**1 Sortieren**

**Einfache Verfahren**

**QuickSort**

**MergeSort**

**Master-Theorem**

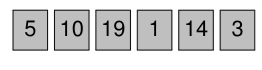
**Untere Schranke**

**Schnelleres Sortieren**

**Selektieren**

**Externes Sortieren**

**Sortierproblem**:

• Gegeben: Ordnung ≤ auf der Menge möglicher Schlüssel

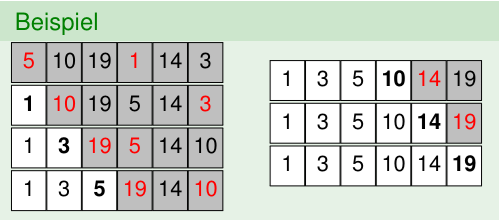
• Eingabe: Sequenz s = ⟨e1, . . . , en⟩ Bsp.:

• Ausgabe: Permutation s′ = ⟨e′1, . . . ,.e′n⟩ von s,

So dass key(e′i) ≤ key(e′i+1) fur alle i ∈ {1, . . . , n − 1} Bsp.:

* 1. **Einfache Verfahren**

**1.1.1 Selectionsort(Sort nach Wahl)**

Wähle das kleinste Element aus der (verbleibenden) Eingabesequenz (“SelectMinSort”) und verschiebe es an das Ende der Ausgabesequenz.首先在未排序序列中找到最小（或最大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（或最大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

void selectionSort(Element[ ] a, int n) {

// a 是待排序的数组，n 是数组的长度

for (int i = 0; i < n; i++) {

// 外部循环，从第一个元素开始，逐步扩展到整个数组

//将{a[i], ..., a[n-1]} 中的最小元素移动到a[i]位置”

int k = i; // 初始化k为当前外部循环的索引i，表示当前最小元素的位置

for (int j = i + 1; j < n; j++) // 内部循环，从i+1开始遍历剩余未排序的元素

if (a[j] < a[k]) k = j; // 如果发现更小的元素，则更新k为该元素的位置

swap(a, i, k); // 调用swap函数交换a[i]和a[k]（即最小元素）的位置

}}**Notiz: 双指针法，两个指针对应嵌套循环，外循环标记当前的数据位置，内循环逐一迭代；通过currMinValIdx = secIdx和arr[secIdx] < arr[currMinValIdx]比较，进而确定在乱序中最小的数据； 最后通过swap交换即可 （上页手抄代码！！！！！！！！）**

简单选择排序是最简单直观的一种算法，基本思想为每一趟从待排序的数据元素中选择最小（或最大）的一个元素作为首元素，直到所有元素排完为止，简单选择排序是不稳定排序。

在算法实现时，每一趟确定最小元素的时候会通过不断地比较交换来使得首位置为当前最小，交换是个比较耗时的操作。其实我们很容易发现，在还未完全确定当前最小元素之前，这些交换都是无意义的。我们可以通过设置一个变量min，每一次比较仅存储较小元素的数组下标，当轮循环结束之后，那这个变量存储的就是当前最小元素的下标，此时再执行交换操作即可

题外话：一个巧妙地swap实现方式：

public static void swap(int []arr,int a,int b){

arr[a] = arr[a]+arr[b];

arr[b] = arr[a]-arr[b];

arr[a] = arr[a]-arr[b];

}

这个swap方法的实现确实是一个巧妙的技巧，用于在不引入临时变量的情况下交换两个数组元素的值。我们可以通过逐步拆解这三行代码来理解它是如何工作的。

假设我们有两个要交换的数组元素arr[a]和arr[b]，其中arr[a]的值为x，arr[b]的值为y。

第一行代码：arr[a] = arr[a] + arr[b];

执行后，arr[a]的值变为x + y，而arr[b]的值仍为y。

第二行代码：arr[b] = arr[a] - arr[b];

此时，arr[a]的值是x + y，arr[b]的值是y。

执行这行代码后，arr[b]的值变为(x + y) - y，即x。注意，这里我们其实已经得到了原arr[a]的值x，并且存放在了arr[b]中。

第三行代码：arr[a] = arr[a] - arr[b];

此时，arr[a]的值是x + y，而arr[b]的值已经变为了x。

执行这行代码后，arr[a]的值变为(x + y) - x，即y。这样我们就成功地将arr[a]和arr[b]的值交换了。

这个技巧依赖于加法和减法的性质，并且有一个重要的前提：数组中的值不能过大，以至于相加后会导致整数溢出。如果arr[a]和arr[b]都是正数且它们的和超过了int的最大值，或者一个是正数另一个是负数且它们的绝对值之和超过了int的最大值（考虑负数的补码表示），那么这种方法就会导致错误的结果。

此外，如果arr[a]和arr[b]中的任何一个是Integer.MAX\_VALUE（即int类型的最大值），并且另一个不是Integer.MIN\_VALUE（即int类型的最小值），那么相加也会导致溢出。类似地，如果两者都是Integer.MIN\_VALUE，相减也会导致溢出（尽管在这种情况下，由于减法的性质，它实际上不会产生一个数学上的负数溢出，但结果仍然不正确，因为我们会试图将Integer.MIN\_VALUE赋值给arr[a]，这是不允许的）。

因此，虽然这种交换方法在某些情况下可以工作，但它并不总是安全的，特别是当处理可能非常大或特殊（如接近int的极限值）的整数时。在大多数情况下，使用临时变量进行交换是更安全、更直观的方法。

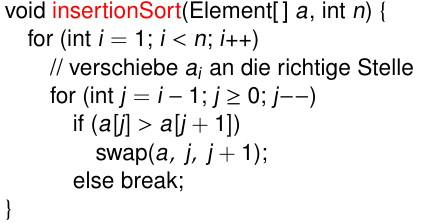
回归原课件，Zeitaufwand: • Minimumsuche in Feld der **Größe i: Θ(i)**

