenreferenzen

Ein Pointer ist ein Verweis/eine Referenz auf ein Zielobjekt/Zeigerziel/Target eines festgelegten Datentyps.

⦿→ □
⦿→ || Nullpointer

Ein Pointer hat zu Beginn der Programmausführung einen undefinierten Zustand, der nicht als solcher erkannt werden kann ⇒ Verwendung eines solchen Pointers kann große Probleme verursachen.

- Zeiger sind kein eigenständiger Typ, sondern nur mit dem Attribut `pointer` gekennzeichnet.
- Zeiger sind streng typisiert, d.h. man kann nur auf Objekte zeigen, deren Typ identisch mit dem Zeigertyp ist ⇒ keine Universalpointer
- Jedes beliebige Objekt vom passenden Objekttyp kann als Ziel eines Zeigers dieses Typs verwendet werden, wenn die Zielvariable das Attribut `target` trägt oder das Objekt ein dynamisches im Heap erzeugtes Objekt ist.
- Jede Pointer-Variable kann als Zeigerziel dienen (ohne `target`-Atribut)
- Implizit werden Pointer immer automatisch dereferenziert außer in den Anweisungen `nullify`, `allocate` und `deallocate`, der Pointer-Zuweisung `p => z` sowie in der `associated`-Abfragefunktion
- Pointer sind in Fortran in der Regel mehr als nur Adressen

Pointer-Kontext
… in dem Pointer nicht automatisch dereferenziert werden

- `nullfify(p1,p2,…)` → versetzt die Pointer in den definierten Zustand Null = nicht assoziiert
- `allocate(p1[Dimensionierung],p2[…],…[,STAT=ivar])` → legt Speicherblöcke im Heap für die Zielobjekte der Pointer an und setzt die Pointer als Referenzen auf ihren jeweiligen Speicherblock ⇒ alle Pointer im definierten Zustand assoziiert
- `deallocate(p1,p2,…[,STAT=ivar])` → gibt die Speicherblöcke, auf die die Pointer zeigen, frei und setzt die Pointer auf Null. Pointer muss assoziiert sein und ein ganzes Objekt sein, d.h. kein Substring, Subarray,… sein.
- Pointerzuweisung: `ptr => tgt` oder `ptr => ptr2`
- Abfragefunktion `associated`:
  - `associated(ptr)` → `true`, wenn auf ein Ziel gezeigt wird; `false`, wenn `ptr` auf Null zeigt
  - `associated(ptr, tgt)` → `true`, wenn `ptr` auf `tgt` zeigt, sonst `false`
  - `associated(ptr1,ptr2)` → `true`, falls beide Pointer denselben Zustand (nicht Null) haben, sonst `false`

Gefahren für den Hauptspeicher, insbesondere Heap
- Verwendung eines nicht definierten oder nicht gültigen Pointers in `deallocate`, `=>`, `associated`-Abfragen und normalen (nicht Pointer-) Kontext, d.h. in Expressions, in denen alle Pointer automatisch dereferenziert werden
- Dangling Pointer entstehen, wenn das Zeigerziel verloren geht, z.B. durch `deallocate` über anderen Pointer oder eines `allocatable`-Feldes oder wenn das Zielobjekt „out of scope“ geht, z.B. durch Verlassen seiner Prozedur
- Speicherleichen, Garbage, memory leaks: haben im Prinzip das ewige Leben im Heap, wenn keine Referenzen mehr auf ein Heap-Objekt existiert, über die man es freigeben könnte

Liste: lineare Anordnung von Objekten/Elementen desselben Typs
- Implementierungsebene: als verzeigerte lineare Struktur oder als eindimensionales Feld

  - 3 Attribute: linear vs. zyklisch, einfach verkettet vs. doppelt verkettet, endogen vs. exogen
- Konzeptionelle/abstrakte Ebene
  - i.a eine Liste mit gewissen Einfüge- und Löschoperationen
  - wenn nur an den beiden Enden der Liste solche Operationen nötig sind, spricht man von einer *Deque* = double-ended queue

Grundoperationen auf einer Liste `L` mit Element `e`

`init(L)`	Initialisierung einer Liste, Anfangszustand „leer“	
`empty(L)`	Abfragefunktion → `true`, falls `L` leer, sonst `false`	
`access_head(L,e)`	als Subroutine → e liefert den Wert des head-Elements	head = Beginn einer Liste
`val_head(L)`	als Abfragefunktion → Ergebnis ist Inhalt des head-Elements	
`access_head(L,e)`	als Subroutine → e liefert den Wert des tail-Elements	tail = Ende einer Liste
`val_head(L)`	als Abfragefunktion → Ergebnis ist Inhalt des tail-Elements	
`val_elem(L,p)`	liefert Inhalt/Wert des durch `p` referenzierten Elements	
insert		
`ins_head(L,e)`	Einfügen eines Elements `e` am Anfang der Liste `L`	anderer Name: `push`
`ins_tail(L,e)`	Einfügen eines Elements `e` am Ende der Liste `L`	anderer Name: `inject`
`ins_after(L,p,e)`	Einfügen eines Elements `e` nach dem von `p` referenzierten Elements	
`ins_before(L,p,e)`	Einfügen eines Elements `e` vor dem von `p` referenzierten Elements	
delete		
`del_head(L,e)`	Löschen eines Elements `e` am Anfang der Liste `L`	anderer Name: `pop`
`del_tail(L,e)`	Löschen eines Elements `e` am Ende der Liste `L`	anderer Name: `eject`
`del_after(L,p,e)`	Löschen eines Elements `e` nach dem von `p` referenzierten Elements	
`del_element(L,p,e)`	Löschen eines Elements `e` welches von p referenziert wird	
traverse: Traversieren (Durchlaufen aller Elemente) der Liste `L` und Ausführen einer Task `T` auf jedem Element		
`trav_forward(L,T[,p])`	vorwärts, optional ab dem von `p` referenzierten Elements	
`trav_backward(L,T[,p])`	rückwärts, optional ab dem von `p` referenzierten Elements	
find/search: Suchen eines Elements mit Inhalt `e`		

`find_forward(L,e[,p])`	vorwärts, optional ab dem von p referenzierten Elements	Ergebnis q ist die gelieferte Referenz (Pointer oder Index) auf das gefundene Element
`find_backward(L,e[,p])`	rückwärts, optional ab dem von p referenzierten Elements	

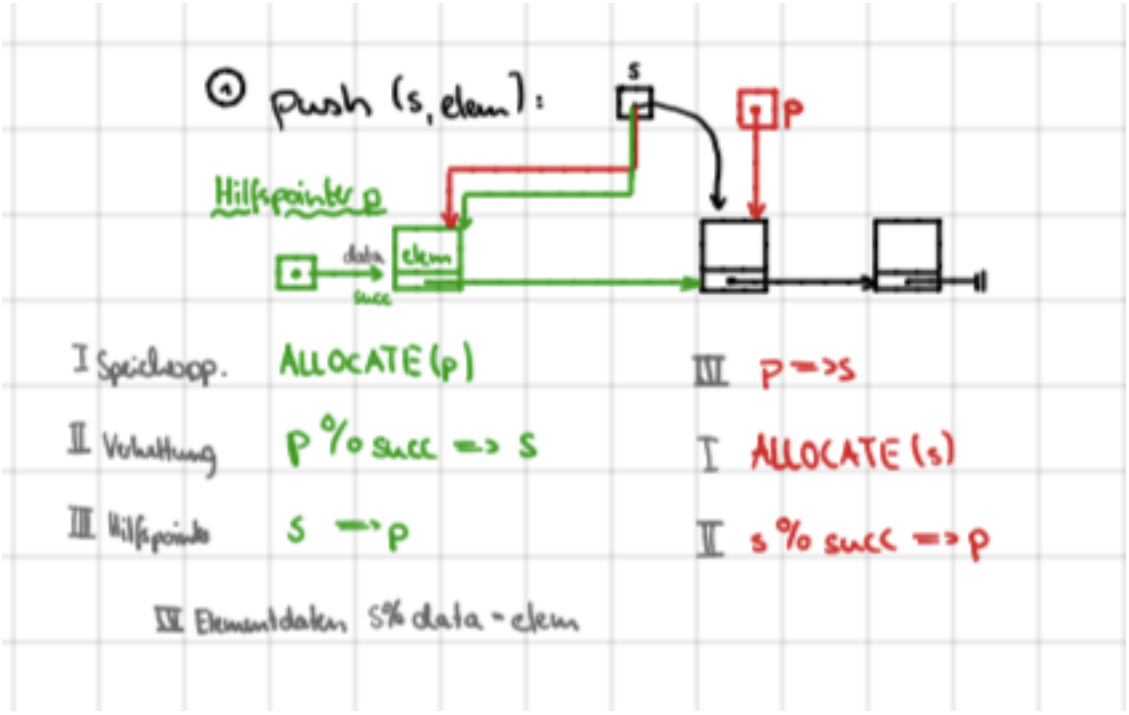
Grundoperationen einer Deque: `push, pop, inject, eject`

