Theoretische Informatik 1

Wintersemester 2012/13

Herzlich willkommen!

Worum gehts? (1/2)

- Was ist machbar?
  - Welche algorithmischen Probleme kann man mit welchen Methoden effizient lösen?

Effizient = schnell

- Und was nicht?
  - Welche Probleme erlauben keine effizienten Lösungen?
  - Welche Probleme lassen sich selbst dann nicht lösen, wenn uns Laufzeit und Speicherplatz egal sind?
- Und wenn ein Problem schwierig ist, was dann?

- Was ist machbar?
  - **Probleme mit schnellen Algorithmen**: Sortieren, Kürzeste Wege, Minimale Spannbäume, String Matching, paarweises Alignment, Scheduling,.....
  - **Mächtige Entwurfsmethoden**: Greedy Algorithmen, Divide & Conquer, dynamisches Programmieren.
- Und was nicht?
  - NP-Vollständigkeit und NP-Härte.
    - NP: Alle Probleme, in denen relativ kurze Lösungen zu raten sind und die Lösungseigenschaft schnell verifiziert werden kann.
    - NP-vollständig: die schwierigsten Probleme in NP
  - Nicht-entscheidbare Entscheidungsprobleme und nicht-berechenbare Funktionen: Da geht absolut nichts!

Worauf wird aufgebaut?

(1) Datenstrukturen:
  - Analyse der Laufzeit: O, o, Ω, w and Rekursionsgleichungen
  - Listen, Schlangen,Stacks, Heaps, Bäume, Graphen.

(2) Analysis und Lineare Algebra:
  - Grenzwerte und
  - Logarithmus

(3) Diskrete Modellierung: Wie beweist man Aussagen?

Literatur

- J. Kleinberg und E. Tardos, Algorithm Design, Addison Wesley 2005.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest und C. Stein, Introduction to Algorithms, Third Edition, MIT Press, 2009.
- M. Nebel, Entwurf und Analyse von Algorithmen, Springer, 2012.
- Skript zur Vorlesung.

Übungen,

bitte unbedingt teilnehmen!!!!

Übungsbetrieb (1/2)

Die Teilnahme am Übungsbetrieb ist für einen erfolgreichen Besuch der Veranstaltung EXTREM WICHTIG!

- Übungsblätter werden Donnerstags ausgegeben und in der darauffolgenden Woche Donnerstags **vor Beginn der Vorlesung** –oder im Briefkasten vor Raum 313 **bis 8:15**– abgegeben.

  Das erste Blatt wird diesen Donnerstag, also am 18. Oktober ausgegeben.
- Übungsgruppen treffen sich zum ersten Mal nächste Woche, also in der Woche des 22. Oktobers.

Übungsbetrieb (2/2)

- Sechs Übungsgruppen werden angeboten.
  - Eine Anmeldung ist nicht notwendig, aber verteilen Sie sich möglichst gleichmässig auf die verschiedenen Gruppen!
  - Informationen zu den Übungsgruppen und ganz allgemein zur Veranstaltung finden Sie auf der *Webseite*.
- Auf drei Übungsblätter folgt eine einwöchige Pause: In der Woche der Pause wird der bisherige Stoff wiederholt.
- Übungsaufgaben können in Gruppen bearbeitet werden, Lösungen müssen **SELBSTSTÄNDIG** aufgeschrieben werden.
  - Plagiate werden beim ersten Mal nicht bepunktet,
  - beim zweiten Mal werden alle Bonuspunkte gestrichen.
- **Idee und** formale Vorgehensweise sind stets zu erläutern.

Klausur

- Die Erstklausur findet am Freitag, den **22. Februar** um 9:15, im Hörsaal V statt. Die Zweitklausur findet am **8. April** um 9:15 statt.
- Benotung: Übungspunkte werden mit bis zu 20% zum Klausurergebnis hinzugerechnet.
- Bestehensquoten, wenn mindestens 50% aller Übungspunkte erreicht wurden:
  - GL1 im WS 2010/11: 93,75%
  - GL2 im SS 2011: 96%.

Freiversuche

Wenn Sie sich im höchstens dritten Fachsemester befinden, dann

- steht Ihnen ein **Freiversuch** zu:
  Eine nicht bestandene Prüfung gilt als nicht stattgefunden.
  - Freiversuche werden automatisch geltend gemacht.
- bzw ein **Freiversuch mit Verbesserungsmöglichkeit**, wenn Sie bisher höchstens vier solcher Freiversuche in Anspruch genommen haben.
  - Freiversuche mit Verbesserungsmöglichkeit müssen im Prüfungsamt angemeldet werden
  - und müssen zum frühestmöglichen Prüfungstermin erfolgen.

Haben Sie im SS 11 begonnen, sind aber in die neue Ordnung gewechselt, dann gelten die obigen Regeln auch für Sie.

Freiversuche stehen Ihnen auch dann für die Zweitklausur im April zu, wenn Sie an der Erstklausur nicht teilgenommen haben.

Sortieren

Das Sortierproblem

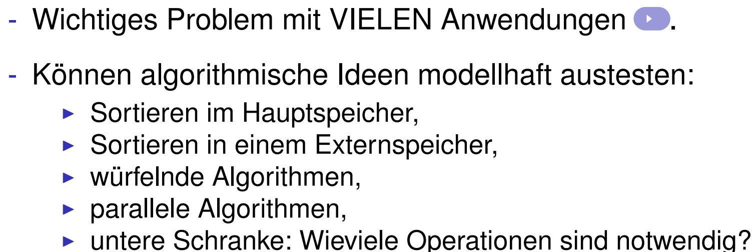
Eingabe: Schlüssel $x_1, \dots, x_n$ aus einer total geordneten Menge S.

Zum Beispiel: $S = \mathbb{R}$, $S = \mathbb{Q}$, $S = \Sigma^*$.

Aufgabe: Bestimme eine sortierte Reihenfolge, also eine Permutation π mit

$$x_{\pi(1)} \leqslant x_{\pi(2)} \leqslant \cdots \leqslant x_{\pi(n)}.$$

Warum betrachten wir das Sortierproblem?