• Gesamtzeit**:∑i=1, n Θ(i) = Θ(n^2)**

• Vorteil: einfach zu implementieren //优缺点已然在笔记中写过了

• Nachteil: quadratische Laufzeit

**1.1.2 InsertionSort (Sort nach Einfügen)**

Nimm ein Element aus der Eingabesequenz und füge es an der richtigen Stelle in die Ausgabesequenz ein从输入序列中取出一个元素并将其插入到输出序列的正确位置

Korrektur：

void insertionSort(int[] arr) {

int n = arr. length;

for (int i = 1; i < n; i++) {

// 将arr[i]插入到arr[0..i-1]中的正确位置

int key = arr[i]; // 保存当前需要插入的值

int j = i - 1;

// 将大于key的元素向后移动

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j = j - 1;

}

// 将key插入到正确的位置

arr[j + 1] = key;

}

} //Erklärung: 内循环的条件和比较逻辑与插入排序的标准实现不完全一致。标准的插入排序在每次外循环迭代时，会将当前元素a[i]与其前面的已排序元素进行比较，并根据需要将它们向右移动以创建一个空间来插入当前元素。但在这个示例中，您描述的是比较a[j]和a[j+1]，并且当a[j] > a[j+1]时执行交换，这实际上是冒泡排序（Bubble Sort）的逻辑，而不是插入排序。

Zeitaufwand: • Einfügung des i-ten Elements an richtiger Stelle :**O(i)**

• Im best-case (Array Schon sortiert): **Θ(1)**

• Gesamtzeit:

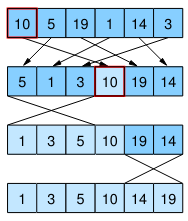
**∑i=1, n O(i) =O(n2)**

**• Im best-case: Θ(n)**

• Vorteil: einfach zu implementieren • Nachteil: quadratische Laufzeit (aber: linear im best-case!)

* 1. **QuickSort**

**• Idee:** Aufspaltung (Partitionierung) in zwei Teilmengen, getrennt durch ein Pivot Element: 通过一个称为“枢轴元素”（Pivot Element）的元素将待排序的数组划分为两个子数组，使得左边子数组的所有元素都小于或等于枢轴元素，而右边子数组的所有元素都大于或等于枢轴元素。然后，递归地对这两个子数组进行同样的操作，直到整个数组有序。(注意，左右半边元素个数不同是正常情况)



QuickSort {

wähle Pivot Element; 选择基准元素

// z.B. erstes, mittleres, letztes oder zufälliges Element splitte in kleinere und größere Schlüssel bzgl. Pivot Element; 基准元素例如数组开头，中间，结尾或者随机

// entweder in temporäre Arrays oder in-place 使用临时数组或者原地排序

// ein Scan des Felds ⇒ **O(n) Zeit** 划分操作的时间复杂度

sortiere Teil Feld mit kleineren Schlüsseln (rekursiv);

sortiere Teil Feld mit größeren Schlüsseln (rekursiv);

}

//外部存储：

public class QuickSortWithTempArrays {

// 使用临时数组进行划分的辅助方法

private static int[] partition(int[] arr, int low, int high) {

// 选择最后一个元素作为枢轴

int pivot = arr[high];

// 初始化两个临时数组

int[] left = new int[high - low + 1];

int[] right = new int[high - low + 1];

// 索引用于填充临时数组

int leftIndex = 0;

int rightIndex = 0;

// 遍历数组进行划分

for (int i = low; i < high; i++) {

if (arr[i] <= pivot) {

left[leftIndex++] = arr[i];

} else {

right[rightIndex++] = arr[i];

}

}

// 将枢轴放到中间位置

int combinedIndex = 0;

for (int i = 0; i < leftIndex; i++) {

arr[combinedIndex++] = left[i];

}

arr[combinedIndex++] = pivot;

for (int i = 0; i < rightIndex; i++) {

arr[combinedIndex++] = right[i];

}

// 返回新的划分点（枢轴的位置）

return new int[]{low + leftIndex, high};

}

// 快速排序主方法

public static void quickSort(int[] arr, int low, int high) {

if (low < high) {

// 获取新的划分点

int[] pivotBounds = partition(arr, low, high);

// 递归排序左右子数组

quickSort(arr, low, pivotBounds[0] - 1);

quickSort(arr, pivotBounds[1] + 1, high);

}

}

//内部存储：这段代码描述了一个典型的快速排序算法的实现，它使用了原地（in-place）排序技术，并且使用了“Lomuto划分方案”（也被称为“简单划分”或“右侧主元划分”）。

void quickSort(Element[ ] a, int ℓ, int r ) { // 假设 Element 是某种可以比较的数据类型，比如 int、Integer 或自定义对象

// a [ℓ . . . r ]: zu sortierendes Feld

if (ℓ < r ) { // 如果左边界小于右边界，则数组长度大于1，需要继续排序

p = a [ r ] ; // Pivot // 选择数组最后一个元素作为枢轴元素

int i = ℓ − 1; // 初始化两个指针，i从左边开始（但不包括l），j从右边开始

int j = r ;