Der einfachste Fall ist der einer linearen, nicht zyklischen, einfach verketten, endogenen Liste mit `s` als head-Pointer.

	insert	delete
1	Speicher beschaffen	Speicher freigeben
2	interne Verzeigerung aktualisieren	
3	Hilfpointer verwenden, externen Zugriff aktualisieren	
4	Daten (Inhalt) kopieren	

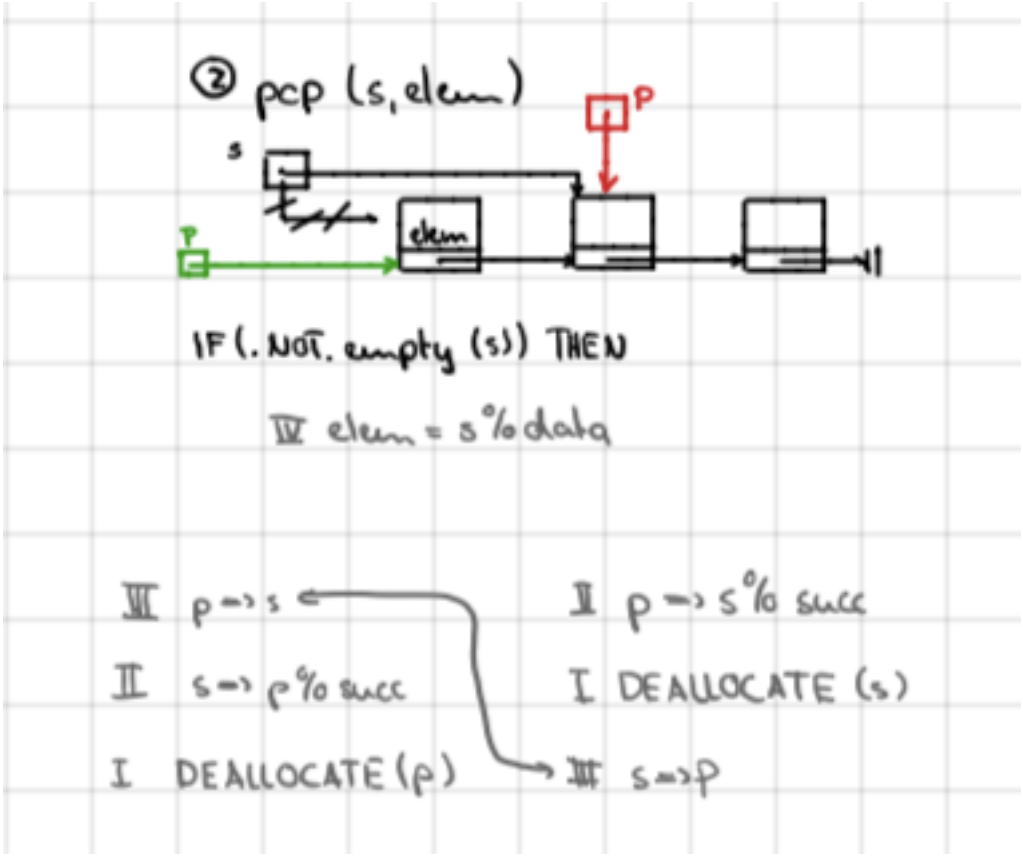
`push(s,elem)`

Speicheroperator	`allocate(p)`
Verkettung	`p%succ => s // succ ist Nachfolger`
Hilfpointer	`s => p`
Hilfpointer	`p => s`
Speicheroperator	`allocate(s)`
Verkettung	`s%succ => p`
Elementdaten	`s%data = elem`



`pop(s,elem)`

Sicherheitsabfrage	`if(.not. empty(L)) then`
Sicherung des Inhalts	`elem = s%data`
Übergabe des Startpointers	`p => s`
	`s => p%succ`
Speicherplatz freiräumen	`deallocate(p)`



`inject(t,elem)`

Speicheroperator	`allocate(p)`
Verkettung	`t%succ => p // succ ist Nachfolger`
	`nullify(p%succ)`
Außenzugriff	`t => p`
Elementdaten	`t%data = elem`

Außenzugriff	`p => t`
Speicheroperator	`allocate(t)`
Verkettung	`p%succ => t`
	`nullify(t%succ)`

`eject(t,elem)`: Problem! Vorgänger des tail-Elements kann nur gefunden werden, indem man die gesamte Liste durchläuft. ⇒ T(n) = O(n), d.h. Laufzeitkomplexität von Listenlänge abhängig.

Queue (= Warteschlange): sollte mit `pop` und `inject` implementiert werden
- FIFO-Prinzip (= firtst-in-first-out queue)
- LIFO-Prinzip (= last-in-first-out queue)
- Output-restricted-queue: deque mit `push`, `pop`, `inject`, ohne `eject`
- Input-restricted-queue: deque mit `pop`, `eject` und entweder `push` oder `inject`
- allgemeine deque: alle 4 Grundoperationen

Grundoperationen einer Queue
- `init`, `empty`, `enqueue` (inject am tail), `dequeue` (pop am head)
- Implementierung mit 1-dimensionalen Feld: `maxlengh`, `ixhead`, `ixtail`, `elems` (Pointer auf Feld)

init(n):
```
allocate(elems(0:n-1))
maxlengh = n
ixhead = 0
ixtail = n-1
```

empty = `mod(ixtail+1,n) == ixhead`

full = `mod(ixtail+2,n) == ixhead` `! ein Element bleibt ungenutzt`

inject:
```
ixtail = mod(ixtail+1,n)
```

pop:
```
ixhead = mod(ixhead+1,n)
```

Aufwand für Rechenoperationen

	einfach verkettet		**doppelt verkettet**	
Grundoperationen	**linear**	**zyklisch**	**linear**	**zyklisch**
`access_head`	c	c	c	c
`push`	c	c	c	c
`pop`	c	c	c	c
`access_tail` [mit tail-Pointer]	O(n) [c]	c	O(n) [c]	c
`inject`	O(n) [c]	c	O(n) [c]	c
`eject`	O(n)	O(n)	O(n) [c]	c
`ins_before`	O(n)	O(n)	c	c
`ins_after`	c	c	c	c
`de[_elem]`	O(n)	O(n)	c	c
`del_after`	c	c	c	c
`trav_forward`	c (pro Element)	c (pro Element)	c (pro Element)	c (pro Element)

	einfach verkettet		doppelt verkettet	
Grundoperationen	**linear**	**zyklisch**	**linear**	**zyklisch**
`trav_backward`	O(n) (pro Element)	O(n) (pro Element)	c (pro Element)	c (pro Element)

Bäume/Trees (und Rekursion)

Definition: Ein Baum ist entweder leer ode besteht aus einer endlichen Menge von Knoten mit einem speziell ausgezeichneten Wurzelknoten und einer endlichen Zahl von Teilbäumen.
⇒ rekursive Definition und rekursive Datenstruktur

⇒ rekursive Algorithmen zur Bearbeitung

Grad: Anzahl der Verzweigungen nach unten
Level: Anzahl der Ebenen, beginnend bei der Wurzel mit 0
Höhe: Weglänge zum weitest entfernten Knoten

Wechselbeziehung rek. Datenstruktur ⇔ rek. Algorithmus

- Die Ausführung rekursiver Algorithmen kann man als Baum darstellen, vgl. Fibonacci-Zahlen
- Die meisten Grundoperationen für Bäume sind rekursiv definiert.

Rechtsrekursion (tail recursion)	**Linksrekursion (head recursion)**
Problem $P_n = \begin{cases} T_0 & n = 0 \\ T_n P_{n-1} & n > 0 \end{cases}$ $\Rightarrow P_n \rightarrow T_n T_{n-1} \dots T_1 T_0 \Rightarrow$ leicht auflösbare Rekursion ⇒ Iteration	Problem $P_n = \begin{cases} T_0 & n = 0 \\ P_{n-1} T_n & n > 0 \end{cases}$ $\Rightarrow P_n \rightarrow T_0 T_1 \dots T_{n-1} T_n \Rightarrow$ nicht leicht auflösbare Rekursion ⇒ benötigt Stack In der Regel kann man nicht direkt $T_0 T_1 \dots T_n$ ausführen, da man die Speicherzustände nicht vorhersehen kann.

