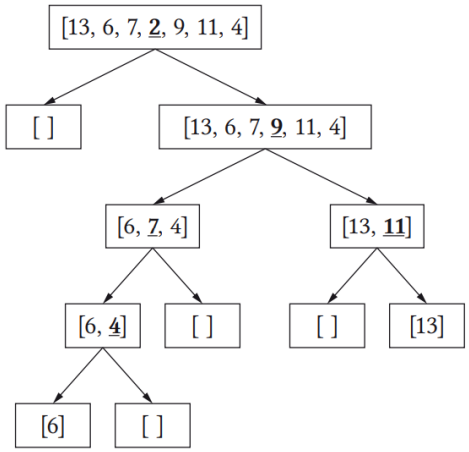
TGI02 Zusammenfassung

QuickSort

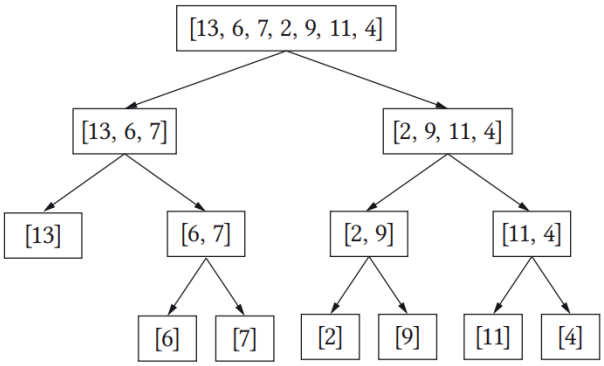
Beim QuickSort wird zuerst geprüft ob die eingegebene Liste <=1 ist. Ist dies nicht der Fall muss die Liste vorerst geteilt werden. Dabei wird ein Pivot und 2 neue Listen definiert.
Nun vergleicht man jedes Element in der Ausgangsliste mit dem Pivot Element, mit Ausnahme des Pivot Elements selbst.
Ist das Element kleiner kommt es in die erste Liste A1, ist es größer kommt es in die zweite Liste A2.



Nun wird die Methode immer wieder rekursiv mit der neu erstellten Liste aufgerufen und somit die Listen immer wieder in zwei geteilt, bis eine einelementige oder leere Liste entsteht.
Vom kleinsten Element beginnend werden nun nacheinander die Elemente in eine neu definierte Liste abgespeichert. Dabei wird zuerst das übrig gebliebene oder leere Element aus A1 hinzugefügt. Danach fügen wir das Pivot Element hinzu und zum Schluss das Element aus A2.

MergeSort

Beim MergeSort wird zuerst geprüft ob die eingegebene Liste <=1 ist. Ist dies nicht der Fall muss die Liste vorerst geteilt werden. Dabei wird ein Pivot und 2 neue Listen definiert. (same like QuickSort)
A1 bekommt einfach alle Elemente der Ausgangsliste mit dem Index 0 → Pivot, A2 bekommt die restlichen inklusive dem Pivot Element.
Wir definieren eine Variable s1 und s2 und rufen solange MergeSort rekursiv auf, bis jedes Element für sich alleine steht (siehe letzte Zeile Bild).
Solange in s1 und s2 noch Elemente vorhanden sind, wird mit dem ersten Element der Liste (Index 0) verglichen. Das kleinere von beiden wird in Asort übertragen und in der Liste, aus der es entnommen wurde, gelöscht.



Sind am Ende nur noch Elemente in einer von beiden Listen, werden sie auch in die Asort liste hinzugefügt und Asort wird ausgegeben. Es wird immer wieder eine neue Liste Asort definiert und die beiden vordersten Elemente mit dem Index 0 verglichen und nacheinander in die neue Liste eingefügt.

QuickSelection (k größten Wert ermitteln)

Beim QuickSelection teilt man zunächst die Liste bezüglich eines frei gewählten pivot in zwei Teillisten auf. Man sucht aber nur auf der Seite weiter, die das gesuchte Element enthält. Dazu muss man sich nur die Länge der Liste und den index des k größten Wertes anschauen.

$L = [3,2,1,6,5,4] \quad k = 2$

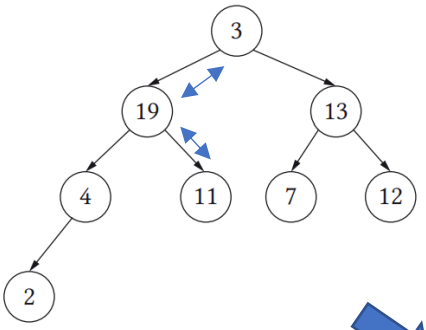
3 pivot

Das k größte Element wird gesucht. Die 3 wurde als pivot herausgepickt. Nun vergleicht man alle Elemente mit der 3, und schreibt die kleineren Links vom pivot in die Liste und die größeren rechts.
→ $L = [\underset{\text{Less}}{2,1},3,\underset{\text{more}}{6,5,4}]$ Somit steht fest, das 3 in der richtigen Index pos. Steht.
Nun sucht man auf der Seite weiter, in der sich das gesuchte K-Element aufhält.