do { // spalte Elemente in a [ℓ, . . . , r − 1 ] nach Pivot p

do { i++ } while ( a [ i ] < p); // 从左到右扫描，直到找到一个大于或等于枢轴的元素

do { j−− } while (j ≥ ℓ ∧ a [j ] > p); // 从右到左扫描，直到找到一个小于或等于枢轴的元素

// 注意：这里应该是 j > l 而不是 j ≥ l，因为我们要避免与 i 相遇时的重复比较

if ( i < j) swap( a , i , j); // 如果 i 和 j 没有交错，交换它们

} while ( i < j); // 重复上述过程，直到 i 和 j 交错

swap ( a , i , r); // Pivot an richtige Stelle // 将枢轴元素交换到其最终位置（即 i 的位置）

quickSort(a, ℓ , i − 1); // 递归地对枢轴左边和右边的子数组进行快速排序

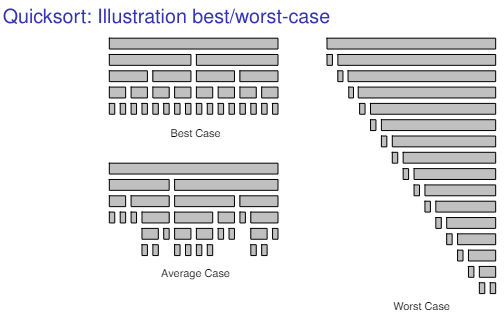
quickSort(a, i + 1, r); } }

**ZF：**

1. 选择一个枢轴元素（在这里是数组的最后一个元素）。
2. 使用两个指针 i 和 j 来扫描数组，并根据枢轴元素重新排列数组中的元素。
3. 当 i 和 j 交错时，将枢轴元素放到其最终位置（即 i 的位置）。
4. 递归地对枢轴左边和右边的子数组进行快速排序。

**Problem：**

• die Aufteilung in Teilprobleme kann ausbalanciert sein (also nur im Optimalfall eine Halbierung)

• im Worstcase quadratische Laufzeit!

(z.B. wenn Pivot Element immer kleinstes oder größtes aller Elemente ist)

• 子问题的划分可能不平衡（即仅在最佳情况下减半）

• 在最坏的情况下，运行时间是二次方！

（例如，如果主元元素始终是所有元素中最小或最大的）

**Lösungen:**

• wähle **zufälliges** Pivot Element: **Laufzeit O(n log n)** mit hoher Wahrscheinlichkeit 随机选取

• berechne **Median** (mittleres Element): mit **Selektionsalgorithmus** (nicht in dieser Vorlesung) 通过选择排序计算中位数

• wechsele zu einem anderen Algorithmus (**mit O(n log n))** wenn die Rekursionstiefe **log n** **überschreitet** 更换算法，递归深度超nlogn

Laufzeit bei zufälligem Pivot-Element:

• Zähle Anzahl Vergleiche (Rest macht nur konstanten Faktor aus) //这意味着在分析快速排序的运行时间时，我们通常只关心比较操作的次数。这是因为除了比较之外，快速排序算法中的其他操作（如交换、索引移动等）所花费的时间通常只占总运行时间的一个常数因子。因此，在评估算法效率时，我们通常只考虑比较操作的次数。

• C¯ (n): erwartete Anzahl Vergleiche bei n Elementen // C¯(n) 表示对 n 个元素进行快速排序时，所期望的比较次数。这里的“期望”是基于枢轴元素随机选择的情况下的平均值。快速排序的性能会受到枢轴元素选择的影响，如果枢轴元素接近数组的中位数，那么划分操作会相对平衡，导致递归调用更加均衡，从而减少比较次数。相反，如果枢轴元素选择得很差（例如总是选择最大或最小的元素），那么划分操作会很不平衡，导致递归调用深度增加，从而增加比较次数。

使用随机枢轴元素是一种常用的优化策略，它可以显著减少算法在最坏情况下的性能。通过随机选择枢轴元素，我们可以期望在平均情况下得到相对平衡的划分，从而得到较好的性能。

**Satz**

**Gegeben sei eine Sequenz e1, . . ., en mit n > 1. Die erwartete Anzahl von Vergleichen für QuickSort mit zufällig ausgewähltem Pivot Element ist //期望比较次数的上界**

**C¯ (n) ≤ 2n ln n**

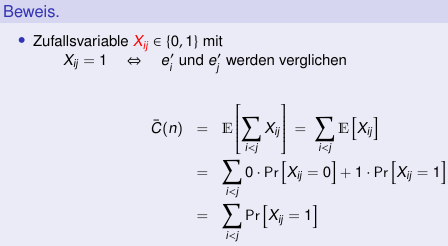
Beweis：

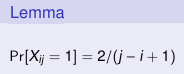
• betrachte sortierte Sequenz ⟨e ′ 1 , . . . , e ′ n ⟩

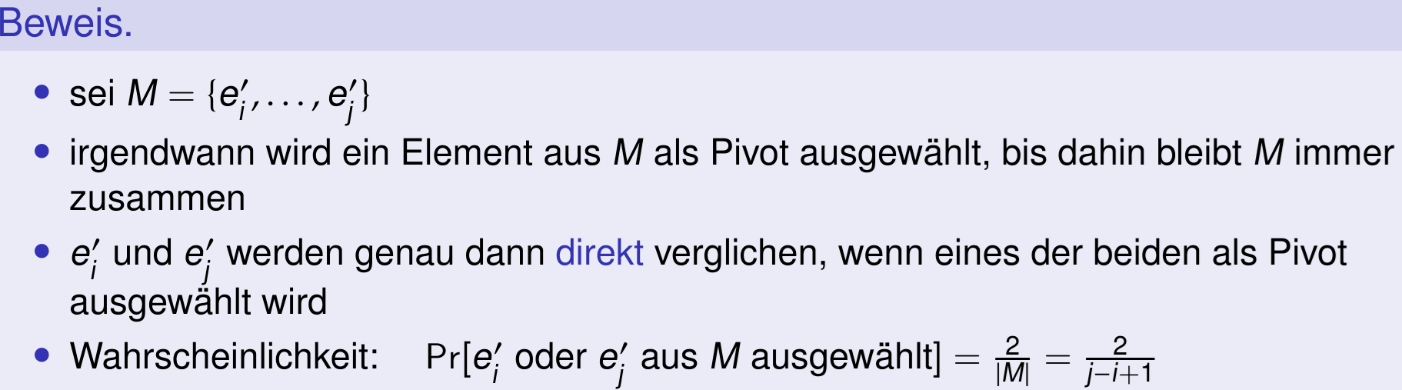
• es gibt nur Vergleiche mit Pivot Element