Allgemeine Rekursion

$$P_{n,i} = \begin{cases} T_0 & n = 0 \\ T_0 P_{n-1,1} T_1 P_{n-1,2} \dots T_{k-1} P_{n-1,k} T_k & n > 0 \end{cases} \quad 1 \leq i \leq k$$

wichtig: Auch hier genügt die Abarbeitung mit einem Stack!

Definition: Ein Binärbaum ist ein Baum mit max. Knotengrad 2.

- max. Anzahl Knoten: auf dem Level l: 2^l, auf ganzen Binärbaum: $N = \sum_{l=0}^{h} 2^l = 2^{h+1} - 1$
- minimale Höhe eines Binärbaums mit N Knoten ist $h_{min} = \lfloor \log_2 N \rfloor$

Binärer Suchbaum
Binärbaum, bei dem im linken Teilbaum eines Knotens nur „kleinere“ Elemente und im rechten Teilbaum nur „größere“ Elemente gespeichert sind. Dabei gibt es immer eine besondere Datenkomponente, die als Schlüssel (Key) dient und deren Werte eine vollständige Ordnung ermöglichen (Ordnungsrelation, typischerweise <).

Elementare Operationen auf Binärbaumen

- Traversieren:
  - Preorder: $P(B) = T(B)P(B_L)P(B_R)$

  - Inorder: $P(B) = P(B_L)T(B)P(B_R)$
  - Postorder: $P(B) = P(B_L)P(B_L)T(B)$
  - Levelorder: schichtenweises Durchlaufen von oben nach unten, links nach rechts
- Einfügen (und suchen) im Suchbaum:
  Beim Durchlaufen (Traversieren) in Inorder erhält man die in aufsteigender Schlüsselreihenfolge sortierten Elemente/Knoten(inhalte).
- Löschen eines Knotens mit gesuchtem Schlüsselwert im Suchbaum
  - Blatt löschen ist einfach (keine Teilbäume)
  - Knoten hat genau einen Teilbaum: listenartige Reparatur
  - Innerer Knoten mit 2 nichtleeren Teilbäumen: 2 Möglichkeiten
    - größeres Element im linken Teilbaum (des zu löschenden Knotens) suchen, dieses hat rechten Teilbaum ⇒ diesem Knoten durch seinen linken Teilbaum ersetzen, Inhalt dieses (ersetzten) Elements in den „zu löschenden“ Knoten kopieren, sodann dieses größere Element (d.h. seinen Knoten) mittels 1 oder 2 löschen (Speicher freigeben!)
    - kleines Element im rechten Teilbaum (des zu löschenden Knotens) suchen, dieses hat linken Teilbaum ⇒ diesem Knoten durch seinen rechten Teilbaum ersetzen, Inhalt dieses (ersetzten) Elements in den „zu löschenden“ Knoten kopieren, sodann dieses kleinere Element (d.h. seinen Knoten) mittels 1 oder 2 löschen (Speicher freigeben!)

Sentinel = Wachposten
Sentinel ist ein Knoten, der den zu suchenden Schlüssel enthält. Alle Nullpointer eines Trees zeigen auf den Sentinel. Das sorgt dafür, dass, wenn man einen Schlüssel sucht, nach links (bei kleiner) bzw. nach rechts (bei größer), ist der Wert gleich, muss man nur noch schauen, ob der gefundene Wert der Sentinel ist, dann ist der gesuchte Wert nicht enthalten, andernfalls schon.

Suchen und Sortieren

Lineares Suchen: in n Elementen (unsortiert): Aufwand zwischen 1 und n, also linear abhängig von Anzahl n der Elemente.

Binäres Suchen: in 1-dim. Feld L(1:n) durch jeweiliges Abfragen des Schlüsselwertes des „mittleren“ Elements im verbleibenden möglichen Indexbereich ⇒ logarithmischer Aufwand:

$T(n) = \mathcal{O}(\log_2 n)$. Voraussetzung ist, dass die Elemente in L nach Schlüsselwerten sortiert sind.

In-situ Sortierverfahren:
- alle n Elemente sind schon zu Beginn (in beliebiger Anordnung) im Feld L gespeichert
- die Elemente werden (bis auf eventuelle notwendige kurzzeitige Auslagerung eines Elements) immer in diesem Feld gehalten
- insbesondere wird nur eine sehr kleine Zahl zusätzlicher skalarer Varianten benötigt (und keine zusätzliche Datenstruktur mit $c \cdot n$ Elementen ($0 < c < 1$))
- Ziel: alle n Elemente liegen in sortierter Reihenfolge im Originalfeld L ⇒ ermöglicht binäre Suche

Stabiles Sortierverfahren: verändert die relative Ordnung von Elementen mit demselben Schlüsselwert nicht.

Externes Suchverfahren (d.h. nicht in-situ):
- Sortierung erfolgt nicht nur im Originalfeld
- benötigt typischerweise $\mathcal{O}(n)$ zusätzlichen Speicherplatz

Mikroschritt/Elementaroperation: besteht in der Regel aus 1 Vergleich von 2 Schlüsselwerten und 1 Kopier- oder Tauschoperation

Makroschritt/Durchlauf: besteht aus $\mathcal{O}(n)$ Mikroschritten, z.B. Durchlauf durch alle noch zu sortierenden Elemente

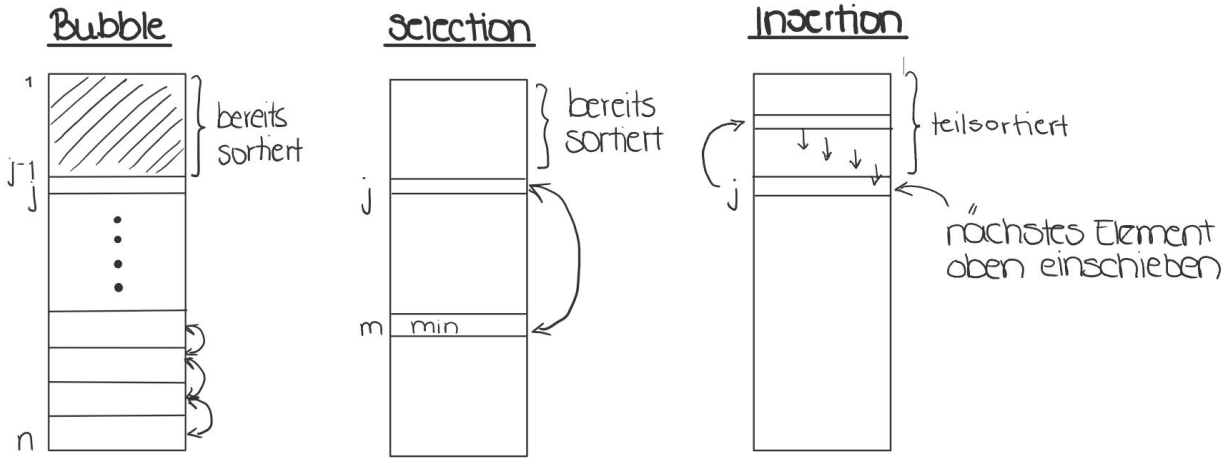
Komplexität $T(n)$:
- $\mathcal{O}(g(n)) = \{f(n) : \exists c > 0, n_0 \in \mathbb{N}_0 \mid 0 \leq f(n) \leq c \cdot g(n) \quad \forall n \geq n_0\}$ (Obergrenze)
- $\Omega(g(n)) = \{f(n) : \exists c > 0, n_0 \in \mathbb{N}_0 \mid 0 \leq c \cdot g(n) \leq f(n) \quad \forall n \geq n_0\}$ (Untergrenze)
- $\Theta(g(n)) = \{f(n) : \exists c_1, c_2 > 0, n_0 \in \mathbb{N}_0 \mid 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \forall n \geq n_0\}$ (Sandwich)

Beispiel: $T(n) = \mathcal{O}(n^2) = \mathcal{O}(n^2 \cdot \log n) = \mathcal{O}(n^2 \cdot \sqrt{n}) = \mathcal{O}(n^3) = \ldots = \mathcal{O}(2^n) = \ldots$

drei verschiedene Komplexitätsangaben werden für Algorithmen gemacht:
- *worst case*: $\mathcal{O}$
- *average case*: $\mathcal{O}$ oder Ω
- *best case*: Ω (z.B. jeder Sortieralgorithmus ist $\Omega(n)$)
- bei worst und average case: evtl. auch Θ

Indexmenge I={1,2,...,n}
Allgemeine Annahme: Sortiert wird immer in einem 1-dimensionalen Feld A mit Indexmenge I mit der Relation $\leq$ (oder <) bezüglich eines Schlüssels (=key) ⇒ aufsteigende Reihenfolge

Es gibt mindestens 2 Möglichkeiten, Datenelemente im Feld A zu sortieren:
1. durch Bewegen der Datenelemente incl. key [direktes Sortieren]
2. durch Erzeugen einer Sortierpermutation σ der Indizes (aus I), wobei nur die Indizes (in einem eigenen Feld) und nicht die Datenelemente bewegt werden [indirektes Sortieren]



Im sortierten Zustand gilt für alle i,j aus I:
für 1. (direktes Sortieren): i < j ⇒ A(i)[%key] <= (<) A(j)[%key}

für 2. (indirektes Sortieren): i < j ⇒ A($\sigma(i)$)[%key] <= (<) A($\sigma(j)$)[%key]

Definition: Eine *Sortierpermutation* σ einer "Liste" A (1-dimensionales Feld) auf einer Indexmenge I={1,...,n} ist eine Permutation von I, d.h. $[\sigma(1), \sigma(2), \ldots, \sigma(n)]$ mit $\sigma(i) \neq \sigma(j)$ für alle $i \neq j \Rightarrow \{\sigma(1), \sigma(2), \ldots, \sigma(n)\} = I$, für die die Sortierbedingung (2) gilt.