Heapsort

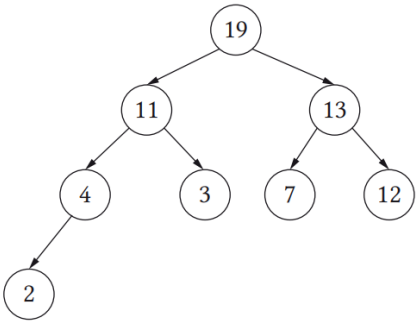
Durchsickern bedeutet, man vertauscht schrittweise mit dem Maximum der jeweils aktuellen Söhne.

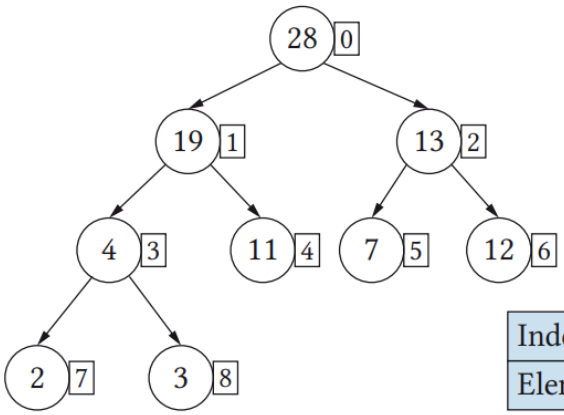
$L = [3, 19, 13, 4, 11, 7, 12, 2]$



Nun werden schrittweise die Zahlen mit dem **Maximum der jeweils aktuelen Söhne** durchgesickert.

$L = [3, 19, 13, 4, 11, 7, 12, 2]$
$L = [19, 3, 13, 4, 11, 7, 12, 2]$
$L = [19, 11, 13, 4, 3, 7, 12, 2]$



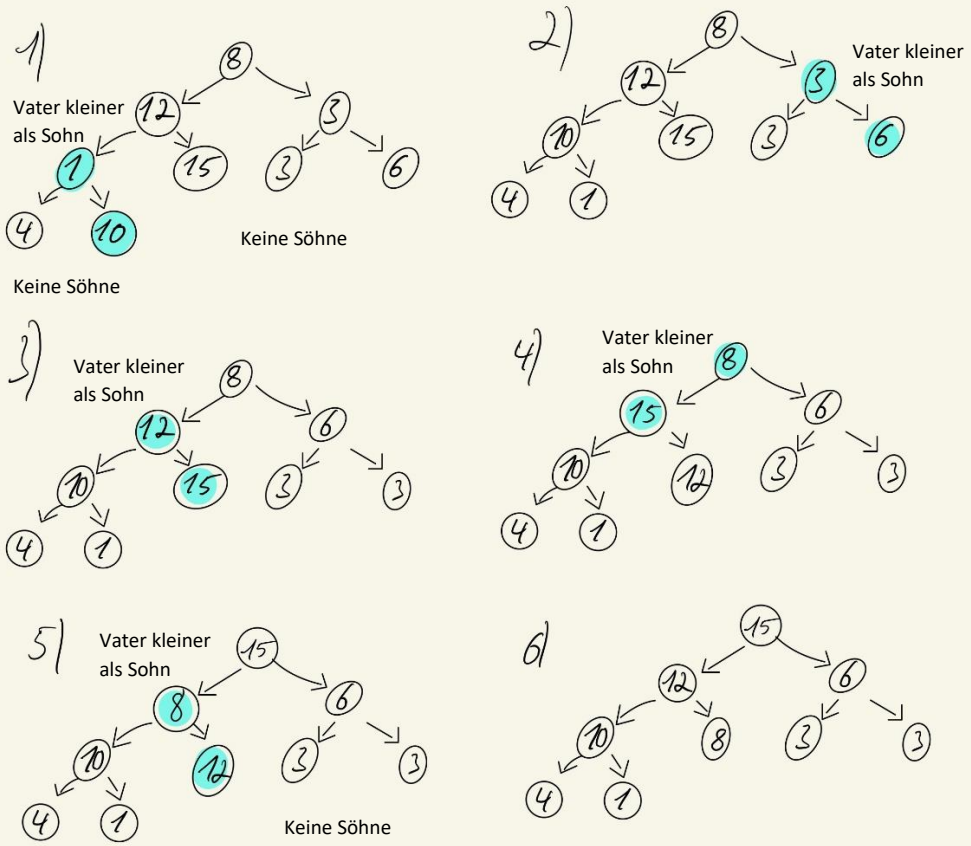


Die Kinder zum Knoten x haben die Indizes (2x+1) und (2x+2)