- Wichtiges Problem mit VIELEN Anwendungen.
- Können algorithmische Ideen modellhaft austesten:
  - Sortieren im Hauptspeicher,
  - Sortieren in einem Externspeicher,
  - würfelnde Algorithmen,
  - parallele Algorithmen,
  - untere Schranke: Wieviele Operationen sind notwendig?

Bubble Sort

Bubble Sort und Kollegen

Wir beginnen mit den elementaren Sortierverfahren Bubble Sort, Selection Sort und Insertion Sort. Wie arbeitet Bubble Sort?

```
active = 1;
for (i=1; active; i++)
{ //Phase i beginnt:
    active = 0;
    for (j=1; j <= n-i; j++)
      if (A[j] > A[j+1]) { active = 1; swap(A, j, j+1); } }
```

- Phase 1:
  - vergleicht Nachbarn sukzessive, von links nach rechts, und vertauscht, wenn nötig.
  - Die größte Zahl erreicht Position n, wo sie auch hingehört.
- Phase i: $\max\{A[1], \ldots, A[n-i+1]\}$ endet in Position $n-i+1$.

Bubblesort: Die worst-case Laufzeit

Wir zählen die Anzahl der Vergleiche.

- $n-1$ Phasen genügen.
- In Phase i gibt es $n-i$ Vergleiche. Insgesamt:

$$\sum_{i=1}^{n-1}(n-i) = \sum_{j=1}^{n-1} j = \frac{n\cdot(n-1)}{2}$$

  Vergleiche.
- Die Eingabe $(2, 3, \ldots, n-1, n, 1)$ erfordert $n-1$ Phasen.

- Die worst-case Laufzeit für n Schlüssel ist $\Theta(n^2)$.
- Beschreibt das worst-case Verhalten Bubblesort akurat oder ist die erwartete Laufzeit besser?

Bubblesort: Die erwartete Laufzeit

Die erwartete Laufzeit, wenn alle Permutionen $\pi = (\pi(1), \ldots, \pi(n))$ der Eingabe $(1, \ldots, n)$ gleichwahrscheinlich sind, ist

$$\text{Erwartete Laufzeit} = \sum_{\pi} \frac{1}{n!} \cdot \text{Vergleiche}(\pi).$$

- Für die Hälfte aller Permutationen liegt die 1 in der rechten Hälfte.
  - Die 1 muss aber nach links zur Position 1 bewegt werden.
  - Eine Iteration bewegt die 1 aber nur um eine Position nach links.
- Mindestens die Hälfte aller Permutationen benötigen $\frac{n}{2}$ Iterationen.
- In $n/2$ Iterationen werden aber mindestens

$$\sum_{i=1}^{n/2} (n/2 - i) = \sum_{j=1}^{n/2-1} j = \frac{n}{4}\left(\frac{n}{2} - 1\right)$$

Vergleiche ausgeführt.

Die Laufzeit von Bubble Sort

- Erwartete Laufzeit $= \sum_{\pi} \frac{1}{n!} \cdot$ Vergleiche(π).
- Für mindestens die Hälfte aller Permutationen werden $\frac{n}{4}(\frac{n}{2} - 1)$ Operationen benötigt.
- Die erwartete Laufzeit ist mindestens $\frac{n}{8}(\frac{n}{2} - 1)$.

$\Theta(n^2)$ ist die worst-case wie auch die erwartete Laufzeit von Bubble Sort.

Bubble Sort ist sehr langsam, aber relativ schnell, wenn sich **alle** Schlüssel bereits *in der Nähe ihrer endgültigen Position* befinden.

Selection Sort

Selection Sort

```
for (i=1; i < n ; i++)
    finde die kleinste der Zahlen A[i],...,A[n] durch lineare Suche
    und vertausche sie mit A[i].
```

- Selection Sort besteht aus genau $n-1$ Iterationen.
- In der i-ten Phase benötigt Selection Sort $n-i$ Vergleiche, aber nur eine Vertauschung.
- Insgesamt:

$$\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{(n-1)\cdot n}{2} \text{ Operationen.}$$

- $\Theta(n^2)$ Vergleiche für **JEDE** Eingabe von n Schlüsseln.
- Positiv: Nur $n-1$ Vertauschungen. (Gut bei langen Datensätzen.)

Insertion Sort

Insertion Sort

for ($i = 1$; $i < n$; i++)
    //Die Zahlen $A[1], ..., A[i]$ sind bereits sortiert.
    Füge $A[i+1]$ in die Folge $(A[1], ..., A[i])$ ein. (Web)

- Insertion Sort arbeitet wie das von Hand Sortieren von Spielkarten.
- In Phase i wird $A[i+1]$ mit bis zu i Zahlen verglichen.
- Für die Eingabe $(n, n-1, n-2, \ldots, 3, 2, 1)$ werden

$$\sum_{i=1}^{n-1} i = \frac{(n-1) \cdot n}{2} \quad \text{Vergleiche durchgeführt.}$$

Auch Insertion Sort führt $\Theta(n^2)$ Operationen im worst-case aus.

Heapsort

Heapsort

Wir wissen schon wie man schneller sortiert: Heapsort!

- Füge alle n Schlüssel in einen anfänglich leeren Heap ein. Dies gelingt in Zeit $O(n \cdot \log_2 n)$ –sogar Linearzeit $O(n)$ ist möglich.
- Dann produziere die sortierte Reihenfolge durch n-malige Anwendung der Deletemin-Operation in Zeit $O(n \cdot \log_2 n)$.

Heapsort sortiert n Schlüssel in Zeit $O(n \cdot \log_2 n)$.

Aber Heapsort ist nur ein „asymptotischer Weltmeister“. Es gibt (um konstante Faktoren) schnellere Sortierverfahren.