3 einfache Sortieralgorithmen

Sortierverfahren	# Vergleiche	# Kopier-/Tauschoperationen
Bubblesort	$\sum_{j=1}^{n-1}(n-j) = \sum_{j=1}^{n-1}(j) = n*(n-1)/2 = \mathcal{O}(n^2) = \Omega(n)$	$\leq 1/2 \cdot n \cdot (n-1) = \mathcal{O}(n^2)$ Tauschoperationen
Selection Sort	$\sum_{j=1}^{n-1}(n-j) = n*(n-1)/2 = \Theta(n^2)$	$\leq n-1 = \mathcal{O}(n)$ Tauschoperationen
Insertion Sort	$\sum_{j=1}^{n-1}(j) = n \cdot (n-1)/2 = \mathcal{O}(n^2)$	$\leq \sum_{j=1}^{n-1} = 1/2n(n-1) = \mathcal{O}(n^2)$
(mit binärer Suche im teilsortierten oberen Teil)	$\sum_{j=1}^{n-1} \log_2 j = \mathcal{O}(n \cdot \log_2 n)$	hier bleibt alles gleich !

allgemein:

- best case: 0 Bewegungen/Kopieren/Tauschvorgänge, $\Omega(n)$ Vergleiche
- worst case: $\mathcal{O}(n^2)$ Vergleiche oder Kopier-/Tauschvorgänge

	Vergleiche	Tauschoperationen
Bubble **stabil**	$\mathcal{O}(n^2)$ (ohne Abbruchbedingung: $\Theta(n^2)$, mit Abbruchbedingung: $\Omega(n)$)	$\mathcal{O}(n^2)$
Selection **nicht stabil**	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Insertion **stabil**	ohne binäre Suche: $\mathcal{O}(n^2)$ mit binärer Suche: $\Theta(n \cdot \log n)$	$\mathcal{O}(n^2)$

$T(n) = \mathcal{O}(n^2)$ für alle 3 einfachen Sortierverfahren

Annahme: Elemente mit paarweise verschiedenen Schlüsselwerten
Satz: Sortierverfahren, die auf dem Schlüsselvergleich (<) von jeweils 2 Elementen (und einer eventuell notwendigen Tauschoperation) beruhen, benötigen im worst case mindestens $\Omega(n \cdot \log n)$ Vergleiche.
Beweis: binärer Entscheidungsbaum der Höhe h zum Sortieren von n Elementen, da jeder Schlüsselvergleich eine binäre Entscheidung liefert. Es gibt $n!$ Permutationen der n verschiedenen Schlüsselwerte, also $n!$ verschiedene Sortierfolgen, d.h. $n!$ Entscheidungspfade. ⇒ binärer

Entscheidungsbaum benötigt mindestens $n!$ Blätter, um alle Anfangszustände in den einen sortierten zu überführen. Ein Binärbaum der Höhe h hat $\leq 2^h$ Blätter. Also muss gelten:

$$n! \leq 2^h$$
$$h \geq \log_2(n!)$$

Stirling: $n! = \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n \left(1 + \mathcal{O}\left(\frac{1}{n}\right)\right)$

$$n! > \left(\frac{n}{e}\right)^n$$

$$h \geq \log_2 \left(\frac{n}{e}\right)^n = n(\log_2 n - \log_2 e) = \Theta(n \cdot \log_2 n)$$

$\Rightarrow$ min. $\Omega(n \cdot \log_2 n)$ Vergleiche im worst case

---

Quicksort

- C.A.R. Hoare (Sir Charles Antony Richard Hoare)
- Idee: divide-and-conquer (teile und herrsche) → Rekursion
- rekursiver Algorithmus zum Sortieren einer „Liste“ L im Indexbereich $a : e$ [in situ]

1. Wähle ein beliebiges Element aus L, dieses habe key W (sog. Pivotelement), ideal wäre, wenn W der Median aller keys wäre, worst case: Extremum
2. Bilde Partition $L_L \,|\, L_R$ der Liste L mit :
   - alle Elemente von L_L haben keys $\leq W$
   - alle Elemente von L_R haben keys $\geq W$
3. Sortieren der beiden Listen mittels rekursiven Aufruf von Quicksort

Im worst case, d.h. das Pivotelement ist ein Extremum, wird immer nur 1 Element abgespalten.
$T(n) = \mathcal{O}(n^2)$
average case: $T(n) = \mathcal{O}(n \cdot \log_2 n)$, weniger als 5-mal so teuer wie im best case
best case: $T(n) = \Omega(n \cdot \log_2 n)$

Eigenschaften
- in situ
- nicht stabil
- hat nur sehr einfache Operationen
- für große n im Durchschnitt sehr schnell
- pro Durchlauf $\mathcal{O}(n)$
- für kleine n eher schlecht

Mergesort (2-Wege)

Mergesort(L): *rekursiv*
- falls L leer oder nur 1 Element enthält → ok, return
- divide: Teile L in 2 möglichst gleich lange Teillisten L_1 und L_2 und mache darauf rekursive Aufrufe Mergesort(L_1) und Mergesort(L_2)
- conquer: Merge/Verschmelzen von L_1 und L_2

- Mergesort ist nicht in situ!
- $T(n) = \mathcal{O}(n \cdot \log_2 n)$
- Alle Lese- und Schreiboperationen in Listen sind streng sequenziell.

iterativ

verwendet 4 Listen L_1, L_2, L_3, L_4

0. Init: Teile L in 2 möglichst gleich große Teillisten L_1 und L_2
1. Erzeuge 2 Listen sortierter Paare, L_3 und L_4, indem positionell sich entsprechende Elemente von L_1 und L_2 jeweils zu einem sortierten Paar gemacht und in die zuletzt nicht benutzte Liste L_3 bzw. L_4 (immer abwechselnd) geschrieben wird
2. Erzeuge 2 Listen sortierter Quadrupel in L_1 und L_2
3. Erzeuge 2 Listen sortierter Oktupel in L_3 und L_4
4. …

- $T(n) = \mathcal{O}(n \cdot \log_2 n)$, tatsächlich ist die Zeit $T(n) = \Theta(n \cdot \log_2 n)$ immer dieselbe, egal ob best- oder worst case

---

Mehrwege (k-Wege) Mergesort

- rekursiv: Daten in jeder rekursiven Aufrufebene in k ähnlich große Listen aufteilen
- mit $2k$ Listen/Dateien arbeiten: k Input-Listen und k Output-Listen
- Bemerkungen:
  - Für Mergeschritt wird ein k-elementiger Vektor von Schlüsselwerten benötigt, um die jeweils aktuellen Kopfelemente der k zu verschmelzenden Listen sortiert zu speichern
  - Anzahl Durchläufe reduziert sich gegenüber 2-Wege-Mergesort von $\lceil \log_2 n \rceil$ auf $\lceil \log_k n \rceil$, also um den Faktor $\frac{1}{\log_2 k} = \log_k 2$, z.B. bei $k = 1024 = 2^{10}$ Teillisten auf $\frac{1}{10}$. Sehr große Datenmengen können auf Dateien reduziert werden!

