Rechtlicher Hinweis

Diese Präsentation ist urheberrechtlich geschützt und darf nur im Rahmen von Lehrveranstaltungen der Friedrich-Schiller-Universität Jena verwendet werden. Eine Nutzung durch Verbreitung oder Veröffentlichung dieses Materials - auch in Auszügen - ist strengstens untersagt und wird die Geltendmachung von Unterlassungs- und Schadenersatzansprüchen durch die Friedrich-Schiller-Universität Jena zur Folge haben.

Legal notice

These slides are protected by copyright and may only be used as part of courses at the Friedrich Schiller University Jena. Any use through the dissemination or publication of this material - even in extracts - is strictly prohibited and will result in the assertion of injunctive relief and claims for compensation by the Friedrich Schiller University Jena.

Informatik I (B.Sc. Physik)

Algorithmen

Dr. Paul Bodesheim
(Paul.Bodesheim@uni-jena.de)

FRIEDRICH-SCHILLER-UNIVERSITÄT JENA

Fakultät für Mathematik und Informatik
Lehrstuhl für Digitale Bildverarbeitung

SoSe 2020

Inhalt

Inhalt

Algorithmen

Algorithmus
Eindeutige Handlungsvorschrift zur Lösung eines Problems oder einer Klasse von Problemen.

Informell: wie ein exaktes Kochrezept

算法
解决一个问题或一类问题的明确准则。
非正式的：像一个确切的食谱
目标：了解，开发和评估典型算法
什么时候该算法对大量数据可行？
您如何加速算法？
有理论上的限制吗？

- Ziel: Kennenlernen, Entwickeln und Bewerten typischer Algorithmen
- Wann ist ein Algorithmus für große Datenmengen praktikabel?
- Wie kann man einen Algorithmus beschleunigen?
- Gibt es theoretische Grenzen?

Eigenschaften von Algorithmen

已终止：算法在有限数量的步骤后结束
确定性的：在每个时间点都明确定义了流程
确定：算法为相同的输入提供相同的结果

正确或有效：该算法可提供理想的结果
高效：算法需要很少的时间和/或存储空间

- **terminiert**: Algorithmus endet nach endlichen vielen Schritten
- **deterministisch**: Ablauf ist zu jedem Zeitpunkt eindeutig definiert
- **determiniert**: Algorithmus liefert bei gleichen Eingaben das gleiche Ergebnis
- **korrekt oder effektiv**: Algorithmus liefert das gewünschte Ergebnis
- **effizient**: Algorithmus benötigt wenig Zeit und/oder Speicherplatz

Terminiertheit

指数函数的计算

非终止计算规则
只能以一定程度的准确性进行测定
经过有限步骤后终止

- Berechnung der Exponentialfunktion

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

- Nicht-terminierende Berechnungsvorschrift
- Bestimmung nur bis zu einer gewissen Genauigkeit möglich
- Abbruch nach endlich vielen Schritten

Determinismus

- Nicht-deterministische Algorithmen durch Verwendung von Zufallszahlen

使用随机数的非确定性算法
例子：
随机初始解（例如牛顿法）
机器学习方法（通常称为人工智能方法）
蒙特卡洛方法基于随机数
每个确定性算法都已确定，但事实并非如此！

- Beispiele:
  - Zufällige Startlösungen (z.B. beim Newtonverfahren)
  - Verfahren des maschinellen Lernens (üblicherweise bekannt unter Verfahren der künstlichen Intelligenz)
  - Monte-Carlo Verfahren basieren auf Zufallszahlen
- Jeder deterministische Algorithmus ist auch determiniert, Umkehrung gilt aber nicht!

Bestimmung von Flächeninhalten

- Aufgabenstellung: Bestimmung der Fläche A unter einer Kurve $f : \mathbb{R} \to \mathbb{R}$
- Verfahren:

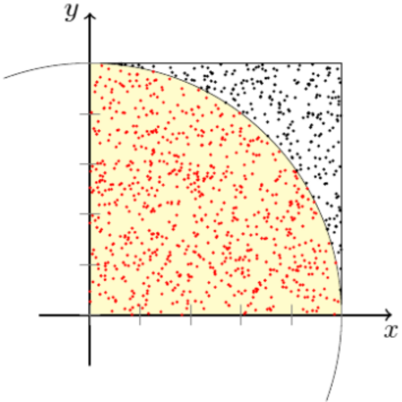
  1. Definiere eine Fläche Ω mit bekannten Flächeninhalt, welche A einschließt ($A \subset \Omega$)
  2. Ziehe zufällig n Punkte x_i von Ω
  3. Bestimme wieviele der Punkte x_i in A liegen