Quicksort

void quicksort (int links, int rechts)

```
// Das Array A = (A[links], ..., A[rechts]) ist zu sortieren.
    { int p, i;
    if (links < rechts)
        { p = pivot (links, rechts);
        // A[p] dient als Pivot.
        i = partition (p, links, rechts);
        // Partition zerlegt A in ein Teilarray mit Schlüsseln ⩽ A[p],
        // gefolgt von A[p] in Position i und den Schlüsseln ⩾ A[p].
        quicksort (links, i-1);
        quicksort (i+1, rechts);
        }
    }
```

Eigenschaften von Quicksort

- Mit induktivem Beweis: Wenn partition (p, links, rechts) korrekt ist, dann ist auch quicksort (links, rechts) korrekt.
- Implementierung der Pivotfunktion. Zum Beispiel
  - pivot (links, rechts) = links
  - pivot (links, rechts) = die Position des mittleren Schlüssels aus $\{A[\text{links}], A[\frac{\text{links}+\text{rechts}}{2}], A[\text{rechts}]\}$
  - pivot (links, rechts) = eine zufällige Zahl aus $\{\text{links}, \ldots, \text{rechts}\}$.
- Um Quicksort als Systemfunktion zu benutzen, müssen wir das letzte Quentchen Geschwindigkeit herausquetschen.
  - Wo müssen wir aufpassen?
  - Die Pivot-Funktion ist unproblematisch.
  - Die Partition-Funktion sollte sehr schnell sein und keinen Speicherplatz neben dem Eingabearray verbrauchen.
  - Der für die rekursiven Aufrufe verwandte Stack sollte minimalen Speicherplatz einnehmen.

partition (int p, int links, int rechts)

(0) Wir arbeiten mit zwei Zeigern l und r. Weiterhin $\alpha = A[p]$.

(1) Setze l = links $-$ 1, r = rechts und vertausche A [rechts] und $A[p]$.
// Das Pivot-Element ist jetzt rechtsaußen.

(2) Wiederhole bis $l \geqslant r$:

  (a) Berechne l++ und verschiebe den l-Zeiger so lange nach rechts bis $A[l] \geqslant \alpha$. // Wir haben links einen Schlüssel $\geqslant \alpha$ gefunden.

  (b) Berechne $r--$ und verschiebe den r-Zeiger so lange nach links bis $A[r] \leqslant \alpha$. // Wir haben rechts einen Schlüssel $\leqslant \alpha$ gefunden.

  (c) Wenn ($l < r$), dann vertausche A [l] und $A[r]$.

(3) Vertausche A [rechts] und A [l].
// Das Pivot-Element kommt an die richtige Stelle?!

- Nach Initialisierung der Zeiger gilt die Invariante:
  Alle Zellen links von *l* (einschließlich *l*) besitzen nur Zahlen $\leqslant \alpha$,
  alle Zellen rechts von *r* (einschließlich *r*) besitzen nur Zahlen $\geqslant \alpha$.
- **Fall 1:** Wenn beide Zeiger zur Ruhe gekommen sind, ist $l < r$:
  Die Invariante gilt nach Vertauschen von $A[l]$ und $A[r]$!
- **Fall 2:** Wenn beide Zeiger zur Ruhe gekommen sind, ist $l \geqslant r$.
  - Alle Zahlen links von *l* sind kleiner oder gleich α. Alle Zahlen rechts von *l*, sind auch rechts von *r* und damit größer oder gleich α. Also ist *l* die endgültige Posititon von α.
  - Alles Paletti! Nix Paletti! Wir haben angenommen, dass beide Zeiger zur Ruhe kommen!

- Der *l*-Zeiger kommt zur Ruhe, er wird spätestens von Position rechts durch das Pivot-Element gestoppt.
- Wer hält den *r*-Zeiger auf? Möglicherweise niemand!

Die Partition-Funktion (3/3)

- Wir könnten den r-Zeiger durch eine Abfrage $r <$ links stoppen, aber diese Abfrage wird häufig gestellt!
- Wir versuchen, ohne Abfragen auszukommen. Setze $A[0] = -\infty$.
  - r wird beim ersten Aufruf durch $A[0]$ gestoppt.
  - Was passiert in weiteren rekursiven Aufrufen quicksort $(1, i-1)$ und quicksort $(i+1, n)$?
    - ★ $A[0]$ dient als Stopper für den ersten Aufruf und
    - ★ das Pivotelement in Position i als Stopper für den zweiten Aufruf!
- partition (p, links, rechts) benötigt $O(\text{rechts-links} + 1)$ Schritte.
- Partition benötigt neben dem Eingabearray keinen zusätzlichen Speicherplatz.

Quicksort ist ein „in place“ Sortierverfahren.

Weitere Verbesserungen von Quicksort

- Zufällige Wahl des Pivot-Elementes: Für **jede** Eingabe wird Quicksort hochwahrscheinlich schnell sein. Beweis später.
- Der administrative Aufwand für die Rekursion rechnet sich bei kleinen Teilproblemen nicht. Sortiere Teilprobleme der Größe höchstens 25(?!) mit Insertion Sort.
- Ein nicht-rekursiver Quicksort:
  - Führe pivot und dann partition aus.
  - Dann lege eines der beiden Teilprobleme auf den Stack.

Aber welches, das kleinere oder das größere?

Wenn der Stack immer möglichst klein sein soll,

dann lege das größere Teilproblem auf den Stack.

Die Stackhöhe ist durch $\lfloor \log_2 n \rfloor$ beschränkt. Warum?

- Zuerst wird ein Problem der Größe $m \geqslant \frac{n}{2}$ auf den Stack gelegt und Quicksort arbeitet auf einem Problem der Größe $n - m \leqslant \frac{n}{2}$ weiter.
- Nach Induktionsannahme wird ein Stack der Höhe höchstens $\lfloor \log_2 (n-m) \rfloor \leqslant \lfloor \log_2 n/2 \rfloor = \lfloor \log_2 n \rfloor - 1$ ausreichen, um das Problem der Größe $n - m$ erfolgreich abzuarbeiten.
- Die Stackhöhe wird in dieser Zeit also $(\lfloor \log_2 n \rfloor - 1) + 1 = \lfloor \log_2 n \rfloor$ nicht überschreiten.
- Das zuunterst liegende Problem hat eine Größe kleiner als n und wird nach Induktionsannahme mit Stackhöhe höchstens $\lfloor \log_2 n \rfloor$ abgearbeitet.