Index	0	1	2	3	4	5	6	7	8
Element	28	19	13	4	11	7	12	2	3

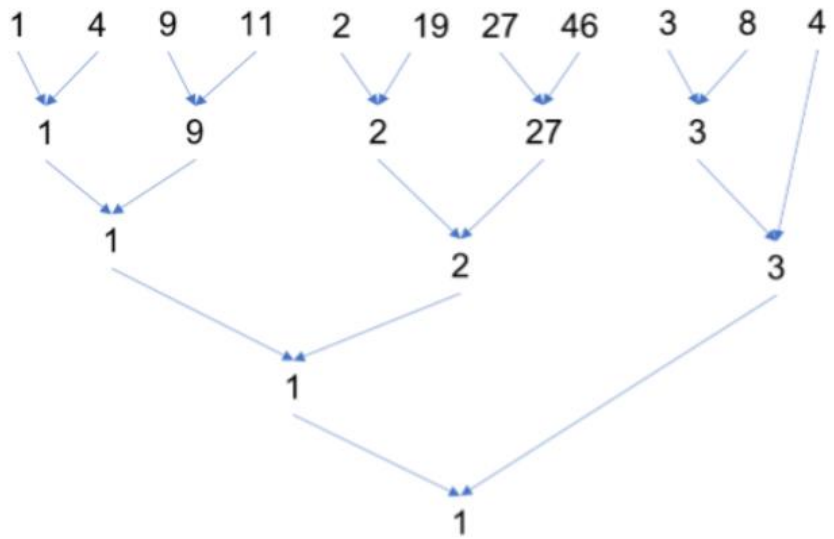
2x+1
2x+2

Beispiel: Gültigen Heap erstellen



Turniermethode

Ich wähle aus der Liste der Elemente Paare von Nachbarelementen und vergleiche die Elemente miteinander. Die jeweils kleineren Elemente kommen weiter und werden erneut in Paaren zusammengetragen. Elemente, die in keine Paare kommen (bei ungerader Listenlänge) werden einfach übertragen. Das kann man schematisch schön in einem Baum darstellen.



Damit steht das Minimum fest, die 1. Um das Zweitkleinste zu finden, kann ich nun auf die im Baum gespeicherte Historie zurückgreifen. Das zweitkleinste Element ist unter denjenigen zu suchen, die gegen das Minimum einmal verloren haben. Das ergibt $3, 2, 9, 4$. Unter diesen Elementen führe ich eine normale Minimumsuche durch. Das ergibt 2. Damit steht auch das Zweitkleinste fest.

Hashfunktion:

Eine Hashfunktion ist **surjektiv**, weil jeder Schlüsselwert $k \in K$ einen Index $h(k)$ in der Hashtabelle zugeordnet wird.
Größe der Hashtabelle H bezeichnet man mit m.
Bsp: $m = 100$ somit Indizes $\{0,1, \dots ,99\}$

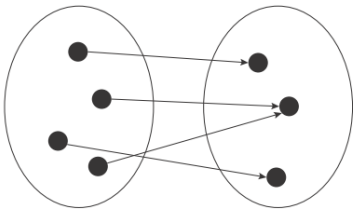


Abb. 3.5: Surjektive Operation

Für eine Menge von Schlüsseln $K = \{k \in \mathrm{N},\ k \leq 1\,000\}$ steht eine Hashtabelle der Größe 101 zur Verfügung. Welche der folgenden Funktionen wäre dann eine geeignete Hashfunktion?

a) $h(k) = k^2$ Nein, diese Funktion liefert zwar als Ergebnis wie gewünscht natürliche Zahlen, aber leider nicht im Bereich zwischen 0 und 100.
b) $h(k) = \sqrt{k}$ Nein, da wir die geforderte Ganzzahligkeit nicht erreichen.
c) $h(k) = 0$ Ja, aber ist sehr schlecht, da wir hier nur Kolliisionen erreichen
d) $h(k) = k\ DIV\ 10$ Ja, mit sehr guten Eigenschaften, da wir hier die schlüssel auf das geforderte Intervall abbilden.

Geeignete Hashfunktionen

- h sollte schnell berechenbar sein
- h sollte **surjektiv** sein
  Zu jedem Element in der Hashtabelle sollte es mindestens einen Schlüssel geben, der dieses Element als Index zgewiesen erhält. Da die Größe der Hashtabelle sehr klein ist verglichen mit der Anzahl der Schlüssel, sollte man die Hashtabelle wenigstens „voll ausnutzen".
- h sollte die zu speichernden Schlüssel möglichst gleichmäßig über die Hashtabelle verteilen.