- Grundidee: empirische Wahrscheinlichkeit liefert relativen Flächeninhalt

$$\frac{|A|}{|\Omega|} \approx \frac{\text{Anzahl der gezogenen Punkte, welche in A liegen}}{\text{Anzahl der gezogenen Punkte}} = \frac{|\{x_i \in A\}|}{n}$$

- Beispiel: $sin(x)$

Bestimmung von π mit Monte-Carlo Verfahren



Monte-Carlo-Algorithmus zur Bestimmung von π

```
#include <iostream>
#include <stdlib.h>
using namespace std;

int main()
{
    int n=1000000;
    int p=0;

    for (int i = 0; i < n; i++)
    {
        // Ziehe zufaelligen Punkt mit 0 <= x,y <= 1
        double x = drand48();
        double y = drand48();
        // Quadratischer Abstand vom Kreismittelpunkt
        double d = x*x + y*y;

        if (d<=1)
            p++;
    }
    cout << "Pi ist etwa: " << 4.0*p/n << endl;
}
```

Randomisierte Algorithmen

- Basieren auf Zufallszahlen / dem Zufallsprinzip
- Zwei Klassen von randomisierten Algorithmen

Monte-Carlo-Algorithmen

- Feste/konstante Laufzeit (oftmals effizient)
- Ergebnis exakt mit gewisser Wahrscheinlichkeit, Ergebnis ist Zufallsvariable (nicht notwendig korrekt, Schätzwert, gewisse Fehlerwahrscheinlichkeit),
- Beispiel: Flächenbestimmung, Schätzung für π

Las-Vegas-Algorithmen

- Zufällige Laufzeit (ggf. nicht terminierend), Laufzeit ist Zufallsvariable
- Liefert immer exaktes Ergebnis, stets korrekt
- Beispiel: Bogo Sort (auch Random Sort oder Stupid Sort)

Inhalt

Entwurf von Algorithmen

结果的正确性不是唯一的目标！
效率对可用性至关重要。
运行效率
内存需求方面的效率
在运行时间和内存需求之间通常需要权衡

- Korrektheit des Ergebnisses ist nicht das einzige Ziel!
- Effizienz kann für Nutzbarkeit entscheidend sein:
  - Laufzeiteffizienz
  - Effizienz bezüglich des Speicherbedarfes
- Es gibt oft einen Kompromiss (trade-off) zwischen Laufzeit und Speicherbedarf

Laufzeit eines Algorithmus

- Wie lange benötigt ein Algorithmus zur Bestimmung der Lösung?
- Möglichkeit 1: Zeit stoppen
  - Unter Linux mit dem Befehl: `time` (`real` gibt die reale Laufzeit an)
  - In C++: Funktion *gettimeofday()* mit Genauigkeit in Mikrosekunden
  - Abhängig von Hardware und anderen aktiven Programmen, etc.
  - Laufzeiten schwer vergleichbar

```
time estimate-pi
Pi ist etwa: 3.14252

real   0m0.020s
user   0m0.020s
sys    0m0.000s
```

算法需要多长时间来确定解决方案？
选项1：停止时间
在Linux下，使用以下命令：time（real表示实际运行时）
在C ++中：函数gettimeofday（）的精度为微秒
取决于硬件和其他活动程序等
运行时间很难比较

Laufzeit eines Algorithmus

- Möglichkeit 2: Zählen von Operationen
- Operationen:
  - Einzelne Rechenoperation (+,-,*,/)

    (flops: floating point operations per second)
  - Vergleichsoperationen (==,<,>)
  - Zugriff auf Feldelemente
  - ...
- Laufzeit in Abhängigkeit von der Eingabegröße n

情况二：计数操作
操作方式：
单算术运算（+，-，*，/）
（触发器：每秒浮点运算）
比较运算（==，<，>）
访问字段元素

运行时间与输入量n的关系

Inhalt

Algorithmus: Suche von Elementen

- Typische Aufgabenstellung in der Informatik: Suchen von Elementen in einer Liste
  - Gibt es Daten eines bestimmten Kunden in einer Datenbank?
  - Suche nach einer Telefonnummer
  - Suche nach einer Datei mit einem gewissen Namen
  - ...
- Gegeben: Feld mit Elementen $x_0, \ldots, x_{n-1}$
- Gesucht: ist der Wert z in der Liste enthalten? (und eventuell wo?)