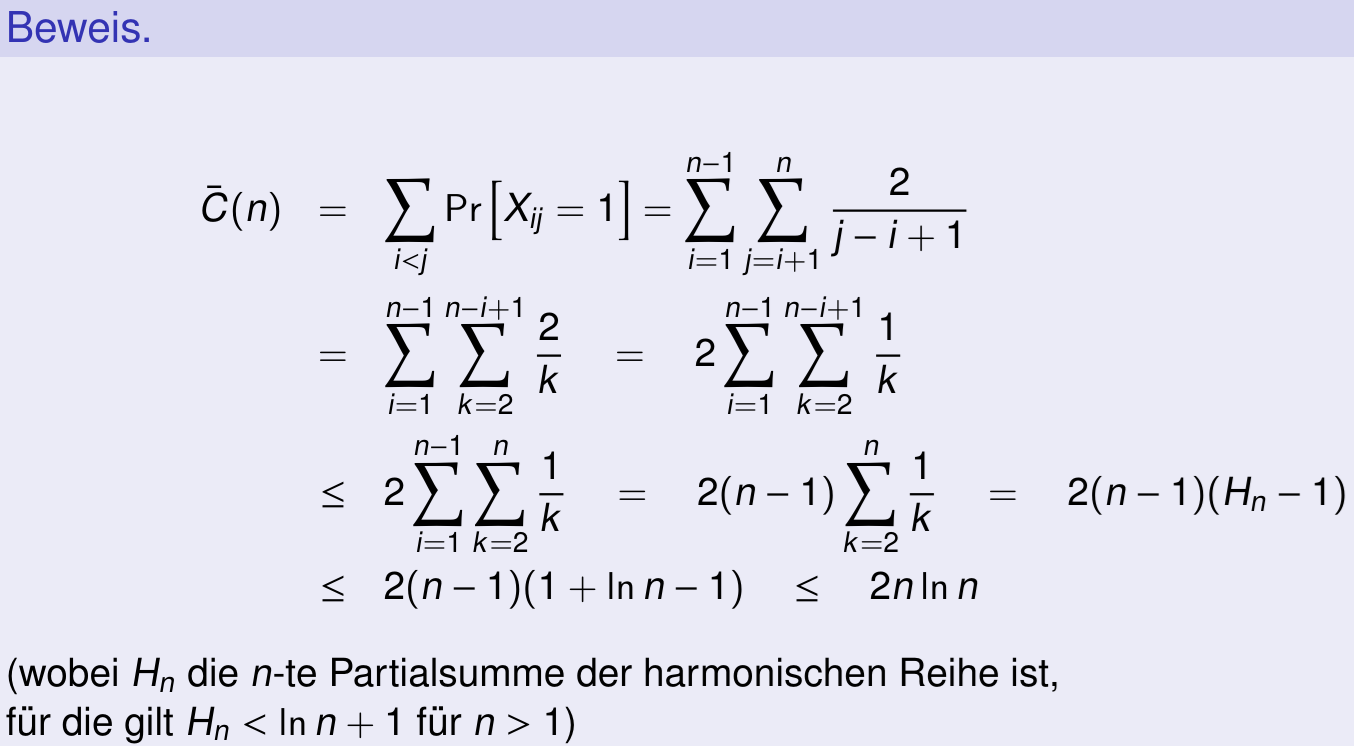
• Pivot Element ist nicht in den rekursiven Aufrufen enthalten

⇒ zwei Elemente e ′ i und e ′ j werden also höchstens einmal verglichen und zwar dann, wenn e ′ i oder e ′ j Pivot Element ist

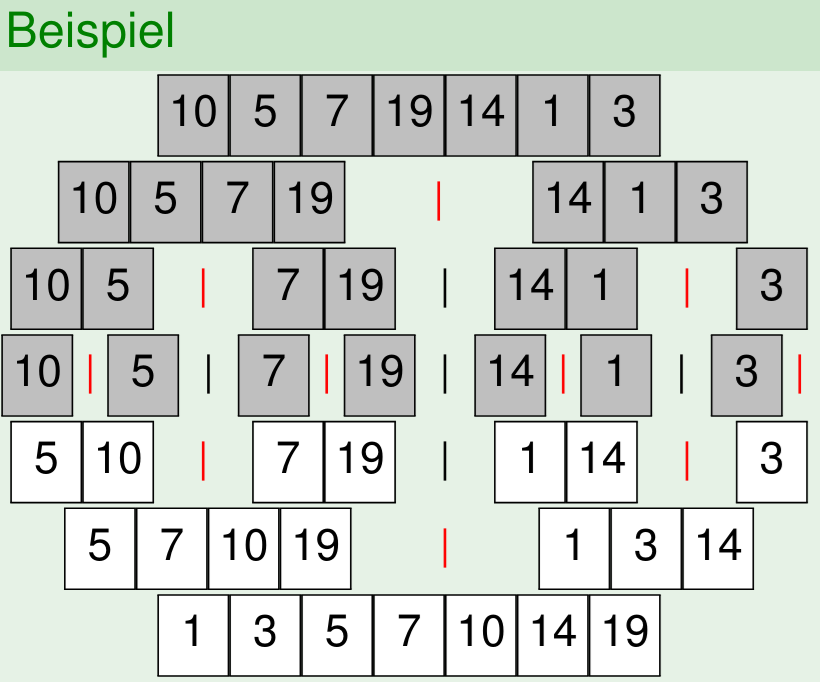








* 1. **MergeSort (Sort nach Verschmelzen)**

Zerlege die Eingabesequenz rekursiv in Teile, die separat sortiert und dann zur Ausgabesequenz verschmolzen werden递归地将输入序列分解为多个部分，分别排序，然后合并到输出序列中