Divisionsrestverfahren
Jede Funktion $h(k) = k\%m$ stellt eine gute Hashfunktion dar. Die Schlüssel werden ganzzahlig durch die Größe der Hashtabelle dividiert und der Divisionrest ergibt den gesuchten Index. Dabei muss die Größe m geeignet wählen. Ungeschickt wäre es, die Zahl 10^x zu wählen, denn das Ergebnis wäre somit immer die letzte Ziffer. Am besten sollte $\boldsymbol{m}$ eine **Primzahl** sein.
Beispiel: $m = 997\ k = 4641 \quad h(4641) = 4641\%997 = 653$

Geschlossene Verfahren (Kollisionsauflösung durch Verkettung)
Geschlossene Verfahren setzen diese Idee um und halten für den Fall einer Kollision in der Hashtabelle einen separaten Überlaufbereich vor. Darin werden die Datensätze, deren Hashadressen bereits belegt sind, in linearen Listen verwaltet.

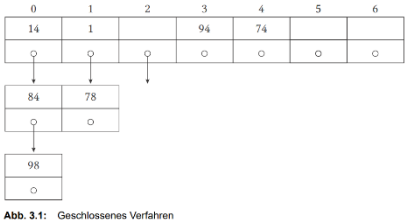


Abb. 3.1: Geschlossenes Verfahren

Offene Verfahren
Offene Verfahren lösen das Kollisionsproblem dadurch, dass sie eine noch freie alternative Adresse innerhalb der Hashtabelle ermitteln. Diese Verfahren suchen quasi noch eine offene Adresse in der Tabelle.

0	1	2	3	4	5	6
14	1	78	84	74	94	98

94 & 98 rutschen soweit nach hinten, weil die anderen Schlüssel zuerst angeguckt wurden & diese dann den Platz geklaut haben.

Vorteil: kein zusätzlicher Speicher | Nachteil: Klümpchenbildung

Berechenbarkeit von Algorithmen

Zeile	Anweisungsschritt	Zeitkomplexität
1.	`i = 1`	c_1
2.	`while (i < 11) {`	$11c_2$
3.	`  x = 5`	$10c_3$
4.	`  i++`	$10c_4$
5.	`}`	

$11c_2 = (10x\ i < 11) + 1x\ testen$

$T_{code}(n) = c_1 + 11c_2 + 10c_3 + 10c_4 = d$

Zeile	Anweisungsschritt	Zeitkomplexität
1.	`A = … // bspw. [7,129,88,4,5,6,7]`	c_1
2.	`sum = 0`	c_2
3.	`n = A.size()`	c_3
4.	`i = 0`	c_4
5.	`(0..n-1).each { i ->`	nc_5
6.	`  sum = sum + A[i]`	nc_6
7.	`}`	
8.	`println sum`	c_8

$T_{code}(n) = c_1 + c_2 + c_3 + c_4 + nc_5 + nc_6 + c_8$
$T_{code}(n) = (c_5 + c_6)n + c_1 + c_2 + c_3 + c_4$
$T_{code}(n) = d_1 n + d_2$

BubbleSort

Zeile	Anweisungsschritt	Zeitkomplexität
	`def bubbleSort(A) {`	$T_{\text{bubbleSort}}(n)$
1.	`  if (A.size() > 1) {`	c_1
2.	`    (0..A.size()-1).each { i ->`	nc_2
3.	`      (0..A.size()-2).each { j ->`	$n(n-1)c_3$
4.	`        if (A[j] > A[j+1]) {`	$n(n-1)c_4$
5.	`          temp = A[j]`	$n(n-1)c_5$
6.	`          A[j] = A[j+1]`	$n(n-1)c_6$
7.	`          A[j+1] = temp`	$n(n-1)c_7$
8.	`        }`	
9.	`      }`	
10.	`    }`	
11.	`  }`	
12.	`  return A`	c_{12}
13.	`}`	

i=0 bis n-1 ergibt **n** Schleifendurchgänge

J=0 →n-2==n-1;
Jetzt gehen wir j=0→n-2 die Schleife durch,
0→n-2 ergibt **n-1 Schleifendurchgänge**.
Dies tun wir **n** mal.
$n(n-1)c_3$

$$T_{bubbleSort}(n) = (c_3 + c_4 + c_5 + c_6 + c_7)n^2 - (-c_2 + c_3 + c_4 + c_5 + c_6 + c_7)n + c_1 + c_{12}$$
$$T_{bubbleSort}(n) = d_1n^2 - d_2n + d_3$$

InsertionSort

Zeile	Anweisungsschritt	Zeitkomplexität
	`def insertionSort(A) {`	$T_{\text{insertionSort}}(n)$
1.	`if (A.size() >= 2) {`	c_1
2.	`(1..A.size()-1).each { i ->`	$(n-1)c_2$
3.	`key = A[i];`	$(n-1)c_3$
4.	`j = i-1;`	$(n-1)c_4$
5.	`while(j>=0 && A[j] > key) {`	$\underbrace{(n-1)c_5}_{1} + c_5\sum_1^n t_i$
6.	`A[j+1] = A[j];`	$c_6\sum_1^n t_i$
7.	`j = j-1;`	$c_7\sum_1^n t_i$
8.	`}`	
9.	`A[j+1] = key;`	$(n-1)c_9$
10.	`}`	
11.	`}`	
12.	`return A`	c_{12}
13.	`}`	

1→n-1 = (n-1)c

$(n-1)c_5$ symbolisiert das **erste** Testen in der for Schleife.
Wurde die While Anforderung bestanden wird mit $c_5\sum_1^n t_i$ jeder Schleifendurchgang mitgezählt.