计算机科学中的典型任务：搜索元素
在清单中
数据库中是否有来自特定客户的数据？
搜索电话号码
查找具有特定名称的文件
。。 。
给定：元素为x0的字段； :::; xn − 1
想要：值z在列表中吗？ （可能在哪里？）

Vollständige Suche

- Vollständige Suche oder Sequentielle Suche:
  Nacheinander alle Elemente des Feldes überprüfen

```
vector<int> v(n);
// Vektor v mit Werten fuellen
// z.B. Lesen aus Datei
...
int z = 0;
cin >> z;

bool found = false;
for (int i = 0; i < v.size() && !found; i++)
{
  if (v[i] == z)
     found = true;
}
...
```

Analyse der Laufzeit

- Ungünstigster Fall (**worst case**): Durchlauf des ganzen Feldes, n Feldzugriffe
- Bester Fall (**best case**): Erfolg beim ersten Element, 1 Feldzugriff
- Mittlere Laufzeit (**average case**):
  - Element ist enthalten: $\frac{n}{2}$ Feldzugriffe
  - Element ist nicht enthalten: n Feldzugriffe
- Wie oft ist der gesuchte Wert enthalten?
- **worst case analysis** ist oft einfacher durchzuführen

最坏的情况：遍历整个字段，n
现场访问
最好的情况：第一个要素是成功，可以进行一次现场访问
平均运行时间（平均情况）：
包含的元素：n
2个现场访问
不包含元素：n个字段访问
您寻找的价值多久包含一次？
最坏情况的分析通常更容易进行

Gibt es ein besseres Suchverfahren?

- Vollständige Suche funktioniert, aber ist sie auch effizient genug?
- . . . auch bei Millionen von Datenelementen?
- Gibt es einen Algorithmus der weniger als *n* Feldzugriffe und Vergleiche benötigt?
- Antwort: **ja, ABER nur** mit Vorwissen
- Beispiel: Werte sind sortiert

完全搜索有效，但是效率足够吗？
。。。即使有数百万个数据元素？
是否有一种算法的字段访问少于n次，并且
需要比较吗？
答：是的，但仅具有先验知识
示例：值已排序

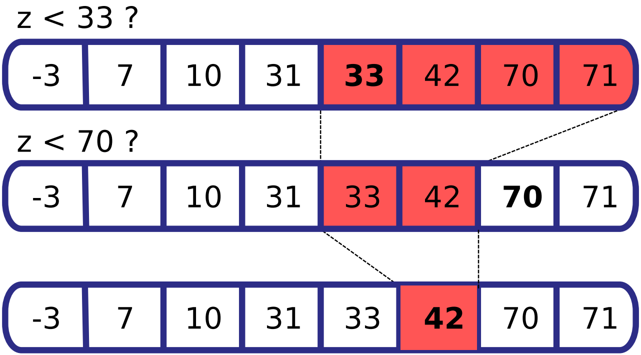
Binäre Suche

字段排序x0≤x1≤:::≤xn － 1
查询数字可指示您要查找的数字是之前还是之后
数字在于
想法：通过查询平均值将搜索区域减半
！ 二进制搜索
原理类似于猜测0到100之间的随机数
迭代问题：值z在字段的哪一半中？

- Feld ist sortiert $x_0 \leq x_1 \leq \ldots \leq x_{n-1}$
- Abfrage einer Zahl liefert Hinweis, ob die gesuchte Zahl vor oder nach der Zahl liegt
- Idee: Halbierung des Suchraumes durch Abfrage des mittleren Wertes

  $\rightarrow$ **Binäre Suche**
- Prinzip ähnlich zum Erraten einer zufälligen Zahl zwischen 0 und 100
- Iterative Fragestellung: In welcher Hälfte des Feldes liegt der Wert z?