Implementierung：

public void mergeSort (Element [] arr, int leftSide, int rightSide) {

if (leftSide == rightSide) return; // 只有一个元素，无需排序，直接返回

int midPos = ⌊ (rightSide + leftSide) /2⌋; // 计算中间索引，也可以写成lefSid + (rigSid - lefSide)/2

mergeSort (arr, leftSide, midPos); // 递归排序左半边

mergeSort (arr, midPos + 1, rightSide); // 递归排序右半边

// 合并两个已排序的子数组, 这里需要一个临时数组tempArr，但在伪代码中未声明

Element [] tempArr = new Element [rightSide - leftSide + 1]; // 假设Element是一个已定义的类型

int leftArrIdx = leftSide; // 左子数组的起始索引

int rightArrIdx = midPos + 1; // 右子数组的起始索引

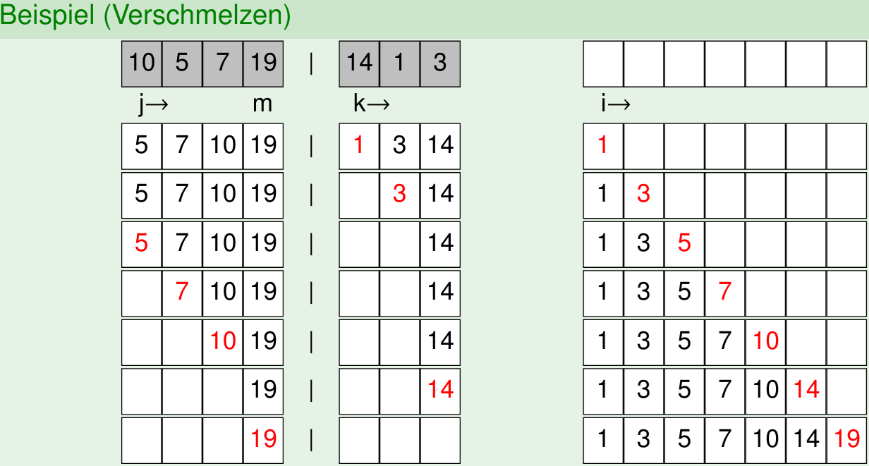
int i = 0; // 临时数组tempArr的索引

// 合并过程

for (i = 0; i < rightSide - leftSide + 1; i++) { // 注意循环条件，确保遍历整个子数组

if (leftArrIdx > midPos) {

tempArr[i] = arr[rightArrIdx];

 rightArrIdx++;

} // 左边已遍历完，将右边剩余元素加入tempArr

else if (rightArrIdx > rightSide) {

tempArr[i] = arr[leftArrIdx];

leftArrIdx++;

} // 右边已遍历完，将左边剩余元素加入tempArr

else if (arr[leftArrIdx] <= arr[rightArrIdx]) {

tempArr[i] = arr[leftArrIdx];

leftArrIdx++;

} // 左边当前元素小于等于右边，将左边元素加入tempArr

else {

tempArr[i] = arr[rightArrIdx];

rightArrIdx++;

} // 否则，将右边元素加入tempArr

}

// 将合并后的数组tempArr拷回原数组arr

for (i = 0; i < rightSide - leftSide + 1; i++) {

arr[leftSide + i] = tempArr[i];}}

Zeitaufwand:

• T(n): Laufzeit bei Feldgröße n

• T (1) = Θ (1)

T(n) = T(⌈n/2⌉) + T(⌊n/2⌋) + Θ(n)

⇒ T(n) ∈ O (n log n) (folgt aus dem Master-Theorem, nächstes Kapitel)

Speicheraufwand:

• O(n) zusätzlicher Speicher benötigt für verschmelzen

* 1. **Master-Theorem**

**1.4.1Intro:**

• Nicht-rekursive Unterprogrammaufrufe sind einfach zu analysieren //非递归子程序调用易于解析

• separate Analyse des Funktionsaufrufs und Einsetzen //单独的函数调用分析和插入

• Rekursive Aufrufstrukturen liefern Rekursionsgleichungen //递归调用结构提供递归方程

• Funktionswert wird in Abhängigkeit von Funktionswerten für kleinere Argumente bestimmt //函数值取决于较小参数的函数值

•Gesucht: nicht-rekursive / geschlossene Form der Rekursionsgleichung, z.B. via Master-Theorem//递归方程的非递归/封闭形式，例如通过主定理

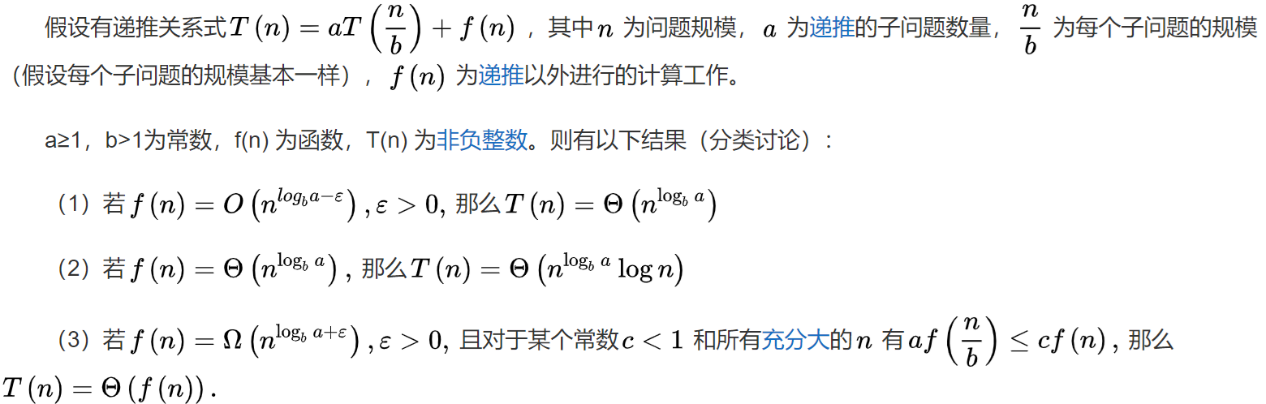
Erweiterung: 主定理 Master Theorem

E101: 简述: 在算法分析中，Master Theorem 提供了一种用渐近符号（大O符号）表示许多由分治法（Divide-and-Conquer）得到的递推关系式的方法。这种方法最初由Jon Bentley、Dorothea Haken 和 James B. Saxe 在1980年提出，并在经典算法教科书《算法导论》（Introduction to Algorithms）中得到了推广。