Makroschritt	produziert	1. k-Tupel sortieren	Anzahl Elemente	Aufwand Insertion-Schritt	Anzahl Tupel	Aufwand
1	k-Tupel: $\mathcal{O}($	$(k \log_2 k +$	$0 \cdot$	$\mathcal{O}(k))$	$. \frac{n}{k})$	$\mathcal{O}(n \log_2 k)$
2	k^2-Tupel: $\mathcal{O}($	$(k \log_2 k +$	$k(k-1) \cdot$	$\mathcal{O}(k))$	$. \frac{n}{k^2})$	$\mathcal{O}(kn)$
3	k^3-Tupel: $\mathcal{O}($	$(k \log_2 k +$	$(k^3 - k) \cdot$	$\mathcal{O}(k))$	$. \frac{n}{k^3})$	$\mathcal{O}(kn)$
…						
						$\Sigma = \mathcal{O}(kn \log_k n)$

Heapsort

Heap: Ein binärer Heap ist ein vollständiger Binärbaum mit der sogenannten Heap-Eigenschaft. Ein vollständiger Binärbaum ht alle Schichten ab der Wurzel voll besetzt bis auf (evtl.) die letzte, die von links nach rechts bis zum „letzten“ Knoten besetzt ist.

Die Vollständigkeit garantiert, dass ein eindimensionales Feld `A(1:n)` mit den Elementen des Heaps in Levelorder abgespeichert keine Lücken aufweist.
Außerdem gilt:
- Left(i) := 2*i !Index des linken Kindknotens des Knotens mit Index i
- Right(i) := 2*i+1 !Index des rechten Kindknotens des Knotens mit Index i
- Partent(i) := i/2 !$\lfloor \frac{i}{2} \rfloor$ ist Index des Elternknotens

⇒ Dualität eines Heaps und eines vollständigen Binärbaums; außerdem Höhe $h = \Theta(\log_2 n)$

Zusätzliche Heap-Eigenschaft: $A_{\text{Parent}(i)} \geq A_i$, d.h. Schlüsselwert des Elternelements ≥ Schlüsselwerte der beiden Kindknoten.

Frage: Wie kann aus einem beliebigen besetzten Feld `A` ein Heap gemacht werden?

```
n = size(A) !zu Beginn
```

`Heapify(A,i)` ≡ `Downheap(A,i)`:

```
r = Rigth(i)
l = Left(i)
maxix = i
if ( l ≤ size and A_l > A_i ) then
     maxix = l
end if
if ( r ≤ size and A_r > A_maxix ) then
     maxix = r
end if
if ( maxix ≠ i ) then
     TAUSCHE ( A_i , A_maxix )
     HEAPIFY ( A , maxix )
end if
```

Zeitaufwand für `Heapify(A,i)` ist proportional zur Höhe des Knotens mit Index i:
$T_{\text{Heapify}} = \mathcal{O}(h)$ mit h = `height(A,i)`, wobei $h = \mathcal{O}(\log_2 n)$

```
BuildHeap(A):
size = n
do i=⌊n/2⌋,1,-1
     Heapify(A,i)
end do
```

Komplexitätsanalyse (naive Komplexitätsschranke):

$$T_{\text{BuildHeap}}(n) = \mathcal{O}(\frac{n}{2} \cdot \log_2 n) = \mathcal{O}(n \cdot \log_2 n)$$

Bessere Analyse mittels Höhe der Knoten:

In einem Heap haben höchstens $\lceil \frac{n}{2^{n+1}} \rceil$ Knoten die Höhe h.

⇒ Gesamtaufwand für BuildHeap:

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil \cdot \mathcal{O}(h) = \mathcal{O}\left(n \cdot \sum_{h=0}^{\lfloor \log_2 n \rfloor} h \cdot \left(\frac{1}{2}\right)^h \right)$$

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$$

$0 < x < 1$: $\sum_{k=0}^{\infty} k \cdot x^k = \frac{1}{1-x} = (1-x)^{-1}$

Differenzieren: $\sum_{k=1}^{\infty} k \cdot x^{k-1} = \frac{1}{(1-x)^2}$

$x = \frac{1}{2}$: $\sum_{k=1}^{\infty} k \cdot \left(\frac{1}{2}\right) = \frac{\frac{1}{2}}{\left(\frac{1}{2}\right)^2} = 2 \Rightarrow T(n) = \mathcal{O}(n)$

Heapsort(A):

```
BuildHeap(A)                    !O(n)
do i = n, 2, -1                 !n-1 Iterationen
      Tausche(A_i, A_1)         !O(1)
      size = size - 1           !O(1)
      Heapify(A,1)              !O(log2 n)
end do
```

$\Rightarrow T_{\text{HeapSort}}(n) = \mathcal{O}(n) + (n-1)\mathcal{O}(\log_2 n) = \mathcal{O}(n \cdot \log_2 n)$

Priority Queue = Priorisierte Warteschlange

Art von Warteschlange, die nicht nach dem FIFO-Prinzip arbeitet, sondern die Elemente entsprechend ihrer Priorität (ein spezieller Key) behandelt

- Heap, implementiert mit 1-dimensionalen Feld

Grundoperationen:

- `init(A)` → leeres Feld A
- `empty(A)` → ist Feld leer?
- `HeapInsert(A, key)` → neues Element mit `key` wird eingefügt in `A`
- `HeapExtractMax(A, key)` → `key` des Wurzelknotens wird in 2. Parameter (oder als Funktionswert) zurückgegeben und dieses Element aus `A` herausgenommen
- `HeapUpdate(A, key, max)` → Kombination aus `ExtractMax` und `Insert`: liefert in `max` (oder als Funktionswert) den Key der Wurzel und ersetzt diesen Wert durch `key`
- `Maximum(A)` → Funktion, liefert den `key` der Wurzel

HeapExtractMax(A, max)

```
if (size ≤ 0) error(„empty heap")
max = A_1
A_1 = A_size
size = size - 1
Heapify(A,1)
```

$T(n) = \mathcal{O}(\log_2 n)$

HeapUpdate(A, key, max)

```
if (size ≤ 0) error(„empty heap")
max = A_1
A_1 = key
Heapify(A,1)
```

$T(n) = \mathcal{O}(\log_2 n)$

HeapInsert(A, key)

```
size = size + 1
```

```
i = size
do while (i > 1 and A_Parent(i) < key)
      A_i = A_Parent(i)
      i = Parent(i)
end do
A_i = key
```
$T(n) = \mathcal{O}(\log_2 n)$

```
BuildHeap*(A):
size = 1
do i=2, n
      HeapInsert(A,A_i)
end do
```
$T(n) = \mathcal{O}(\log_2 n)$ schlechter als $T(n) = \mathcal{O}(n)$

Counting Sort

Annahme: Alle möglichen Schlüsselwerte sind aus der Menge $S = \{0,1,...,k-1\}$. Wenn $k = \mathcal{O}(n)$, dann können wir mit Counting Sort in der Zeit $T(n) = \mathcal{O}(n)$ sortieren. Benötigt werden 3 1-dimensionale Felder:

- `A(1:n)` Originaldaten/-schlüsselwerte
- `B(1:n)` für sortierte Werte
- `C(0:k-1)` Feld von Zählern

```
CountingSort(A,B,k)
do i = 0, k-1
      C(i) = 0
end do

do j = 1, n
      C(A(j)) = C(A(j)) + 1
end do

do i = 1, k-1
      C(i) = C(i) + C(i-1)
end do

do j = n, 1, -1
      B(C(A(j))) = A(j)
      C(A(j)) = C(A(j)) - 1
end do
```

Radix-/Distribution Sort

Annahme: Der Schlüssel z lässt sich als Zahl mit d Ziffern zur Basis k, d.h. mit Ziffern $\in \{0,1,...,k-1\}$ schreiben: $z = [z_{d-1}z_{d-2}\dots z_1 z_0] = \sum_{i=0}^{d-1} z_i k^i$

```
RadixSort(A[,B],k)
do i = 0, d-1
      CountingSort(A,B,i,k)
end do
```

$T(n) = \Theta(d(n+k))$
- Falls $k = \mathcal{O}(n)$: $\Theta(dn)$
- Falls d relativ klein: $\Theta(n)$

Rekursion, Iteration, Komplexität

$T(n)$ - Zeitkomplexität
$S(n)$ - Speicherkomplexität

Fakultät

- rekursiv: $T(n) = \Theta(n)$ $S(n) = \Theta(n)$
- iterativ: $T(n) = \Theta(n)$ $S(n) = \Theta(1)$

Reverse String

- rekursiv: $T(n) = \Theta(n)$ $S(n) = \Theta(n)$
- iterativ: $T(n) = \Theta(n)$ $S(n) = \Theta(n)$

Primzahlen(tabelle)

- rekursiv: $T(n) = \mathcal{O}(\sqrt{n})$ $S(n) = ?$
- iterativ: $T(n) = \mathcal{O}(\sqrt{n})$ $S(n) = \Theta(1)$

Fibonacci-Zahlen

- rekursiv: $T(n) = \Theta(\Phi^n)$ $S(n) = \mathcal{O}(2^n)$
- iterativ: $T(n) = \Theta(n)$ $S(n) = \Theta(1)$
- direkt: $F_n = \frac{\Phi^n - (1-\Phi^n)}{\sqrt{5}}$