Binäre Suche: Beispiel



Binäre Suche: Konzept

算法步骤：
1选择整个字段作为搜索区域
2检查中间元素：
相同的z：完成
小于z：在右侧子区域中进一步搜索
大于z：在左侧子区域中进一步搜索
3如果未找到并且区域不为空，请返回步骤2

- Ablauf des Algorithmus:
  1. Auswahl des ganzen Feldes als Suchbereich
  2. Überprüfen des mittleren Elementes:
     - gleich z: fertig
     - kleiner als z: weitere Suche im rechten Teilbereich
     - größer als z: weitere Suche im linken Teilbereich
  3. Wenn nicht gefunden und Bereich nicht leer zurück zu Schritt 2

Algorithmus in C++

```
bool binSearch(const vector<int> &x, int z)
{
  int beginn = 0;
  int ende = x.size() - 1;

  bool gefunden = false;

  while ((!gefunden) && (ende >= beginn))
    {
      int mitte = (beginn + ende) / 2;
      if (z == x[mitte])
        gefunden = true;
      else if (z < x[mitte])
        ende = mitte - 1;
      else
        beginn = mitte + 1;
    }
  return gefunden;
}
```

Laufzeit

Warum ist die binäre Suche schneller als die vollständige?

- In jedem Schritt (k) wird der Suchbereich halbiert:

$$n_k = \frac{n}{2^k}$$

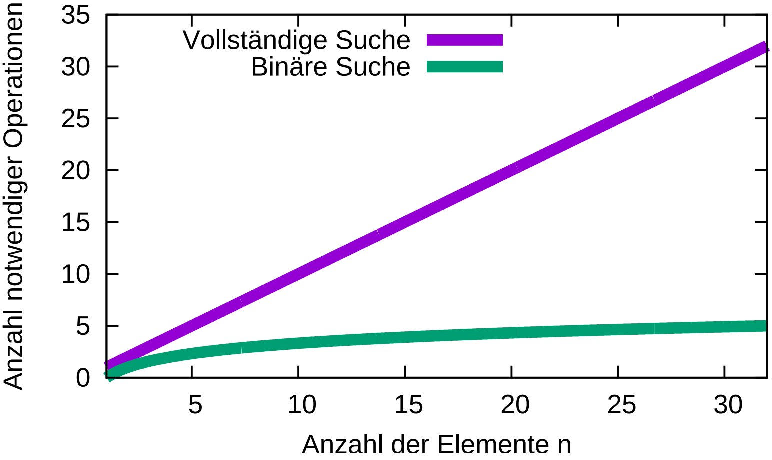
- Algorithmus terminiert, wenn $n_k == 1$

$$\frac{n}{2^k} = 1$$

$$k = \log_2(n)$$

- $\approx \log_2(n)$ Feldzugriffe notwendig (Logarithmus zur Basis 2)

Laufzeitvergleich



Inhalt

Bewertung der Laufzeit

- Laufzeitbewertung: Anzahl der Berechnungsschritte/Operationen $f(n)$ in Abhängigkeit von der Größe n der Eingabe

- Bewertung/Laufzeit der einzelnen Berechnungsschritte abhängig von:
  - Maschine
  - Programmierung
  - ...

- $A(n)$: Anzahl der Feldzugriffe bei Feldgröße n

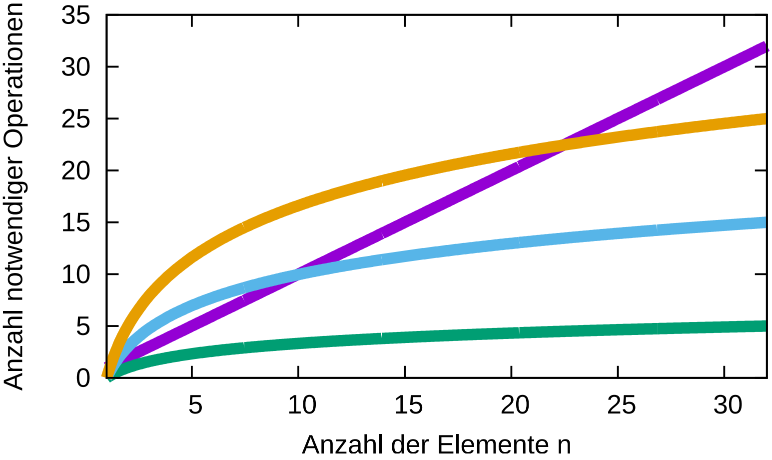
- Anzahl Berechnungsschritte/Operationen pro Feldzugriff: c_1

$$\Rightarrow \; f(n) = c_1 * A(n)$$

- Dies entspricht einem konstanten Faktor!