E102: 应用场景

Master Theorem 适用于分析分治法产生的递推关系式。分治法的基本思想是将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。在这个过程中，通常会涉及到递归调用，而这些递归调用会产生递推关系式。

E103: 形式



**1.4.2 Anwendung von Divide-and-Conquer-Algorithmen**

• Gegeben: Problem der Größe n = b^k (mit k ∈ N0 und b > 0)

• Falls k ≥ 1:

• Zerlege das Problem in d Teilprobleme der Größe n/b

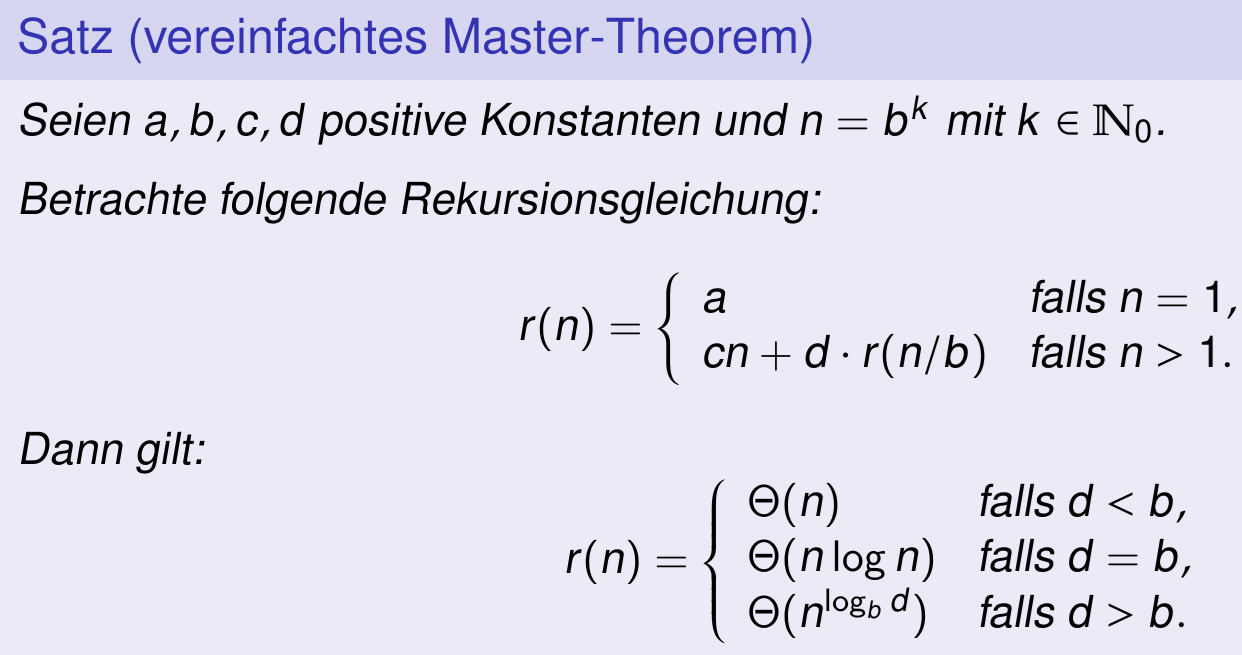
• Löse die Teilprobleme (d rekursive Aufrufe)

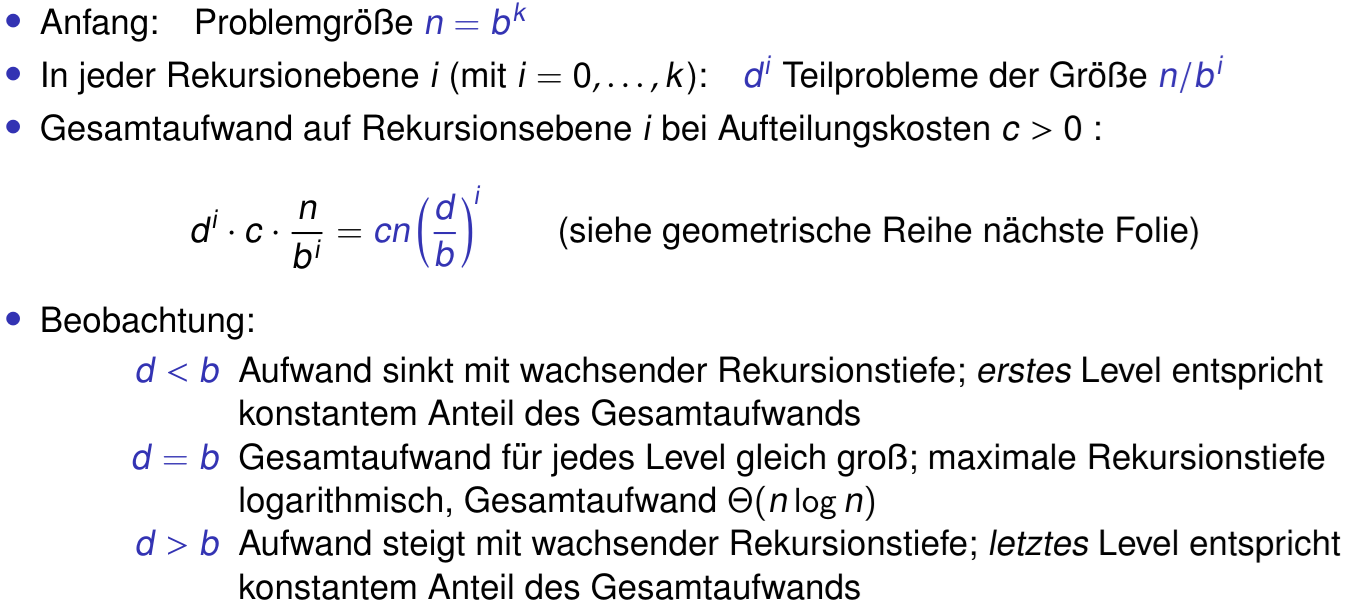
• Setze aus den Teillösungen die Lösung zusammen