Laufzeit von Quicksort für n Schlüssel

- Die worst case Laufzeit (wenn partition(i,links,rechts) = links):
  Das sortierte Array $(1, 2, \ldots, n)$ produziert die rekursiven Aufrufe quicksort $(1, n)$, quicksort $(2, n)$, quicksort $(3, n), \ldots,$
  - quicksort (i, n) wird, über seine Partitionsfunktion, $n - i$ Vergleiche ausführen.
  - Also werden insgesamt mindestens

$$\sum_{i=1}^{n}(n-i) = 1 + 2 + 3 + \cdots + n - 1 = \; = \frac{n \cdot (n-1)}{2}$$

    Vergleiche benötigt.

- Die best case Laufzeit: Das jeweilige Sortierproblem wird stets in gleich große Teilprobleme aufgebrochen. Wir erhalten die Rekursionsgleichung

$$T(n) \leqslant 2 \cdot T(n/2) + c \cdot n$$

  für eine Konstante c. Und die Lösung ist

$$T(n) = O(n \cdot \log_2 n).$$

Laufzeit von Quicksort für n Schlüssel

Fazit

- Die worst-case Laufzeit von Quicksort ist $\Theta\left(n^2\right)$, die best-case Laufzeit ist $\Theta(n \log_2 n)$.
- Wem sollen wir trauen, der worst case oder der best case Laufzeit?
- In der Praxis wird ein sehr gutes Laufzeitverhalten beobachtet: Wir untersuchen die erwartete Laufzeit.

- Die Annahme: Alle Permutationen $\pi = (\pi(1), \ldots, \pi(n))$ von $(1, \ldots, n)$ sind gleichwahrscheinlich.
- Dann ist

$$E_n = \sum_{\pi} \frac{1}{n!} \cdot \text{Vergleiche}(\pi)$$

die erwartete Anzahl durchgeführter Vergleiche.

Die erwartete Laufzeit E_n

Wir definieren die Zufallsvariable

$$V_{i,j} = \begin{cases} 1 & \text{falls es zum Vergleich der Werte } i \text{ und } j \text{ kommt,} \\ 0 & \text{sonst.} \end{cases}$$

Dann ist

$$\begin{aligned} E_n &= \text{die erwartete Anzahl der Vergleiche} \\ &= \text{Erwartungswert}[\sum_{i=1}^{n} \sum_{j=i+1}^{n} V_{i,j}] \\ &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \text{Erwartungswert}[V_{i,j}]. \end{aligned}$$

Wir müssen den Erwartungswert von $V_{i,j}$ bestimmen!
Sei $p_{i,j}$ die Wahrscheinlichkeit für das Ereignis $V_{i,j} = 1$. Dann ist

$$\text{Erwartungswert}[V_{i,j}] = p_{i,j} \cdot 1 + (1 - p_{i,j}) \cdot 0 = p_{i,j}.$$

Mit welcher Wahrscheinlichkeit $p_{i,j}$ werden i und j verglichen?

Es sei $i < j$.

- i und j werden genau dann verglichen, wenn sich beide im selben rekursiven Aufruf befinden und einer das Pivotelement ist.
- i und j werden genau dann in denselben rekursiven Aufruf weitergeleitet, wenn das Pivot-Element kleiner als i oder größer als j ist.
- **Entscheidend**: Welcher Schlüssel k aus dem Intervall $[i, j]$ wird *als Erster* zum Pivot-Element?
  - $k = i$ oder $k = j$: i und j werden verglichen.
  - $k \neq i$ und $k \neq j$: i und j werden nicht miteinander verglichen, da sie durch das Pivot-Element k in verschiedene rekursive Aufrufe weitergeleitet werden.

Erwartungswert$[V_{i,j}] = p_{i,j} = \frac{2}{j-i+1}$.

Die erwartete Anzahl E_n der Vergleiche

Was ist die erwartete Anzahl E_n aller Vergleiche?.

$$\begin{aligned} E_n &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \text{Erwartungswert}[V_{i,j}] \\ &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} p_{i,j} \\ &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &= \sum_{i=1}^{n} (\frac{2}{2} + \frac{2}{3} + \frac{2}{4} + \cdots \frac{2}{n-i+1}) \\ &\leqslant \sum_{i=1}^{n} 2 \cdot \ln(n) \\ &= O(n \cdot \log_2 n). \end{aligned}$$

Zusammenfassung

Die Laufzeit von Quicksort

(a) Die worst-case Laufzeit ist $\Theta(n^2)$.

(b) Die best-case und die erwartete Laufzeit ist $\Theta(n \cdot \log_2 n)$.

Wir fixieren eine *beliebige* Eingabe und wählen das Pivot-Element zufällig. Was ist die erwartete Laufzeit?

Zufällige Pivotwahl

- Für **jede** Eingabe von n Schlüsseln ist $\Theta(n \log_2 n)$ die erwartete Laufzeit bei zufälliger Pivotwahl.
- Die Performance von Quicksort ist bei zufälliger Pivotwahl nicht mehr von der Eingabe abhängig!

Das Auswahlproblem

Das Auswahlproblem

Eingabe: Ein Array A, zwei Positionen links und rechts in A sowie die Zahl s

Aufgabe: Bestimme den s-kleinsten Schlüssel des Arrays $(A[\text{links}], \ldots, A[\text{rechts}])$.

Das Auswahlproblem ist leichter als das Sortierproblem:
Baue einen superschnellen Algorithmus.

Wende die Ideen von Quicksort an

- Wähle einen Pivot und zerlege das Array mit `partition`.
- Sei i die neue Position des Pivot-Elements.
  - Wenn $s \leqslant i - \text{links}$, dann ist der s-kleinste Schlüssel im Teilarray $A[\text{links}], \ldots, A[i-1]$ zu finden.
  - Wenn $s = i - \text{links} + 1$, dann stimmt der s-kleinste Schlüssel mit dem Pivotelement überein.
  - Ansonsten müssen wir im Teilarray $(A[i+1], \ldots, A\,[\text{rechts}])$ weitersuchen. (Mit $s = s - (i - \text{links} + 1)$.)