运行时评估：计算步骤/操作数
f（n）作为输入大小n的函数
各个计算步骤的评估/持续时间取决于：
机
程式设计
。。。
A（n）：字段大小为n的字段访问次数
每个字段访问的计算步骤/操作数：c1
）f（n）= c1 * A（n）
这相当于一个常数！

Asymptotische Laufzeit



Definition: Asymptotische Laufzeit (Komplexität)

考虑大量输入的运行时间
不影响乘法常数和加法常数
Gross-O / Landau表示法：f与简单的比较
比较功能g
当且仅当f 2 O（g）
9c> 0 9n0> 0 8n≥n0：f（n）≤c g（n）
从输入的确定大小n0开始，运行时复杂度为
f（n）受c · g（n）限制

- Betrachtung der Laufzeit für große Eingabemengen
- Kein Einfluss von multiplikativen und additiven Konstanten
- Gross-O/Landau Notation: Vergleich von f mit einfachen Vergleichsfunktionen g

$$f \in O(g) \quad \text{genau dann wenn}$$
$$\exists c > 0 \; \exists n_0 > 0 \; \forall n \geq n_0 : \quad f(n) \leq c \cdot g(n)$$

- Ab einer bestimmten Größe n_0 der Eingabe ist die Laufzeitkomplexität $f(n)$ nach oben durch $c \cdot g(n)$ beschränkt

Zusätzliche Erläuterung

乘法常数无关紧要！
可加常数无关紧要！
曲线f（n）的一般过程很重要，而不是精确的价值观。
f的增长至关重要。
Landau表示法可以灵活定义
输入数据：字段元素数还是字段数
所需位的计数无关紧要。

- Multiplikative Konstanten sind irrelevant!
- Additive Konstanten sind irrelevant!
- Der allgemeine Verlauf der Kurve $f(n)$ zählt und nicht die genauen Werte.
- Das Wachstum von f ist entscheidend.
- Landau Notation erlaubt eine flexible Definition der Größe der Eingabedaten: Ob die Anzahl der Feldelemente oder die Anzahl der benötigten Bits gezählt wird, ist irrelevant.

Typische Klassen der Laufzeitkomplexität

1. $O(1)$: konstante Laufzeit
2. $O(\log n)$: logarithmische Laufzeit
3. $O(n)$: lineare Laufzeit
4. $O(n \log n)$: "quasi lineare"/"fast lineare" Laufzeit
5. $O(n^2), O(n^3), \ldots$: quadratische, kubische, ... Laufzeit
6. $O(n^t), t \geq 1$: polynomielle Laufzeit
7. $O(a^n), a \geq 1$: exponentielle Laufzeit

Es gibt weitere Komplexitätsklassen, auch bezüglich des Speicherplatzbedarfs
(Komplexitätstheorie als Teilgebiet der theoretischen Informatik)

Beispiele

- Binäre Suche: $c_1 \cdot \log(n) \in O(\log(n))$, d.h. logarithmische Laufzeit
- Vollständige Suche: $c_2 \cdot n \in O(n)$, d.h. lineare Laufzeit
- Bestimmung des Maximums oder Minimums ...
- Bestimmung aller paarweiser Abstände zwischen n Punkten ...
- Bestimmung des Histogrammes eines Bildes ...

Wachstum von Laufzeiten

$f(n)$	$n = 2$	$2^4 = 16$	$2^8 = 256$	$2^{10} = 1024$	$2^{20} = 1048576$
$\log(n)$	1	4	8	10	20
n	2	16	256	1024	1048576
$n \cdot \log(n)$	2	64	2048	10240	20971520
n^2	4	256	65536	1048576	$\boxed{\approx 10^{12}}$
n^3	8	4096	16777200	$\boxed{\approx 10^{9}}$	$\boxed{\approx 10^{18}}$
2^n	4	65536	$\boxed{\approx 10^{77}}$	$\boxed{\approx 10^{308}}$	$\boxed{\approx 10^{315653}}$

Anzahl der Atome im Weltall $\approx$ 10^{77} (nach A. Beutelspacher)

Inhalt

Sortieren von Elementen

- Eingabe: unsortiertes Feld von n Elementen: $x_1, \ldots, x_n$
- Ausgabe: sortiertes Feld
- Es existieren zahlreiche Sortieralgorithmen:
  1. Bubble Sort
  2. Selection Sort
  3. Insertion Sort
  4. Merge Sort
  5. Quick Sort
  6. Distribution Sort
  7. . . .