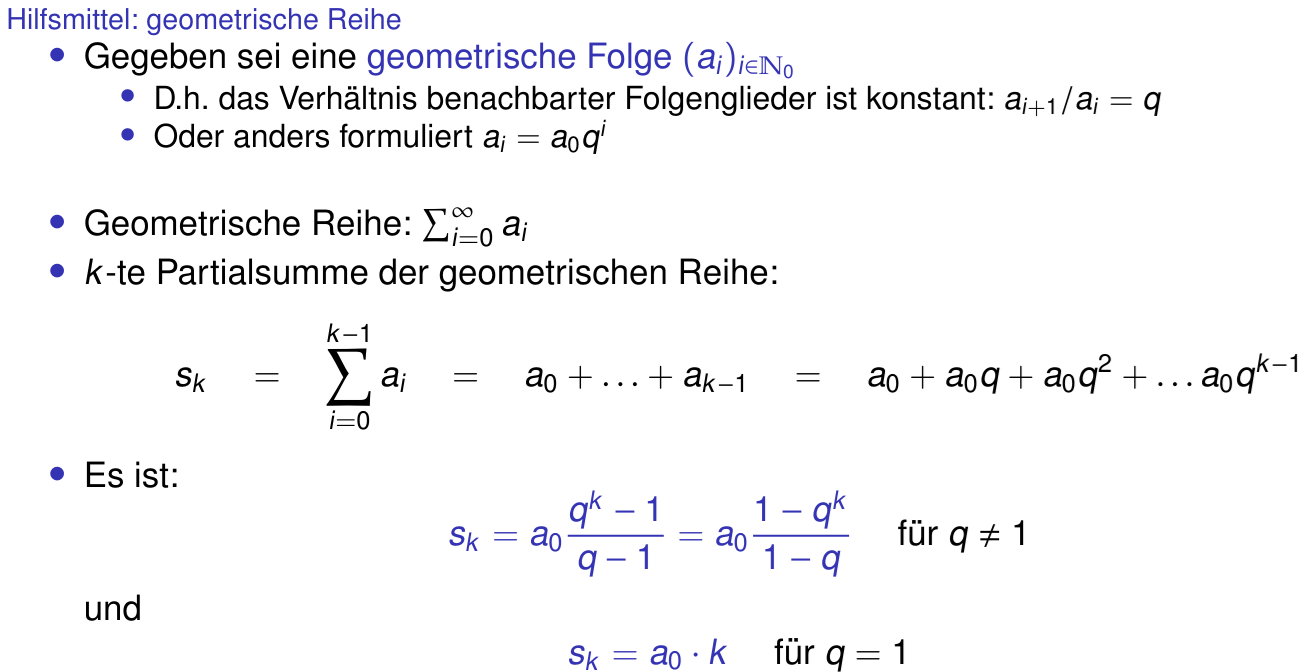
• Falls k = 0 bzw. n = 1: investiere Aufwand a zur Lösung