Beispiel

Einfügen der 8

4	8 |	6	1

$t_1 = 0$ c_5

Einfügen der 6

4	8	6 |	1

$t_2 = 1$ $c_5 + 1(c_5 + c_6 + c_7)$

Einfügen der 1

4	6	8 |	1

$t_3 = 3$ $c_5 + 3(c_5 + c_6 + c_7)$

While Anforderung bestanden somit c_5. Nun zählen wir jeden **weiteren** Schleifendurchgang mit $c_5\sum_1^n t_i$ mit. $c_5 + 3(c_5 + c_6 + c_7)$

testen + 3x Schleifendurchgänge = 4x testen + 3 Schleifendurchgänge

$$3c_5 + (t_1 + t_2 + t_3)(c_5 + c_6 + c_7) = 7c_5 + 4(c_6 + c_7)$$

$$T_{InsertionSort}(n) = c_1 + c_{12} + (n-1)(c_2 + c_3 + c_4 + c_5 + c_9) + (c_5 + c_6 + c_7)\sum_1^n t_i$$

$$T_{InsertionSort}(n) = d_5 + (n-1)d_6 + d_3\sum_1^n t_i$$

$$T_{InsertionSort}(n) = d_5 + nd_6 - d_6 + d_3\sum_1^n t_i$$

$$T_{InsertionSort}(n) = d_1 + nd_2 + d_3\sum_1^n t_i$$

TGI02 Zusammenfassung

Funktionswachstum

Sei $f(n) = 0{,}5n^2$ gegeben. Entscheiden Sie, welche Funktionen der folgenden Auswahl Lower Bounds und welche Upper Bounds sind.

a) $g_1(n) = 0{,}3n^2$ $g_1(n)$=lower bound. Beide Funktionen haben den gleichen höchsten Exponenten aber der Koeffizent von $g_1(n)$ ist kleiner

b) $g_2(n) = 0{,}4n^2$ $g_2(n)$=lower bound. Selbe wie $g_1(n)$

c) $g_3(n) = 0{,}8n^2$ $g_3(n)$=upper bound. Beide Funktionen haben den gleichen höchsten Exponenten aber der Koeffizent von $g_3(n)$ ist größer.

d) $g_4(n) = 2n^3$ $g_4(n)$=upper bound. $g_4(n)$ hat den größeren Exponenten.

e) $g_5(n) = 5{,}5n^1 + 77$ $g_5(n)$=lower bound. Der Höchste Exponent von $g_5(n)$ ist kleiner als f(n)

f) $g_6(n) = 1^n$ $g_6(n)$=lower bound, da 1^n nicht anderes ist als die konstante Funktion 1.

g) $g_7(n) = 2^n$ $g_7(n) = upper\ bound$, Exponentialfunktion wächst immer schneller als Polynominalfunktion

h) $g_8(n) = 4\log_3(n)$ $g_8(n) = lower\ bound$, Logarithmische Funktionen wachsen sehr langsam.

Sei $f(n) = n^2$ gegeben. Entscheiden Sie, für welche i der folgenden Auswahl gilt: $g_i(n) = O(f)$, und begründen Sie Ihre Entscheidung.

a) $g_1(n) = 0{,}3n^2$ $g_1(n) = 0{,}3n^2 = O(n^2)$. Die Funktion n^2 liegt bereits immer über $0{,}3n^2$

b) $g_2(n) = 1{,}4n^2$ $g_2(n) = 1{,}4n^2 = O(n^2)$. Dank der multiplikativen Konstanten in $f(n)$

c) $g_4(n) = 2n^3$ $g_4(n) = 2n^3 \notin O(n^2)$. $n^3 > C * n^2$

d) $g_5(n) = 5{,}5n^1 + 77 = O(n^2)$. n^2 wird diese Funktion irgendwann überholen

e) $g_6(n) = 1^n$ $g_6(n) = \mathbf{1}^n = O(n^2)$. Jede konstante Funktion, liegt unter einer quadratischen.

f) $g_7(n) = 2^n$ $g_7(n) = 2^n \notin O(n^2)$. Es gibt keine multiplkative Konstante um mitzuhalten

g) $g_8(n) = 4\log_3(n)$ $g_8(n) = 4\log_3 n = O(n^2)$. Logarithmische Funktionen wachsen sehr langsam, die Funktion n^2 wächst wesentlich schneller.