Ganzzahliges Potenzieren

- rekursiv: $T(n) = \Theta(\log_2 n)$ $S(n) = \Theta(\log_2 n)$
- iterativ: $T(n) = \Theta(\log_2 n)$ $S(n) = \Theta(1)$
- naiv: $T(n) = \Theta(n)$ $S_{iter}(n) = \Theta(1)$ $S_{rek}(n) = \Theta(n)$
- direkt: $x^n = e^{\ln x^n} = e^{n \cdot \ln x}$ → Funktionen e und $\ln$ zu benutzen ist teurer als iterativ

---

Größter gemeinsamer Teiler

euklidischer Algorithmus, f Fibonacci-Zahl:
$\text{ggT}(f_{k+1}, f_k) = \text{ggT}(f_k, f_{k+1} \bmod f_k) = \text{ggT}(f_k, f_{k-1})$

- rekursiv: $T(n) = \mathcal{O}(\log_\Phi b) = \mathcal{O}(\log_\Phi(\min\{a,b\}))$
- iterativ: $T(n) = \mathcal{O}(\log_\Phi b)$

Binomialkoeffizient

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

- rekursiv: $T(n) = \mathcal{O}(c^n) = \mathcal{O}(2^n), 1 < c < 2 \qquad S(n) = \mathcal{O}(n)$
- iterativ: $T(n) = \mathcal{O}(n^2) \qquad S(n) = \Theta(n)$
- direkt: $\binom{n}{k} = \frac{n}{1} \cdot \frac{n-1}{2} \cdot \frac{n-2}{3} \cdot \ldots \cdot \frac{n-k+1}{k}$

  $T(n) = \Theta(k) = \mathcal{O}(n) \qquad S(n) = \Theta(1)$

Komische Funktion (Lothar Collatz 1937)

```
komifun(p,n)
      do while (n /= 1)
            if(mod(n,2) == 1) then
                  n = p * n + 1
            else
                  n = n/2
            end if
      end do
end komifun
```

Für $p = 5$ ergibt sich:

n	**komifun(5,n)**
1	1
2	2 → 1
3	16 → 8 → 4 → 2 → 1
4	4 → 2 → 1
5	26 → 13 → 66 → 33 → 166 → 83 → 416 → 208 → 104 → 52 → 26 → …

⇒ nicht immer berechenbar!

Für $p = 3$ (originales Collatz-Problem) ergibt sich

n	**komifun(3,n)**
1	1
2	2 → 1
3	10 → 5 → 16 → 8 → 4 → 2 → 1
4	4 → 2 → 1
5	16 → 8 → 4 → 2 → 1

⇒ Berechenbarkeit? (bis $n < 2^{61}$ gezeigt)

Multiplikation zweier n-stelliger Zahlen x und y

- iterativ: $T(n) = \Theta(n^2)$
  allgemein: $T(x \cdot y) = \Theta(\log_b x \cdot \log_b y) = \Theta(m \cdot n)$ mit $\text{digits}(x) = m$ und $\text{digits}(y) = n$
- rekursiv: Idee: rekursive Halbierung der Zifferngruppen
  $T(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1,525})$

Von 1971 - 2007: Schönhage-Strassen-Algorithmus (diskrete Fourier-Transformation), schnellster Multiplikationsalgorithmus, $T(n) = \Theta(n \cdot \log n \cdot \log(\log n))$

Matrizenmultiplikation

- iterativ: $T(n) = \Theta(n^3)$
- rekursiv: $T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2,8075})$ (Strassen)
  $T(n) = \mathcal{O}(n^{2,376})$ beste obere Schranke bis heute

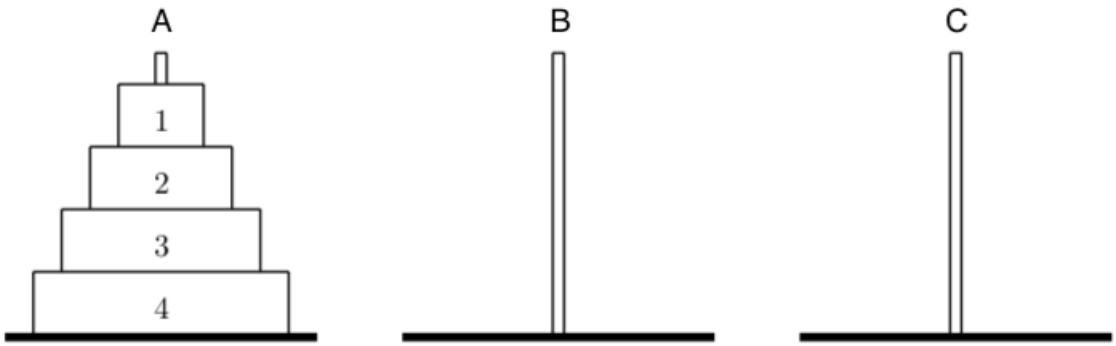
---

Zyklische Zahl

Finde $z_1, \ldots, z_n$ mit: $5 \cdot [z_1 z_2 \ldots z_n 5] = [5 z_1 z_2 \ldots z_n]$

- einfaches Durchprobieren aller Zahlen bis zu einer Zahl $n \in \mathbb{N}$ erzeugt Aufwand $T(n) = \mathcal{O}(n \cdot \log_2 n)$
- mit systematischer Berechnung der Ziffern von hinten nach vorne: $T(n) = \mathcal{O}(\log_2 n)$, wenn n die gesuchte Zahl ist
- $n = [z_1 \ldots z_n] = 102040816326530612244897959183673469387 75$

Türme von Hanoi



- rekursiv: $T(n) = 2^n - 1$
  `Move(A,C,n)` ⇒ `Move(A,B,n-1), Move(A,C,1), Move(B,C,n-1)`
- Aufwand wird für iterativen Algorithmus nicht kleiner

	Schritt i	**Scheibe** d	**Quelle** s	**Ziel** t
$n = 1$	1	1	A	C
$n = 2$	1	1	A	B
	2	2	A	C
	3	1	B	C
$n = 3$	1	1	A	C
	2	2	A	B
	3	1	C	B
	4	3	A	C
	5	1	B	A
	6	2	B	C
	7	1	A	C

Im i-ten Schritt:
- Scheibe k wird bewegt, mit `MOD(i, 2**(k-1)) = 0`, aber `MOD(i, 2**k) != 0`
- Scheibe k wird zum ersten Mal im Schritt 2^{k-1} und dann alle 2^k Schritte bewegt
- Die größte Scheibe n wird immer genau einmal im Schritt 2^{n-1} von A nach C bewegt, d.h. rückwärts durch die Positionsindizes, die nächstkleinere Scheibe $n - 1$ zweimal vorwärts A → B → C, die nächstkleinere $n - 2$ viermal rückwärts: A → C → B → A → C, …

Kochkurve

- rekursiv: $T = \mathcal{O}(4^n)$
- iterativ: schwierig

---

n-Damen-Problem

n = 1: triviale Lösung
n = 2: keine Lösung
n = 3: keine Lösung
n = 4: bis auf Symmetrien 1 Lösung

	D		
			D
D			

		D	

- primitiv: n Damen beliebig auf n^2 Felder platzieren: $\binom{n^2}{n}$ Möglichkeiten
- in jeder Spalte genau 1 Dame: $n^n < \binom{n^2}{n}$ Möglichkeiten
- zusätzlich bereits belegte Zeilen vermeiden: $n!$ Möglichkeiten
- Wenn auch Diagonalen berücksichtig werden sollen, gibt es viel weniger Möglichkeiten. Da eine systematische Planung vorausschauend alle bedrohten Felder für weitere Platzierungen kennen müsste, wird zur Lösung Backtracking verwendet.