Und die Laufzeit?

Die erwartete Laufzeit

- Im Unterschied zu Quicksort erhalten wir stets nur *ein* Teilproblem.
- Wenn wir das Pivot-Element zufällig auswürfeln, dann sollte die erwartete Größe des EINEN Teilproblems ungefähr $n/2$ betragen.
- Die Laufzeit „sollte ungefähr“ die Rekursion

$$T(n) = T(n/2) + c \cdot n$$

für eine Konstante c besitzen.

Wahl des s-kleinsten Schlüssels

Wenn das Pivot-Element zufällig ausgewürfelt wird, dann kann der s-kleinste aus n Schlüsseln in erwarteter Zeit $O(n)$ bestimmt werden.

Beweis analog zur Analyse von Quicksort.

Mergesort

Mergesort

Wir sortieren, indem wir

- zuerst die linke und rechte Hälfte rekursiv sortieren und
- dann die beiden sortierten Hälften „mischen“. (Web)

```
void mergesort (int links, int rechts)
        { int mitte;
        if (rechts > links)
            { mitte = (links + rechts)/2;
            mergesort (links, mitte);
            mergesort (mitte + 1, rechts);
            merge (links, mitte, rechts); } }
```

Mischen

Mergesort funktioniert, wenn Merge die beiden sortierten Teilfolgen korrekt sortiert.

```
void merge (int links, int mitte, int rechts)
        { int i, j, k;
        for (i = links; i <= mitte; i++)
            B[i] = A [i];
        // B ist ein global definiertes integer-Array.
        // Das erste Teilarray wird nach B kopiert.
        for (i = mitte + 1; i <= rechts; i++)
            B [i] = A [rechts - i + mitte + 1];
        // Das zweite Teilarray wird in umgekehrter Reihenfolge nach
        // B kopiert. Jetzt kann gemischt werden.
        i = links, j = rechts;
        for (k = links; k <= rechts; k++)
            A [k] = (B[i] < B[j])? B[i++] : B[j--]; }
```

Laufzeit von Mergesort

- Merge mischt die beiden sortierten Teilarrays, indem es sukzessive nach der kleinsten Zahl in beiden Teilarrays fragt.
- $n = \text{rechts} - \text{links} + 1$ ist die Anzahl der beteiligten Schlüssel. Merge rechnet in Zeit $O(n)$.
- Für Mergesort erhalten wir die Rekursionsgleichung

$$T(n) = 2T(\frac{n}{2}) + O(n).$$

Mergesort sortiert n Schlüssel in Zeit $O(n \cdot \log_2 n)$.

\+ Positiv: Mergesort ist sehr schnell.

\- Negativ: Mergesort benötigt das zusätzliche Array B und ist damit kein in place Sortierverfahren.

Ein nicht-rekursiver Mergesort

n ist eine *Zweierpotenz*. Sortiere das Array $(A[1], \ldots, A[n])$.

- Die rekursive Sichtweise: Für mergesort(i, j) führe zuerst mergesort$(i, (i+j)/2)$ und mergesort$((i+j)/2+1, j)$ aus. Dann mische.
- Eine nicht-rekursive Sichtweise:
  - Zuerst mische alle *Paare*

    $$(A[2i+1], A[2 \cdot (i+1)])$$

    für $i = 0, \ldots, \frac{n}{2} - 1$.
  - Dann mische *Vierergruppen*

    $$(A[4i+1], A[4i+2], A[4i+3], A[4 \cdot (i+1)])$$

    für $i = 0, \ldots, \frac{n}{4} - 1$.
  - Allgemein, in der $k$ten Phase, mische

    $$(A[2^k i+1], A[2^k i+2], \ldots, A[2^k i+2^k-1], A[2^k \cdot (i+1)])$$

    für $i = 0, \ldots, \frac{n}{2^k} - 1$.

Externspeicher (1/2)

Modell eines Externspeichers

- Ein Zugriff ist um den Faktor 100.000 oder mehr langsamer als der Prozessortakt.
- Aber in einem Zugriff können viele **aufeinanderfolgende** Daten eingelesen werden.

Schnelle Sortierverfahren für Externspeicher arbeiten auf großen Blöcken aufeinanderfolgender Daten.

Wähle die Zweierpotenz B maximal so dass $2 \cdot B$ Schlüssel in den Hauptspeicher passen.

- Sortiere die Blöcke $(A[B \cdot i + 1], \ldots, A[B \cdot (i + 1)])$ nacheinander mit Quicksort.
- Zeit $= O(\frac{n}{B} \cdot B \cdot \log_2 B) = O(n \cdot \log_2 B)$.
- Jetzt wende den nicht-rekursiven Mergesort auf die $\frac{n}{B}$ sortierten Blöcke an.
  - Wieviele Iterationen? Nur $\log_2 \frac{n}{B}$ Iterationen.
  - Laufzeit des nicht rekursiven Mergesort = $O(n \cdot \log_2 \frac{n}{B})$.
- Wieviele Speicherzugriffe?
  - Für das anfängliche Sortieren mit Quicksort $\frac{n}{B}$ Zugriffe.
  - In $\log_2 \frac{n}{B}$ Iterationen jeweils $\frac{n}{B}$ weitere Zugriffe.

- Laufzeit insgesamt = $O(n \cdot \log_2 B) + O(n \cdot \log_2 \frac{n}{B}) = O(n \cdot \log_2 n)$.
- Anzahl der Speicherzugriffe = $O(\frac{n}{B} \log_2 \frac{n}{B})$.

Zusammenfassung: Quicksort und Mergsort

Quicksort mit zufälliger Pivot-Wahl

(a) Quicksort sortiert jede Folge von n Schlüsseln in place in erwarteter Zeit $\Theta(n \cdot \log_2 n)$.

(b) Mit einem Quicksort-Ansatz wird die s-kleinste Zahl in erwarteter Zeit $\Theta(n)$ bestimmt.