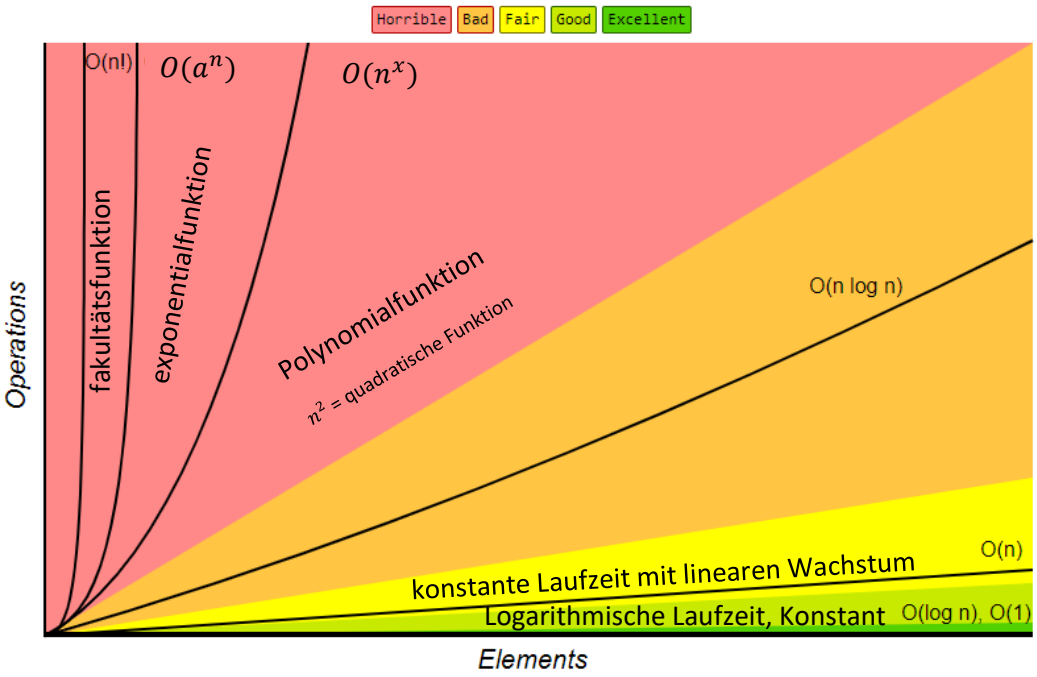
$g_i(n) = O(n)$ wenn $O(x * n^2) > g_i(n)$ ist. Dabei darf x frei gewählt werden.

$$\log n \leq \sqrt{n} \leq n \overset{*}{=} 2^{\log n} \leq n \log n \leq n\sqrt{n} \leq n^2 \leq n^3 \leq 2^n \leq n! \leq n^n \ .$$

für alle n größer als ein gewisses n_0.
$(*)$ Die Aussage "$n = 2^{\log n}$" gilt nur für den Logarithmus zur Basis 2.

$$\mathcal{O}(1) < \mathcal{O}(\log x) < \mathcal{O}(x) < \mathcal{O}(x \cdot \log x) < \mathcal{O}(x^2) < \mathcal{O}(x^n) < \mathcal{O}(x^{\log x}) < \mathcal{O}(2^x)$$

$$O(n^2) \subset O(n^3) \subset O(2^n) = O(2^{n+3}) \subset O(2^{2n})$$



Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

- $f(n) = 2^n + n^5 = O(3^n)$
  Die Aussage stimmt, denn für fast alle n gilt $\boldsymbol{2^n < C * 3^n}$ **und** $\boldsymbol{n^5 < C * 3^n}$ **somit ist die Bedigung erfüllt**
  $\boldsymbol{f(n) = O(3^n)}$

- $f(n) = n! = O(2^n)$
  Diese Aussage stimmt nicht, denn für fast alle n gilt = $n! > C * (2^n * \log n)$.
  Somit wird die Bedingung $\boldsymbol{n! \leq c * 2^n}$ **verletzt.**
  $f(n!) \notin O(2^n * \log n)$

- $f(\log n) \in O(1)$
  Die Aussage stimmt nicht, denn für fast alle n gilt = $\log n > c * 1^n$.
  Somit wird die Bedingung $\boldsymbol{\log n \leq c * 1^n}$ **verletzt.**

Sei eine Zeitkomplexitätsfunktion **T(n)** wie folgt gegeben:

$T(n) = O(1)$ $\quad falls\ n \leq n_0$

$\boldsymbol{T(n) = a * T\left(\frac{n}{b}\right) + n^k}$ $\quad falls\ n > n_0$

$Falls\ a \geq 1, b > 1\ gilt\ auch$:

$T(n) = O(n^k)$ $\quad falls\ a < b^k\ \ Fall\ 1$

$T(n) = O\left(n^k + \log_2(n)\right)$ $\quad falls\ a = b^k\ \ Fall\ 2$ } des Masterthoerem

$T(n) = O(n^{\log_b a})$ $\quad falls\ a > b^k\ \ Fall\ 3$

Sei eine Zeitkomplexitätsfunktion **T(n)** wie folgt gegeben:

$\boldsymbol{T(n) = b * T(n - c) + f(n)}$

$T(n) = O(b^{\frac{n}{c}})$ $\quad falls\ b > 1 \quad Chip\ and\ Be\ Conquered$

$T(n) = \sum_{d=0}^{n/c} f(cn) \approx \frac{1}{c}\int_0^n f(x)dx$ $\quad falls\ b = 1 \quad Chip\ and\ Conquer$

$$\frac{1}{c}\int_0^n f(x)dx = \frac{1}{2}\left(n * f(x) - 0 * f(x)\right) = 1/2\left(n * f(x)\right) = O(n * f(x))$$

Beispiel

a) $T(n) = T\left(\frac{n}{3}\right) + n^2 = T(n) = a * T\left(\frac{n}{b}\right) + n^k \rightarrow a = 1\ \ b = 3\ \ k = 2$
$a \geq 1\ und\ b > 1\ somit\ gilt$
$\boldsymbol{T(n) = O\left(n^k\right)}$ $\quad a < b^k \rightarrow 1 < 3^2\ Fall\ 1$

b) $T(n) = 9T\left(\frac{n}{3}\right) + n = T(n) = a * T\left(\frac{n}{b}\right) + n^k \rightarrow a = 9\ \ b = 3\ \ k = 1$
$a \geq 1\ und\ b > 1\ somit\ gilt$
$\boldsymbol{T(n) = O\left(n^{\log_3 9}\right) = O(n^2)}$ $\quad a > b^k \rightarrow 9 > 3^1\ Fall\ 3$

c) $T(n) = 3T\left(\frac{n}{4}\right) + n * \log_2(n) \rightarrow a = 3\ \ b = 4\ \ k = n * \log(n) \cong 2;$
$3 < 4^k \rightarrow 3 < 4^k\ (egal\ wie, es\ wird\ größer\ werden)$
$\boldsymbol{T(n) = O(n^k)}$ $\quad Fall\ 1$

d) $T(n) = T(n-2) + n^d = T(n) = b * T(n-c) + f(n) \rightarrow b = 1\ \ c = 2\ \ f(n) = n^d$
$$T(n) = \frac{1}{c}\int_0^n n^d dx = \frac{1}{2}\left(n^d * n - n^d * 0\right) = \frac{1}{2}(n^d * n) = \boldsymbol{O\left(n^{d+1}\right)}$$

e) $T(n) = T(n-1) + \log_2(n) \rightarrow b = 1\ \ c = 1\ \ f(n) = \log_2(n)$
$$T(n) = \frac{1}{1}\int_0^n \log_2(n)dx = \log_2 n * n - \log_2 n * 0 = \log_2 n * n = \boldsymbol{O(n * \log n)}$$

f) $T(n) = 4T(n-2) + n^3 \rightarrow b = 4\ \ c = 2 \rightarrow T(n) = \boldsymbol{O(b^{\frac{n}{2}})}$

Platzkomplexität

Zeile	Programmcode	Zeitkomplexität	Platzkomplexität
	`def berechne(n) {`	$T(n)$	$S(n)$
1.	`i = 1`	c_1	c_{int}
2.	`a = 2*i`	c_2	c_{int}
3.	`i++`	c_3	0
4.	`(0..n).each {…} ..}`	nc_4	$\log(n)$, c_{int}

Eine Schleife braucht vor allem eine Laufvariable. Eine Zahl der Größe n als Laufvariable für eine Schleife benötigt im Binärsystem $\log(n)$ Zellen, um gespeichert zu werden.

(Wichtige Komplexitätsklassen)

Man bezeichnet mit

- **LOGSPACE** die Klasse aller Probleme, deren Platzkomplexität logarithmisch ist,
- **P** die Klasse aller Probleme, deren Zeitkomplexität eine Polynomialfunktion (wie $T(n) = O(n^3)$) ist,
- **NP** die Klasse aller Probleme, deren Zeitkomplexität zur Berechnung des Ergebnisses zwar keine Polynomialfunktion sein muss, die aber einen entsprechenden Algorithmus in Polynomialzeit haben, mit dem überprüft werden kann, ob die Berechnung richtig war,
- **PSPACE** die Klasse aller Probleme, deren Platzkomplexität eine Polynomialfunktion ist, und
- **EXPTIME** die Klasse aller Probleme, deren Zeitkomplexität eine Exponentialfunktion ist.

$$\ldots \subseteq \boldsymbol{LOGSPACE} \subseteq \boldsymbol{P} \subseteq \boldsymbol{NP} \subseteq \boldsymbol{PSPace} \subseteq \boldsymbol{EXPTIME} \subseteq \cdots$$

Definition 6.6:

Die Zeit-Platz-Komplexität $TS(n) = T(n) \cdot S(n)$ eines Algorithmus beschreibt das asymptotische Verhalten der Funktion, die sich aus der Multiplikation der Zeit- und Speichergrößenfunktionen des Algorithmus ergibt, wobei n die Größe oder Länge der Eingabe charakterisiert.