Legepuzzle

2-dimensional mit n Teilen

- rekursiv naiv: eine Seite eines Teils mit jeweils allen offenen Seiten vergleichen
  $T(n) = \mathcal{O}(n^2)$
- iterativ mit k Teilmengen (z.B. Farbgruppen) mit $\approx \frac{n}{k}$ Teilen, wobei $k \leq \frac{n}{k}$, also z.B.
  $k = \sqrt{n} = \frac{n}{k}$: $T_k(n) = \mathcal{O}(n + \frac{n^2}{k} + k^2) = \mathcal{O}(n^{1.5})$

Ackermann-Funktion

$$A(i,j) = \begin{cases} 2^j & i = 1 \\ A(i-1,2) & j = 1 \\ A(i-1,A(i,j-1)) & i \geq 2, j \geq 2 \end{cases}$$

i \ j	1	2	3	4	5	6	7	8
1	2	4	8	16	32	64	128	256
2	4	$2^{2^2} = 16$	$2^{2^{2^2}} = 65536$	2^{65536}				
3	$2^{2^2} = 16$	$A(2,A(3,1)) = A(2,16)$						
4	$A(2,A(3,1)) = A(2,16)$	$A(3,A(4,1))$						

- Ackermann Funktion ist nicht primitiv rekursiv

Knuth Up-Arrow Notation

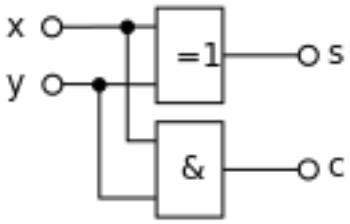
- $a \uparrow b = a^b = \underbrace{a \cdot \ldots \cdot a}_{b}$
- $a \uparrow\uparrow b = a \uparrow^2 b = {}^b a = \underbrace{a \uparrow \ldots \uparrow a}_{b}$

- $a \uparrow\uparrow\uparrow b = a \uparrow^3 b = \underbrace{a \uparrow\uparrow a \uparrow\uparrow \ldots \uparrow\uparrow a}_{b}$
- $a \underbrace{\uparrow \ldots \uparrow}_{n} b = \underbrace{a \underbrace{\uparrow \ldots \uparrow}_{n-1} a \underbrace{\uparrow \ldots \uparrow}_{n-1} \ldots \underbrace{\uparrow \ldots \uparrow}_{n-1} a}_{b}$
- $3 \uparrow 2 = 3^2 = 9$
- $3 \uparrow\uparrow 2 =^2 3 = 3^3 = 27$
- $3 \uparrow\uparrow 3 =^3 3 = 3^{3^3} = 3^{27} > 7 \cdot 10^{12}$

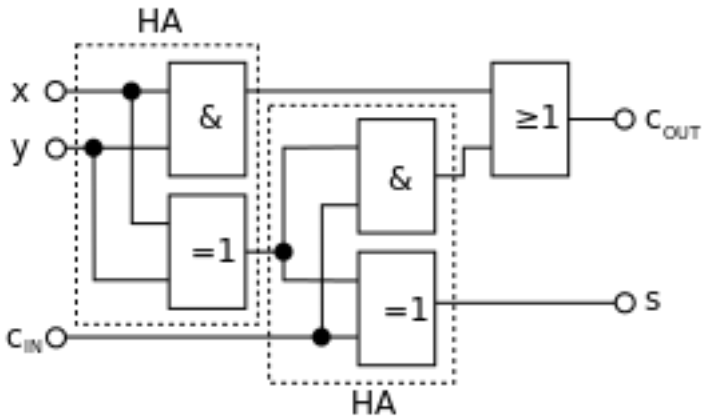
Addition n stelliger ganzer Zahlen

- OR: $x + y$
- AND: $x \cdot y$
- NOT: $\overline{x}$
- XOR: $x \oplus y$

Halbaddierer:
- Übertrag: $c = a \cdot b$ (carry)
- Summe: $s = a \oplus b$

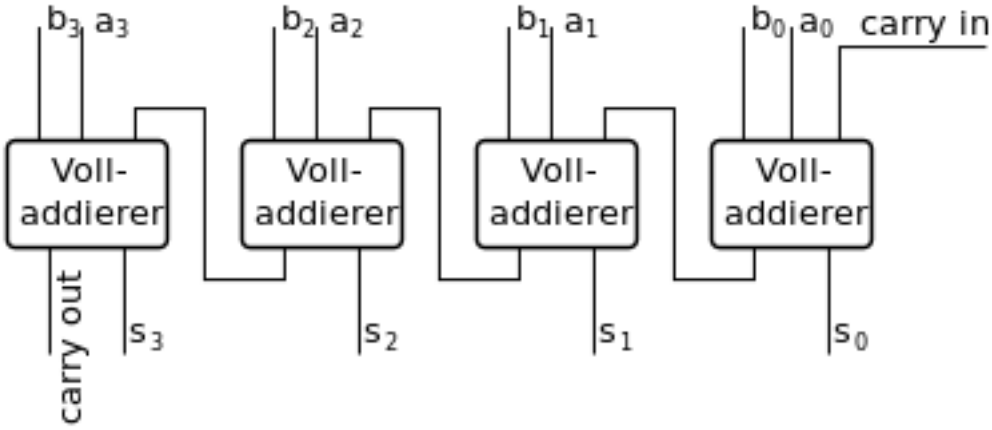


Volladdierer:
- Übertrag: $c_{out} = ab + ac_{in} + bc_{in}$
- Summe: $s = a\overline{b}\,\overline{c_{in}} + \overline{a}b\,\overline{c_{in}} + \overline{a}\,\overline{b}c_{in} + abc_{in} + a \oplus b \oplus c_{in}$



Addition n stelliger Zahlen mittels Carry-Ripple Adder:

- $a = \left[a_{n-1}a_{n-2}\ldots a_1a_0\right]_2$
- $b = \left[b_{n-1}b_{n-2}\ldots b_1b_0\right]_2$



$\Rightarrow T(n) = \mathcal{O}(T_{FA} \cdot n)$

Übertragsanalyse:

Eine Bitposition $i \in \{0,1,\ldots,n-1\}$ kann 3 verschiedene Übertragsfälle annehmen:

- kein Übertrag möglich, wenn $a_i = b_i = 0$
- Übertrag wird weitergeleitet (carry propergate) $p = a_i \oplus b_i$
- Übertrag wird auf jeden Fall generiert (generate bit) $g = a_i \cdot b_i$

Rekursionsbeziehung: $c_{i+1} = g_i + p_i \cdot c_i = a_ib_i + (a_i \oplus b_i) \cdot c_i = a_ib_i + (a_i + b_i) \cdot c_i$

$$c_{i+1} = \underbrace{g_i + p_ig_{i-1} + p_ip_{i-1}g_{i-2} + \ldots + p_ip_{i-1}\ldots p_1g_0}_{G_{0,i}} + \underbrace{p_ip_{i-1}\ldots p_1p_0}_{P_{0,i}} \cdot c_0$$

2er Komplement:

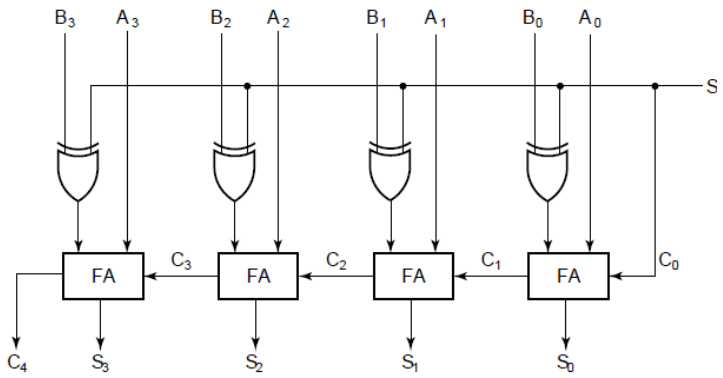
- $B(x) = \left[x_{n-1}x_{n-2}\ldots x_1x_0\right]_2 = \sum_{i=0}^{n-1} x_i 2^i$ Bit-Vektor, der x repräsentiert
- Im 2er Komplett gilt immer: $B(a) + B(-a) = 2^n$

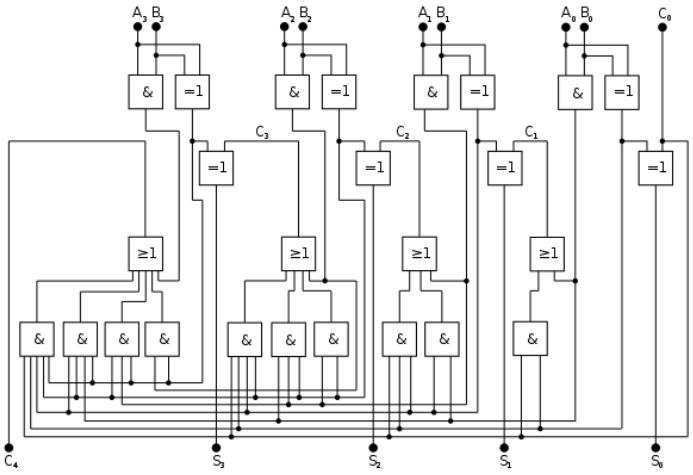
Subtraktion n stelliger ganzer Zahlen

- möglichst die gleiche Hardware wie für die Addition nutzen
- $a - b = a + \underbrace{(-b)}_{\text{2er Komplement}} = a + \underbrace{(\overline{b})}_{\text{1er Komplement}} + 1$
- auch in der hinteren Position wird ein Volladdierer benötigt. Addition und Subtraktion laufen über die selben Hardware