输入：n个元素的未排序字段：x1; :::; n
输出：排序字段
有许多排序算法：
1个气泡排序
2选择排序
3插入排序
4合并排序
5快速排序
6分布排序

Bogo Sort / Random Sort / Stupid Sort

拉斯维加斯算法示例
1创建元素的新（随机）顺序
2检查元素是否正确排序
3如果不是这种情况，请转到步骤1，否则会出现
找到解决方案
算法的开销？ 最坏的情况下？
最坏的情况：必须检查所有可能的顺序将
订单数量：n！ （n阶乘）
渐近运行时间：n！ 2 0（nn）

- Beispiel für einen Las-Vegas-Algorithmus
  1. Erzeuge eine neue (zufällige) Reihenfolge der Elemente
  2. Überprüfe ob die Elemente richtig sortiert sind
  3. Wenn dies nicht der Fall ist, gehe zu Schritt 1, ansonsten ist eine Lösung gefunden
- Aufwand des Algorithmus? Ungünstigster Fall?
- Ungünstigster Fall: Alle möglichen Reihenfolgen müssen überprüft werden
- Anzahl der Reihenfolgen: $n!$ (n Fakultät)
- Asymptotische Laufzeit: $n! \in O(n^n)$

Bubble Sort

气泡排序：
1对于该领域的所有要素
2如果它大于后继元素
3交换给他的继任者
4如果在整个运行过程中都必须进行交换，请转到步骤1
算法工作量：
每次运行必须进行n-1个比较
最多需要n-1次运行
渐近运行时间：（n-1）（n-1）2 0（n2）

- Bubble Sort:
  1. Für alle Elemente des Feldes
  2. Falls es größer als das Nachfolgeelement ist
  3. Vertausche es mit seinem Nachfolger
  4. Wenn im gesamten Durchlauf ein Tausch nötig war, gehe zu Schritt 1

- Aufwand des Algorithmus:
  - Bei jedem Durchlauf müssen $n-1$ Vergleiche durchgeführt werden
  - Es sind höchstens $n-1$ Durchläufe nötig
  - Asymptotische Laufzeit: $(n-1)\cdot(n-1) \in O(n^2)$

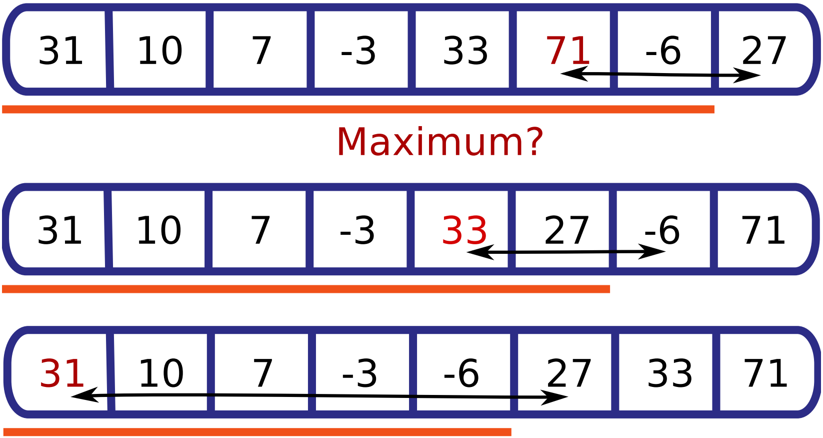
Selection Sort: Idee

确定最大值（最小值）
在新字段的末尾（开头）写上
确定剩余值的最大值（最小值）
写在上一个最大值之前（之后）
（最低）
重复直到所有元素都已使用
备用名称：“ 最大排序 ” 或 “ 最小排序 ”

- Bestimme Maximum (Minimum)
- Schreibe dieses an Ende (Anfang) eines neuen Feldes
- Bestimme Maximum (Minimum) der verbleibenden Werte
- Schreibe dieses an die Stelle vor (hinter) das vorherige Maximum (Minimum)
- Wiederhole bis alle Elemente verwendet wurden

- Alternative Bezeichnungen: Max Sort bzw. Min Sort

Selection Sort: in-place



Selection Sort: Vorgehen

确定递减子字段中的最大值
将最大值移到正确位置
确定剩余子字段中的最大值
将最大值移到子场之后的后方位置
排序的序列增长，其余子字段变小
重复直到无法分类
第一步：最大化到达后方位置
第二步：第二个最大值达到倒数第二个位置

- Bestimmung des Maximums in einem kleiner werdenden Teilfeld
- Verschieben des Maximums an die richtige Position

- Bestimmung des Maximums im verbleibenden Teilfeld
- Verschieben des Maximums auf die hintere Position nach dem Teilfeld

- Sortierte Folge wächst, verbleibendes Teilfeld wird kleiner
- Wiederholung bis nichts mehr zu sortieren ist

- Erster Schritt: Maximum gelangt an die hintere Position
- Zweiter Schritt: Zweitgrößter Wert gelangt an die vorletzte Position
  . . . u.s.w.