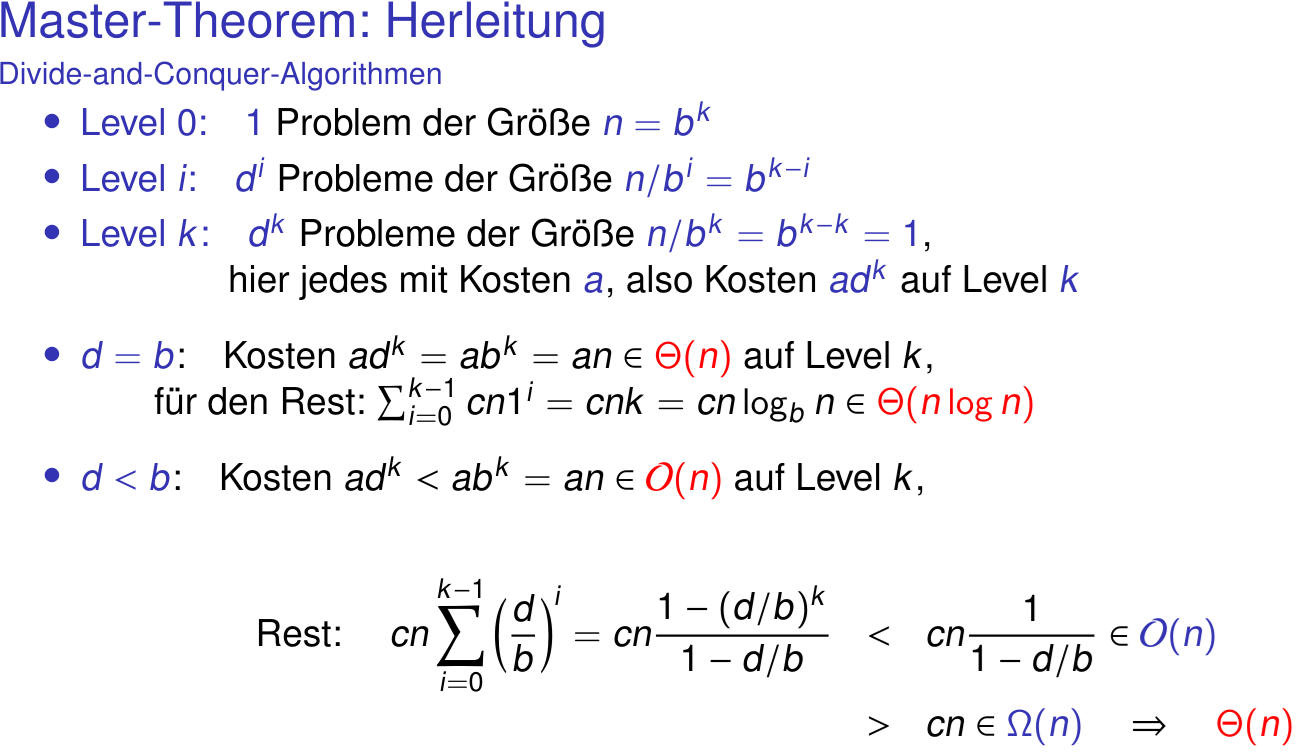
Beispiel: MergeSort mit d = b = 2

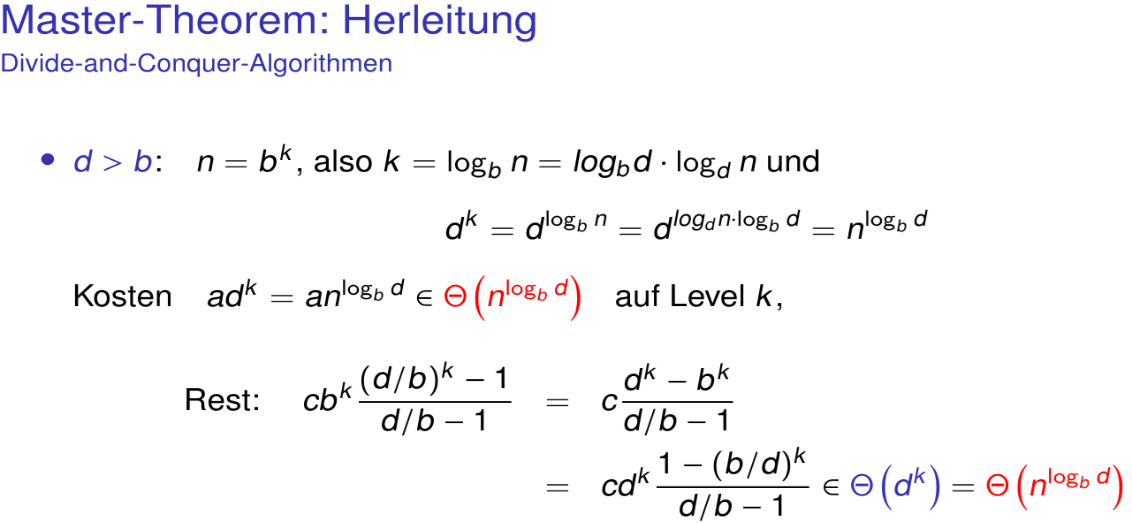
1.**4.3 Satz und Lösung von Rekursionsgleichungen**











* 1. **Untere Schranke und Herleitung(Vergleichsbasiertes Sortieren)下界和导数**

• MergeSort hat Laufzeit O(n log n) im worst-case

• InsertionSort hat im best-case lineare Laufzeit O(n)

Ü: Gibt es Sortierverfahren mit Laufzeit besser als O(n log n) im worst-case? (z.B. O(n) oder O(n log log n))

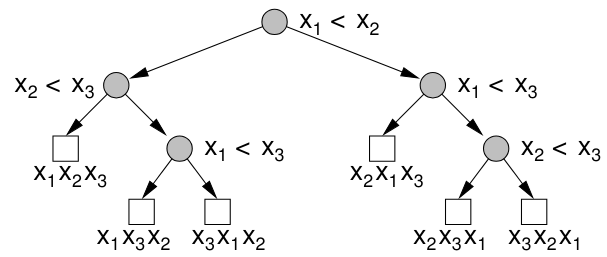
⇒ **Nein**, nicht auf der Basis einfacher Schlüsselvergleiche! (i.e. Entscheidungen: xi < xj → ja / nein)

**Satz**

**Jeder vergleichsbasierte Sortieralgorithmus benötigt im worst-case mindestens**

**n log n − O(n) ∈ Θ(n log n) Vergleiche.**

• Beispiel: Folge mit drei Elementen x1, x2, x3

• Entscheidungsbaum mit Entscheidungen an den Knoten:

• die untere Schranke muss insbesondere auch gelten, wenn alle n Schlüssel verschieden sind ⇒ Annahme: alle verschieden

• wie viele verschiedene Ergebnisse gibt es beim Sortieren? ⇒ alle Permutationen:

• Binärbaum der Höhe h hat höchstens 2^h Blätter bzw. Binärbaum mit b Blättern hat mindestens Höhe log2 b

* 1. **Schnelleres Sortieren (Bsp. Radix Sort)**

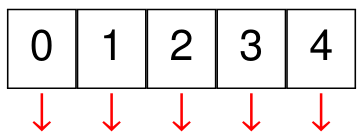
•Laut Untere Schranke und Herleitung wissen wir, dass Sortieren nie besser als Θ(n log n)

• Was aber, wenn die Schlüsselmenge mehr Struktur hat? (z.B. Zahlen / Strings bestehend aus mehreren Ziffern / Zeichen)

• um zwei Zahlen / Strings zu vergleichen reicht oft schon die erste Ziffer / das erste Zeichen

• nur bei gleichem Anfang kommt es auf mehr Ziffern / Zeichen an

• Annahme: Elemente sind Zahlen im Bereich {0, . . . ,K − 1}

• Strategie: verwende Feld von K Buckets (z.B. Listen)

Buckets bei Sortierung

Sequence kSort(Sequence s) {

Sequence [] b = new Sequence[K];

foreach (e ∈ s)

b[key(e)].pushBack(e);

return concatenate(b); // Aneinanderreihung von b[0],. . . ,b[k-1] }

Laufzeit: Θ(n + K) Problem: nur gut für K ∈ o(n log n)

Speicher: Θ(n + K)

• Wichtig: kSort ist stabil, d.h. Elemente mit dem gleichen Schlüssel behalten ihre relative Reihenfolge ⇒ Elemente müssen im jeweiligen Bucket hinten angehängt werden