Der nicht-rekursive Mergesort

(a) n Schlüssel werden durch Mergesort in worst-case Zeit $\Theta(n \cdot \log_2 n)$ sortiert.

(b) Höchstens $O(\frac{n}{B} \log_2 \frac{n}{B})$ Zugriffe auf den Externspeicher.

$n = 1$ Billion, $B = 1$ Milliarde und 10.000 Speicherzugriffe genügen.

Kann man in linearer Zeit sortieren?

Ist $\Omega(n \cdot \log_2 n)$ eine Schallmauer?

- Alle bisher betrachteten Sortierverfahren
  (Heap-, Bubble-, Selection-, Insertion-, Quick- und Mergesort)
  sortieren, indem sie Schlüssel miteinander **vergleichen.**
- Wir haben nie die Laufzeit $O(n \cdot \log_2 n)$ schlagen können.

Wir zeigen, dass $\Omega(n \cdot \log_2 n)$ eine Schallmauer für

vergleichsorientierte

Sortierverfahren ist.

Vergleichsorientierte Sortierverfahren

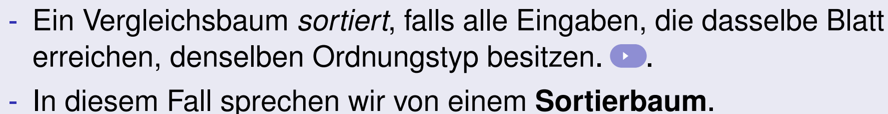
Vergleichsorientierte Algorithmen können durch Vergleichsbäume modelliert werden:

- Der **Ordnungstyp** einer Eingabe $a = (a_1, \ldots, a_n)$ ist die Permutation π, die a sortiert; also

$$a_{\pi(1)} < a_{\pi(2)} < \cdots < a_{\pi(n)}.$$

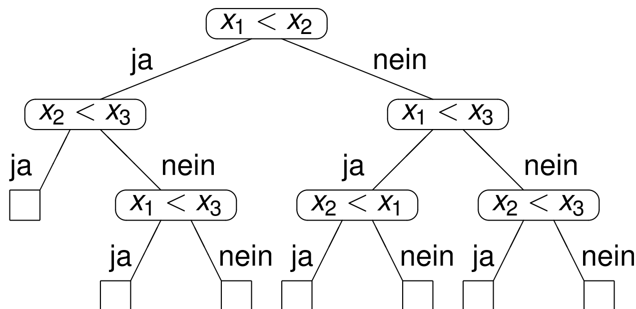
- Ein **Vergleichsbaum** für n Eingaben $x_1, \ldots, x_n$ ist ein binärer Baum, so dass jedem inneren Knoten genau ein Vergleich der Form $x_i < x_j$ zugeordnet ist.
  - Eine Eingabe $(a_1, \ldots, a_n)$ durchläuft den Baum, beginnend mit der Wurzel.
  - Falls der Knoten v erreicht wird und falls $x_i < x_j$ auszuführen ist, dann weiter mit dem linken (rechten) Kind, falls $a_i < a_j$ $(a_i \geqslant a_j)$.

Sortierbäume



- Ein Vergleichsbaum *sortiert*, falls alle Eingaben, die dasselbe Blatt erreichen, denselben Ordnungstyp besitzen. .
- In diesem Fall sprechen wir von einem **Sortierbaum.**

Der Sortierbaum von Bubblesort für $n = 3$ Schlüssel:



Sortierverfahren und ihre Sortierbäume

Wenn ein vergleichsorientiertes Sortierverfahren im worst case mit $T(n)$ Vergleichen sortiert, dann hat sein Sortierbaum die Tiefe $T(n)$.

- Mindestens ein Blatt pro Ordnungstyp.
  - Wieviele Blätter hat ein Sortierbaum mindestens?
  - Mindestens $n!$ Blätter.
- Ein Sortierbaum ist ein Binärbaum.
  - Wieviele Blätter hat ein Sortierbaum der Tiefe t höchstens?
  - Höchstens 2^t Blätter.
- Wenn Tiefe t ausreicht, dann muss gelten: $2^t \geqslant n!$. $\Rightarrow$

$$\begin{aligned} t &\geqslant \log_2(n!) \geqslant \log_2(n) + \log_2(n-1) + \cdots + \log_2(\frac{n}{2}) \\ &\geqslant \frac{n}{2} \cdot \log_2 \frac{n}{2}. \end{aligned}$$

Eine untere Schranke

Satz

- Jeder Sortierbaum für n Schlüssel hat Tiefe $\Omega(n \cdot \log_2 n)$.
- Jedes vergleichsorientierte Sortierverfahren führt mindestens $\Omega(n \cdot \log_2 n)$ Vergleiche durch.

Distribution Counting

Sortierverfahren müssen Vergleiche benutzen und deshalb sind mindestens $\Omega(n \cdot \log_2 n)$ Vergleiche notwendig!

Schwachsinn!

Distribution Counting

Sortiere Array $(A[1], \ldots, A[n])$ mit $A[i] \in \{0, \ldots m-1\}$:

```
void counting ( )
    { int Zaehle[m]; int wo = 0;
    for (int i = 0; i < m; i++) // m Iterationen
        Zaehle[i] = 0;
    for (int j = 1; j <= n; j++) // n Iterationen
        Zaehle[A[j]]++;
        // Füge A[j] in die Zelle mit Position A[j] ein.
    for (i = 0; i < m; i++) // n + m Iterationen
        { for (j = 1; j <= Zaehle[i]; j++)
              A[wo + j] = i;
          wo += Zaehle[i]; } }
```

Eine Verallgemeinerung von Distribution Counting

Distribution Counting

Distribution Counting sortiert n Zahlen aus der Menge $\{0, \ldots, m-1\}$ in Zeit $O(n+m)$.

+ Extrem schnell, solange nur kleine Zahlen zu sortieren sind.
- Völlig unpraktikabel, wenn die Zahlen etwa so groß wie n^2 sein können.

Kann Distribution Counting auf größere Zahlen effizient verallgemeinert werden?