Analyse von Selection Sort

- In Schritt k wird das Maximum von $n-k+1$ Elementen bestimmt
- Anzahl der Gesamtschritte:

$$\begin{aligned} f(n) &= \sum_{k=1}^{n}(n-k+1) \\ &= n^2 - \frac{n(n+1)}{2} + n \\ &= \frac{2n^2 - n^2 - n + 2n}{2} \\ &= \frac{n^2+n}{2} \in O(n^2) \end{aligned}$$

alternativ:

$$\begin{aligned} f(n) &= \sum_{k=1}^{n}(n-k+1) \\ &= \sum_{\tilde{k}=1}^{n} \tilde{k} \\ &= \frac{n(n+1)}{2} \\ &= \frac{n^2+n}{2} \in O(n^2) \end{aligned}$$

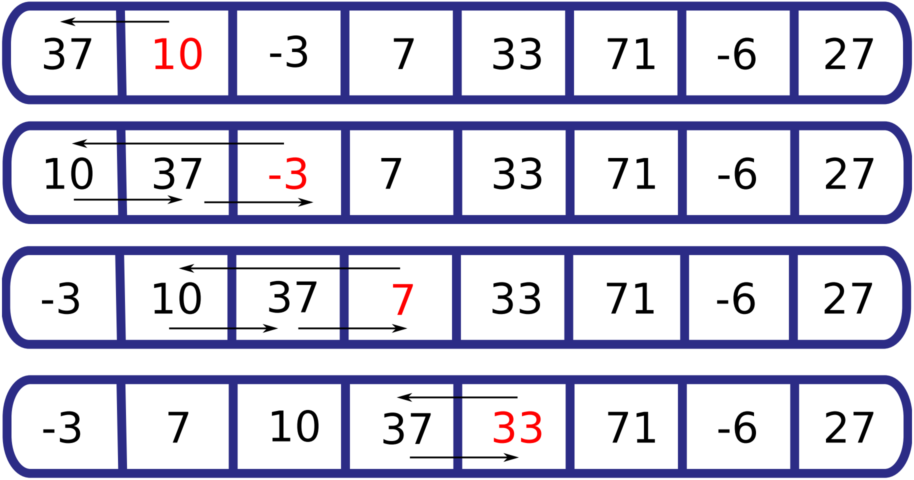
- Worst case und best case: $O(n^2)$
- Besser als Bubble Sort?

Insertion Sort: Idee

仅一次输入字段
管理排序的（子）字段
逐步将输入字段的元素插入到正确位置的排序字段中

- Durchlaufe Eingabefeld nur einmal
- Verwalte sortiertes (Teil-) Feld
- Füge nach und nach die Elemente des Eingabefeldes in das sortierte Feld **an der richtigen Position** ein

Insertion Sort: in-place



Insertion Sort: Vorgehen

Wiederhole folgenden Vorgang vom zweiten bis zum letzten Element:

1. Suchen der nächsten Position mit größerem Vorgänger
2. nicht gefunden: fertig
3. Suchen der Einfüge-Position
4. Verschieben der Nachfolger und Einfügen des aktuellen Elements
5. Gehe zu 1

Vergleich mit Selection Sort:

- Analyse schwieriger, aber worst case auch $O(n^2)$
- Best case: $O(n)$ (falls bereits sortiert)
- Benötigt Verschiebung mehrerer Elemente für Einfügen

Merge Sort

典型的“分而治之”算法
基本思路：
1将字段分为两个子字段
2通过算法的递归应用对子字段排序
3“合并”两个排序的子字段以形成整体结果

- Typischer "Teile und Herrsche" Algorithmus (divide and conquer)
- Grundidee:
  1. **Teile** das Feld in zwei Teilfelder
  2. Sortiere jeweils die Teilfelder durch **rekursive** Anwendung des Algorithmus
  3. "Mische" die zwei sortierten Teilfelder zum Gesamtergebnis