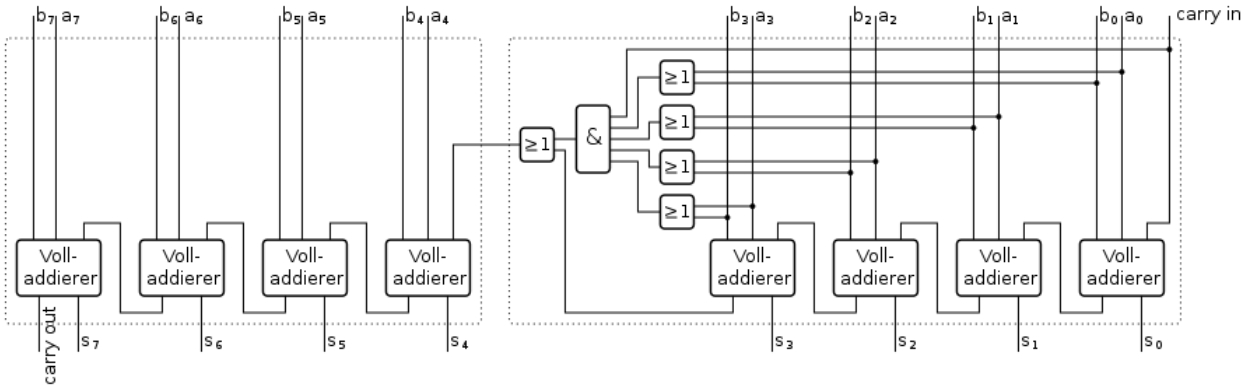
Carry-Ripple-Adder: $T(n) = \mathcal{O}(n) \Leftarrow (3n+1)$ tu

Carry-Lookahead-Adder: $T(n) = \mathcal{O}(\log_2 n)$





Carry-Lookahead-Adder



Carry-Skip-Adder

Gatter-Laufzeiten (tu = time units)
- 0 tu für `NOT`
- 1 tu für `AND` und `OR`
- 2 tu für `XOR`
- 4 tu für Volladdierer
- 3 tu für Carry in Ripple-Prozess

Multiplikation n stelliger ganzer Zahlen

Shifting:

- $\text{LShift}_1(a)$ … verschiebt das Bitmuster von a um 1 Zeichen nach links, a_k fliegt raus, es werden Nullen eingefügt
- $\text{RShift}_1(a)$… verschiebt das Bitmuster von a um 1 Zeichen nach rechts, a_0 fliegt raus, es werden Nullen eingefügt

Multiplikation mit wiederholter Summation

$$a \cdot b = r \Rightarrow r = \sum_{i=0}^{n-1} a_i \cdot 2^i \cdot b$$

Optimierung:

- Gruppe von k Nullen in a: sofortiger $\text{ARShift}_k(a)$
- Gruppe von k Einsen in a: $a = \ldots 0 \underbrace{\underbrace{1 \ldots 1}_{l} \underbrace{1 \ldots 1}_{j}}_{k} 0$. Dann $\sum_{i=j}^{l} 2^i = 2^{l+1} - 2^j = 2^{j+k} - 2^j$

Booth-Algorithmus:

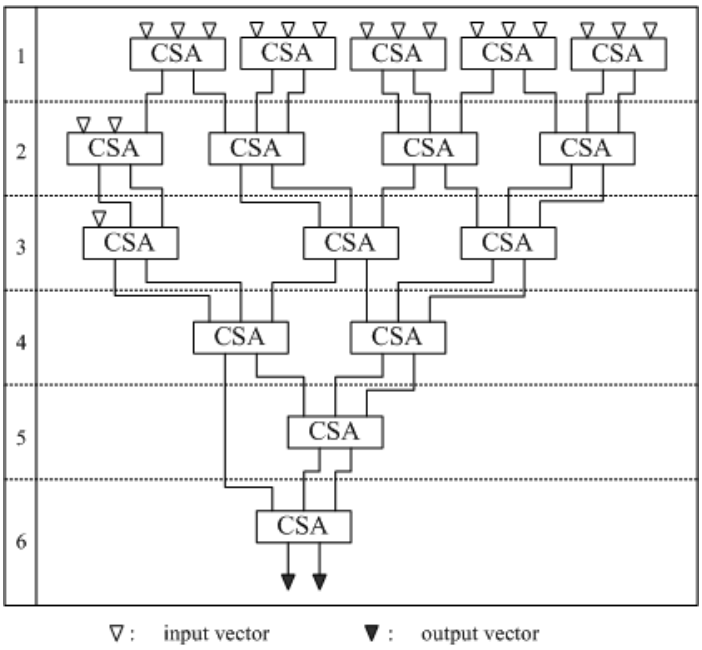
- Idee: $a \cdot b$ mit $b = c - d$ ergibt $a \cdot b = a \cdot c - a \cdot d$
- Kodierung des 1. Faktors: Dem mittels Booth zu kodierenden Operand $Y = (y_{n-1}, \ldots, y_0)$ fügt man eine weitere Stelle $y_{-1} = 0$ ein. Der kodierte Operand $Y' = (y'_{n-1}, \ldots, y'_0, y_{-1})$ wird wie folgt berechnet: $y'_i = y_{i-1} - y_i$
- Berechne $44_{10} = (00101100)_2 \cdot 17_{10} = (00010001)_2$
- Kodierter 1. Faktor: $(0, +1, -1, +1, 0, -1, 0, 0)$

									0	0	0	1	0	0	0	1	2. Faktor
·									0	1	-1	1	0	-1	0	0	Kodierung des 1. Faktors
+		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	keine Addition
+		0	0	0	0	0	0	0	0	0	0	0	0	0	0		keine Addition
+		1	1	1	1	1	1	1	1	0	1	1	1	1			2er-Komplement (2. Faktor)
+		0	0	0	0	0	0	0	0	0	0	0	0				keine Addition
+		0	0	0	0	0	0	1	0	0	0	1					2. Faktor
+		1	1	1	1	1	0	1	1	1	1						2er-Komplement (2. Faktor)
+		0	0	0	0	1	0	0	0	1							2. Faktor
+		0	0	0	0	0	0	0	0								keine Addition
1	0	0	0	0	0	0	1	0	1	1	1	0	1	1	0	0	Ergebnis ohne Überlauf
=							1	0	1	1	1	0	1	1	0	0	Ergebnis mit Überlauf

- Statt mit 0100000, 0001000 und 0000100 zu multiplizieren und die Ergebnisse zu addieren, wird nun also mit 1000000, 0100000, 0010000 und 0000100 multipliziert und entsprechend die Ergebnisse addiert bzw. subtrahiert.

- Wie man am Beispiel sieht, kann sich die Anzahl der Additionen auch erhöhen (im Beispiel von 3 auf 4), was ja aber gerade nicht erwünscht ist. Im statistischen Durchschnitt werden im Booth-Verfahren genauso viele Additionen gebraucht wie ohne Booth-Verfahren. Der Vorteil liegt aber darin, dass in der Informatik keine Gleichverteilung von Zahlen vorliegt. Vielmehr gibt es häufig Zahlen mit vielen Nullen und durch das Zweierkomplement bei negativen Zahlen häufig viele Einsen am Anfang. Nur durch diese Tatsache hat das Booth-Verfahren Vorteile gegenüber einer normalen Multiplikation.
- $44 \cdot 17 = 748 = (1011101100)_2$

Wallace-Tree:



- Idee: $\underbrace{\left(\sum_{k=0}^{n} a_k 2^k\right)}_{\text{Binärdarstellung } a} \cdot \underbrace{\left(\sum_{k=0}^{n} b_k 2^k\right)}_{\text{Binärdarstellung } b} = \sum_{k=0}^{2n} \sum_{i+j=k} a_i b_j 2^k$
- Der Wallace-Tree-Multiplizierer geht in drei Schritten vor:
  - Berechne für jedes Paar (i, j) mit $1 \leq i \leq n$ und $1 \leq j \leq k$ das Partialprodukt $a_i b_j 2^{i+j}$
  - Addiere die Resultate dieser Berechnung innerhalb der für den Wallace-Tree-Multiplizierer spezifischen Baumstruktur stufenweise mithilfe von Voll- und Halbaddierern, bis nur noch 2 Zahlen übrig sind, die addiert werden müssen
  - Addiere diese beiden Zahlen mit einem normalen Addierwerk
- CSA: Carry-Save-Adder: kann 3 Zahlen addieren, gibt 2 Zahlen aus: Sequenz der Partialsummen, Sequenz der der Übertragsbits → 2 gleich lange Zahlen
- Output des Wallace-Tree kommt in Carry-Lookahead-Adder
- $T(n) = \mathcal{O}(\log n)$