Radixsort

Radixsort: Die Idee

- Arbeite mit einer Basis $b \leqslant n$.
- Berechne die b-äre Darstellung für jede Zahl.
  - Wende Distribution Counting an, um die Zahlen nach ihrer niedrigstwertigen Ziffer zu sortieren.
  - Dann wende wieder Distribution Counting an, um die Zahlen nach ihrer zweitniedrigstwertigen Ziffer zu sortieren.
    - Aber wie behalten wir die bisherigen Vergleiche (bei gleicher zweiter Ziffer) in Erinnerung?

Radixsort

1. Bestimme die b-äre Darstellung für alle Zahlen:

$$A[i] = \sum_{k=0}^{L} z_{i,k} \cdot b^k : \qquad z_{i,k} \text{ ist die } k\text{te Ziffer von } A[i].$$

2. for (k=0; k $\leqslant$ L; k++)

   // Die **Verteilphase** beginnt.

   { for (i=1; i $\leqslant$ n, i++)

   Füge $A[i]$ in die **Schlange** mit Nummer $z_{i,k}$ ein.

   // Die **Sammelphase** beginnt. (Ist die Phase notwendig?)

   for (s=0;$s \leqslant b$, s++)

   Entleere die Schlange mit Nummer s in das Array A. }

- Bevor die $k$te Ziffer untersucht wird, möge $A[i]$ vor $A[j]$ stehen.
- Dann steht $A[i]$ vor $A[j]$ nach Untersuchung der $k$ten Ziffer, solange $z_{i,k} \leqslant z_{j,k}$. Warum? (Animation)

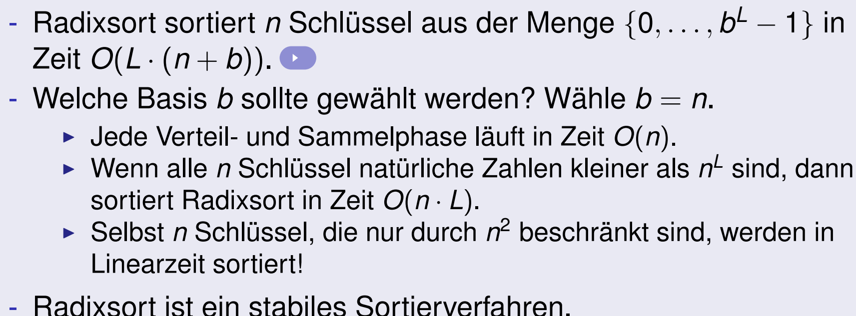
Stabilität

Ein Sortierverfahren heißt **stabil**, wenn es die Reihenfolge von Elementen mit gleichem Schlüssel bewahrt.

Wird z.B. eine nach Vornamen geordnete Namensliste stabil nach den Nachnamen sortiert, so erhält man eine Liste, in der Personen mit gleichem Nachnamen nach Vornamen sortiert sind.

- Ist Radixsort ein stabiles Sortierverfahren?
- Welche Basis b sollte man wählen?
- Warum führt $b < n$ auf keine wesentlich bessere Laufzeit als $b = n$?

Eigenschaften von Radixssort & Wahl der Basis



- Radixsort sortiert n Schlüssel aus der Menge $\{0, \ldots, b^L - 1\}$ in Zeit $O(L \cdot (n + b))$.
- Welche Basis b sollte gewählt werden? Wähle $b = n$.
  - Jede Verteil- und Sammelphase läuft in Zeit $O(n)$.
  - Wenn alle n Schlüssel natürliche Zahlen kleiner als n^L sind, dann sortiert Radixsort in Zeit $O(n \cdot L)$.
  - Selbst n Schlüssel, die nur durch n^2 beschränkt sind, werden in Linearzeit sortiert!
- Radixsort ist ein stabiles Sortierverfahren.

Sample Sort

Message Passing Interface (MPI)

p Rechner sind durch ein Kommunikationsnetzwerk miteinander verbunden. Zu den (MPI-)Kommunikationsfunktionen gehören:

- Send und Receive: Point-to-Point Kommunikation zwischen einem Sender und einem Empfänger.
- Broadcast: Ein Sender schickt eine Nachricht an alle Rechner.
- und personalisierter Broadcast: Ein Sender schickt individualisierte Nachrichten an alle Rechner.

Was ist zu tun?

Eingabe: Der $i$te Rechner erhält das $i$te Intervall der n Schlüssel.

Aufgabe: Die Rechner arbeiten parallel, um alle Schlüssel zu sortieren. Der $i$te Rechner soll das $i$te Intervall der sortierten Folge ausgeben.

Eine parallele Variante von Quicksort

- Quicksort ist schon fast ein paralleler Algorithmus:
  - die Folge der Schlüssel, die kleiner als das Pivotelement sind, und die Folge der Schlüssel, die größer als das Pivotelement sind, können unabhängig voneinander sortiert werden.
  - Allerdings können wir nur zwei Rechner beschäftigen.
- Stattdessen, wähle ein „Sample“ S von $p-1$ Pivotelementen.
  - Jeder Rechner partitioniert seine Schlüssel gemäß S in p Intervalle,
  - verschickt $p-1$ Intervalle mit personalisiertem Broadcast
  - und sortiert die erhaltenen und das verbliebene Intervall.
- Berechnung der Stichprobe S:
  - Jeder Rechner sortiert seine $\frac{n}{p}$ Schlüssel und sendet die p Schlüssel in den Positionen $i \cdot \frac{n}{p^2}$ (für $i = 0, \ldots p-1$) an Rechner 1.
  - Rechner 1 sortiert die p^2 erhaltenen Schlüssel und „broadcastet“ die Schlüssel in den Positionen $i \cdot p$ (für $i = 1, \ldots, p-1$) als Stichprobe S.
  - $\Theta(\frac{n}{p})$ Schlüssel befinden sich zwischen zwei aufeinanderfolgenden Schlüsseln der Stichprobe. (Übungsaufgabe)

Sample Sort

(1) Jeder Rechner sortiert seine $\frac{n}{p}$ Schlüsssel und sendet eine Stichprobe der Größe p an Rechner 1. // Zeit $O(\frac{n}{p} \log_2 \frac{n}{p})$.