Analyse von Merge Sort

- Mischen zweier sortierter Listen benötigt $O(n)$
- Anwendung der Rekursion für die Bestimmung der Anzahl der Rechenoperationen:

$$
\begin{aligned}
f(n) &= \underbrace{2 \cdot f\left(\frac{n}{2}\right)}_{\text{Sortieren der Teilfelder}} + \underbrace{c \cdot n}_{\text{Mischen der sortierten Teilfelder}} \\
&= 2\left(2 \cdot f\left(\frac{n}{4}\right) + c \cdot \frac{n}{2}\right) + c \cdot n
\end{aligned}
$$

$$
\begin{aligned}
f(n) &= 4f\left(\frac{n}{4}\right) + 2c \cdot n \\
&= 8f\left(\frac{n}{8}\right) + 3c \cdot n \\
&= 16f\left(\frac{n}{16}\right) + 4c \cdot n \\
&= n \cdot f(1) + \log_2(n) \cdot c \cdot n \quad \in O(n \cdot \log_2(n))
\end{aligned}
$$

Quick Sort (Idee)

- Grundidee: "Teile und Herrsche" wie bei Merge Sort
- Aufteilung in Teilfelder anhand eines Elements p (**Pivot-Element**)

- Werte $\leq p$ in linkes Teilfeld
- Werte $> p$ in rechtes Teilfeld
- Rekursives Wiederholen wie bei Merge Sort bis Teilfelder der Länge 1

- Vorteil gegenüber Merge Sort: Zusammenfügen der Teilfelder einfach durch Aneinanderhängen (kein Mischen)
- Problem: Wahl des Pivot-Elements entscheidend, ggf. Ungleichgewicht bei Größe der Teilfelder

Distribution Sort

- Annahme: die Elemente aus dem Feld haben nur einen begrenzten Wertebereich (feste Anzahl von möglichen Elementen, endliche Menge)

假设：字段中的元素只有有限的值范围（可能元素的固定数量，有限集）
基本思路：
1直方图的确定
2输出带有直方图的排序列表

- Grundidee:
  1. Bestimmung eines Histogrammes
  2. Ausgabe der sortierten Liste mit dem Histogramm

Distribution Sort: Beispiel

4	0	1	2	0	3	1	4

0: 2 Elemente
1: 2 Elemente
2: 1 Element
3: 1 Element
4: 2 Elemente

Histogramm

0 0 1 1 2 3 4 4

Sortierte Liste

Analyse von Distribution Sort

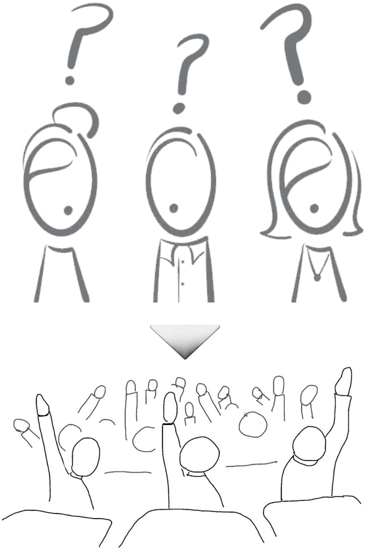
- Bestimmung des Histogrammes: $O(n)$
- Ausgabe der Liste mit dem Histogramm: $O(n)$
- Insgesamt für worst und best case: $O(n)$
- Speicherbedarf ist abhängig vom Wertebereich
- Nicht geeignet für allgemeine `int`, `float`, ... Werte

Theoretische Grenzen und weitere Anmerkungen

基于两个元素比较的算法至少需要渐近运行时间才能进行排序
的n个日志（n）
诸如“合并排序”或“快速排序”之类的算法始终可以实现此目的
（最坏的情况下）。
此外：并行排序算法
原位或就地算法不会创建新字段，而是直接在输入字段中工作

- Algorithmen, die auf dem Vergleich zweier Elemente basieren, benötigen für das Sortieren mindestens eine asymptotische Laufzeit von $n \cdot \log(n)$
- Algorithmen wie Merge Sort oder Quick Sort erreichen dies immer (worst case).
- Weiterhin: Algorithmen für paralleles Sortieren
- **in-situ** oder **in-place** Algorithmen legen kein neues Feld an, sondern arbeiten direkt im Eingabefeld

Gibt es Fragen? (Es gibt keine dummen Fragen!)



"Excuse me, is this the Society for Asking Stupid Questions?"