MergeSort: Die Best-Case Laufzeit von Mergesort ist O(n · log n). Mergesort halbiert das Array unabhängig von der Reihenfolge der Arrayelemente, bis nur noch Arrays mit jeweils einem einzelnen Element übrig bleiben. Diese Halbierungen ergeben O(log n) viele Ebenen mit disjunkten Arrayabschnitten. Für jede dieser Ebenen wird die merge-Operation ausgeführt, die lineare Laufzeit hat (auch unabhängig von der Reihenfolge der Arrayelemente). Insgesamt ergibt sich also für Best-, Average- und Worst-Case die gleiche asymptotische Laufzeit von O(n · log n).

Tab. 6.1: MergeSort

Zeile	Anweisungsschritt	Zeitkomplexität
	`def mergeSort(A) {`	$T(n)$

1.	`if (A.size() <= 1) return A`
	`int pivot = (A.size() / 2)`
	`def A1 = []`
	`def A2 = []`
		c_1

2.	`A.eachWithIndex { e, i ->`
	`if ((i < pivot)) A1 << e`
	`if ((i >= pivot)) A2 << e`
	`}`
		nc_2

3.	`def S1 = mergeSort(A1)`	$T_{\text{mergeSort}}(n/2)$
4.	`def S2 = mergeSort(A2)`	$T_{\text{mergeSort}}(n/2)$
5.	`def Asort = []`	c_5

6.	`while ((S1.size() + S2.size()) > 0) {`
	`...`
	`}`
		??

7.	`return Asort` `}`	c_7

Tab. 6.4: QuickSort

Zeile	Programmcode	Zeitkomplexität
	`def quickSort(A) {`	$T(n)$

1.	`int pivot = (A.size() / 2)`
	`def A1 = []`
	`def A2 = []`
		c_1

2.	`A.eachWithIndex { e, i ->`
	`if ((i != pivot) &&`
	`(e <= A[pivot])) A1 << e`
	`if ((i != pivot) &&`
	`(e > A[pivot])) A2 << e`
	`}`
		nc_2

3.	`def Asort = []`	c_3
4.	`Asort.addAll(quickSort(A1))`	??
5.	`Asort << A[pivot]`	c_5
6.	`Asort.addAll(quickSort(A2))`	??
7.	`return Asort` `}`	c_7

TGI02 Zusammenfassung

HeapSort

HeapSort zerfällt in zwei Phasen: den Aufbau des Heaps und dann das schrittweise Entnehmen des Maximums und Durchsickern. Eine Analyse ist wirklich einfach: Ein Heap mit n Elementen hat maximal $n/2$ Blätter und damit eine Höhe $\log_2(n/2)$. Runden wir auf: n Nichtblätter (statt genaugenommen $n\ /\ 2$) in einen Heap der Höhe $\log_2(n/2)$ einspeichern kostet $n * \log_2(n)$. Das Gleiche trifft für die zweite Phase zu. Damit ist HeapSort wirklich optimal. (Allerdings bedeutend aufwendiger zu implementieren.)
Die zweite Phase wird sich immer so verhalten,wie die erste Phase, das Bilden des Heaps kann man auch mit O(n) abschätzen. Die meisten Elemente fallen beim Sickern nicht weit

RadixSort

RadixSort zerfällt auch in zwei Phasen: In einem einzigen Durchlauf werden die Elemente angesehen und in die sich außerhalb der Liste befindenden Boxen einsortiert. Das kostet etwa O(n). RadixSort lohnt sich nur, wenn die Elemente schön gleichmäßig auf die k einzelnen Boxen verteilt wurden und wir eine maximale Anzahl von Elementen, sagen wir m pro Box, haben. Jede Box sortiert man dann mit $m * \log_2(m)$, was wir wiederum mit einer Konstante d großzügig abschätzen. $k * d$ ergibt eine weitere Konstante, womit sich am Gesamtaufwand nichts ändert und RadixSort mit O(n) wirklich linear sortiert. Aber vergessen wir nicht, RadixSort benötigt auch extra Speicher und eine Gleichverteilung auf die Boxen wird voraussetzt.

QuickSelection:

Zeile	Anweisungsschritt	Zeitkomplexität
	def quickSelection(A,k)	$T_{QS}(n)$
1	If (A.size() <= 1) return A[A.size()-k] Int pivot = (A.size() /2) A1=[] A2=[]	c_1

2	A.eachWithIndex { e, i ->
	if ((i != pivot) && (e <= A[pivot])) A1 << e
	if ((e > A[pivot]) A2 <<e
	}
		nc_2

3	if (A2.size() >= k) return quickSelection(A2, k)
	If (A2.size() +1 ==k) return A[pivot]
	return quickSelection (A1, k-A2.size()-1)
		$T_{QS}(n-1)+n$
		c_3

$T_{QS,b}(n) = T(1) = O(1)$
$T_{QS,w}(n) = T_{QS,w}(n-1) + n = O(n^2)$