(2) Rechner 1 sortiert die erhaltenen $(p-1) \cdot p$ Schlüssel zusammen mit den eigenen p Schlüsseln und versendet eine endgültige Stichprobe S der Größe $p-1$ per Broadcast. // Zeit $O(p^2 \log_2 p^2)$.

(3) Jeder Rechner partitioniert seine Schlüssel in die p durch S definierten Intervalle $I_1 =]-\infty, s_1[, I_2 = [s_1, s_2[, \ldots, I_p = [s_{p-1}, \infty[$. Dann werden alle Schlüssel, die im Intervall I_j liegen, an Rechner j mit personalisiertem Broadcast weitergeleitet. // Zeit $O(\frac{n}{p} \cdot \log_2 p)$.

(4) Jeder Rechner mischt die p sortierten Folgen. // Zeit ?

Sample Sort: Die Laufzeit

Wenn man die Kommunikationskosten NICHT zählt, ist die Zeit durch

$$\begin{aligned} & \frac{n}{p}\log_2\frac{n}{p} + p^2\log_2 p^2 + \frac{n}{p}\cdot\log_2 p + ? \\ \leqslant \quad & 2\frac{n}{p}\log_2 n + p^2\log_2 p^2 + ? \end{aligned}$$

asymptotisch beschränkt. Wenn $p^2 \leqslant \frac{n}{p}$, also wenn $p \leqslant n^{1/3}$, dann ist die Laufzeit höchstens

$$O(\frac{n}{p}\log_2 n + ?).$$

Eine perfekte Parallelisierung:
Die sequentielle Laufzeit wird durch den Einsatz von p Prozessoren um den Faktor $O(p)$ gesenkt –falls $? = O(\frac{n}{p}\log_2 n)$.

Sortieren: Zusammenfassung

- Quicksort:
  - Ein schnelles in place Sortierverfahren. Quicksort ist ideal, wenn alle Daten in den Hauptspeicher passen.
  - Die zufällige Pivotwahl garantiert, dass **jede** Folge von n Zahlen in **erwarteter Zeit** $O(n \cdot \log_2 n)$ sortiert wird.
  - Eine Quicksort-Variante löst das Auswahlproblem in Zeit $O(n)$.
- Mergesort:
  - Sortiert jede Folge von n Zahlen in **worst case Zeit** $O(n \cdot \log_2 n)$.
  - Eine nicht-rekursive Variante sortiert n auf einem Externspeicher gespeicherte Zahlen mit höchstens $O(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{M})$ Speicherzugriffen, solange B Zahlen in einem „Schwung" vom Externspeicher in den Hauptspeicher der Größe M gebracht werden können.
- Jedes vergleichsorientierte Sortierverfahren benötigt mindestens $\Omega(n \cdot \log_2 n)$ Vergleiche, aber
- Radixsort sortiert n Zahlen aus der Menge $\{0, \ldots, n^L - 1\}$ in Zeit $O(n \cdot L)$ ohne irgendeinen Vergleich auszuführen.

Suche in einem Array

Eingabe: Ein Array A mit n Schlüsseln und ein Schlüssel x.

Frage: Kommt x in A vor?

- Wenn A nicht sortiert ist:
  - Wir müssen im schlimmsten Fall alle n Schlüssel von A inspizieren.
  - $\Rightarrow$: Laufzeit $O(n)$.
- Wenn A sortiert ist:
  - Suche x mit Binärsuche!
  - $\Rightarrow$: Laufzeit $O(\log_2 n)$.

Wer hat bezahlt?

Eingabe: Eine Liste A von n ausgestellten Rechnungen sowie eine Liste B von $m < n$ bezahlten Rechnungen.

Aufgabe: Bestimme die Liste C der nicht bezahlten Rechnungen.

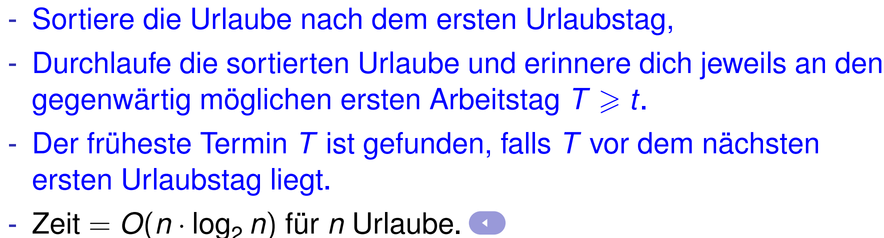
- Naiver Ansatz: Durchlaufe B und vermerke für jede bezahlte Rechnung in A, dass bezahlt wurde.
  - Für jede der m bezahlten Rechnungen sind bis zu n Operationen notwendig. $\Rightarrow$ Zeit $= O(m \cdot n)$.
- Schneller: Sortiere die Liste A. Dann durchlaufe B und vermerke für jede bezahlte Rechnung in A, dass bezahlt wurde.
  - $\Rightarrow$ Zeit $= O(n \cdot \log_2 n + m \cdot \log_2 n)$.
- Noch schneller: Sortierte die Liste B. Dann durchlaufe A und vermerke, ob Rechnung bezahlt.
  - $\Rightarrow$ Zeit $= O(m \cdot \log_2 m + n \cdot \log_2 m)$.

Wann sind alle da?

Eingabe: Eine Liste von genehmigten Urlauben der Form

(1. Urlaubstag, 1. Arbeitstag nach dem Urlaub).

Aufgabe: Bestimme zu einem Stichtag t den frühstmöglichen Termin *nach* t, zu dem kein Mitarbeiter im Urlaub ist.



- Sortiere die Urlaube nach dem ersten Urlaubstag,
- Durchlaufe die sortierten Urlaube und erinnere dich jeweils an den gegenwärtig möglichen ersten Arbeitstag $T \geqslant t$.
- Der früheste Termin T ist gefunden, falls T vor dem nächsten ersten Urlaubstag liegt.
- Zeit $= O(n \cdot \log_2 n)$ für n Urlaube.