Datenstrukturen & Algorithmen 11. März, 2015

Handout 3

Sebastian Millius, Sandro Feuz, Daniel Graf

Thema: Sortieren

Referenz: Widmayer, Ottman Kapitel 2, Cormen et. al. Kapitel 6-8

Links

- Visualierung diverser Sortierverfahren: http://www.cs.usfca.edu/~galles/visualization/download.html (http://goo.gl/gozps)
- Visualization Quick Sort: http://goo.gl/jqnjg
- Visualization Heap: http://people.ksp.sk/~kuko/bak/ (http://goo.gl/HfFr1)
- Auf Wikipedia sind Erklärungen, Visualisierungen und Verweise zu Implementationen für die meisten Sortierverfahren zu finden
- Obama - Computer Science Question: http://goo.gl/66qU6
- Sortierverfahren vorgetanzt: http://goo.gl/vsDLQ
- Sortierverfahren als Melodie: https://www.youtube.com/watch?v=kPRA0W1kECg

Gegeben ein Array A von n Elementen. Gebe eine Permutation A_s von A an, so dass

$$\forall k \in \{1, \dots, n-1\} : \text{KEY}(A_s[k]) \leqslant \text{KEY}(A_s[k+1])$$

Sortieren gehört zu den ältesten, wichtigsten und am besten verstandenen Problemen in der Informatik. Viele Probleme und Algorithmen in Praxis und Theorie benötigen Sortieren.

Ein Sortieralgorithmus heisst *stable* (dt. stabil), wenn er die relative Ordnung von Elementen mit gleichem Schlüssel beibehält. Stabilität kann allgemein forciert werden, in dem z.B. der Index in der unsortierten Reihenfolge an den Schlüssel angehängt wird (Beispiel: Array $A[1 \dots n]$, wir sortieren nach Tuple $(A[i], i)$).

(Vergleichsbasierte) Sortierverfahren

"Suboptimale" Algorithmen

Insertion Sort

Sortieren von Karten mit Insertion Sort

Iterativ wird jedes Element in die sortierte Reihenfolge eingefügt. Jedes zusätzliche Element, welches beim Erweitern des bereits sortierten Teiles hinzukommt, wird an die korrekte Stelle geschoben. Dazu müssen bis zu $\mathcal{O}(n)$ Elemente verschoben werden, nämlich wenn das neue Element kleiner als alle bisherigen ist. Das Verfahren funktioniert auf die gleiche Weise, wie viele eine Hand Karten sortieren.

```
InsertionSort(A)
1  for j = 2 to length[A]
2      key = A[j]
3      // Füge A[j] in die sortierte Sequenz A[1 ... j − 1]
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

- *Vergleiche*: $\mathcal{O}(n^2)$ (Best Case: $\mathcal{O}(n)$)
- *Verschiebungen*: $\mathcal{O}(n^2)$ (Best Case: 0)
- *in place?*: Ja
- *stable?*: möglich

Bubble Sort

Wiederholt wird die unsortierte Sequenz durchlaufen und benachbarte Elemente, die in der falschen Reihenfolge liegen, vertauscht.

- *Vergleiche*: $\mathcal{O}(n^2)$ (Best Case: $\mathcal{O}(n)$)
- *Verschiebungen*: $\mathcal{O}(n^2)$ (Best Case: 0)
- *in place?*: Ja
- *stable?*: Ja

Heap Sort

Heaps sind eine einfache und elegante Datenstruktur die effizient die Wörterbuchoperationen *insert* und *extract-min* implementieren. Sie funktionieren durch den Erhalt einer Teil-/Partialordnung die schwächer ist als die totale Ordnung (so dass die Struktur effizient erhalten werden kann) aber stark genug um das minimale Element schnell zu finden.

Ein Binary Heap ist ein Binärbaum mit folgenden Eigenschaften

- *Heapbedingung*: Jeder Knoten ist das Minimum seines ganzen Subtrees (bzw. Maximum bei einem Max-Heap)
- *Struktur*: Alle Schichten bis auf die letzte sind ausgefüllt. Die letzte Schicht wird von "links" aufgefüllt (linksbündig)

Stop and Think: Who's where in the heap? (Heaps vs. Binary Search Trees)
Wie kann man effizient nach einem bestimmten Schlüssel in einem Heap suchen?

Lösung: Man kann nicht. Binäre Suche funktioniert nicht, da der Heap kein binärer Suchbaum ist. Man weiss praktisch nichts (ausser der Heap-Bedingung) über die relative Ordnung der Elemente in einem Heap - jedenfalls nicht genug um schneller zu sein als eine lineare Suche über die Elemente des Heaps zu machen.

Stop and Think: O Maximum, Where Art Thou?
Wo kann das Maximum in einem Min-Heap überall sein?

Repräsentation in einem Array

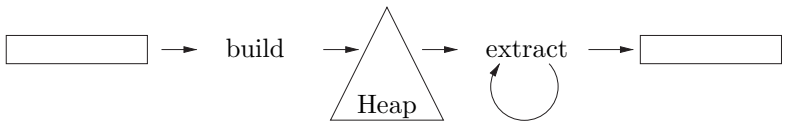
Durch die balancierte und ausgerichtete Struktur ist es sehr einfach möglich, den Heap in einem Array zu repräsentieren.

Für ein Array A mit den Elementen $A[0]...A[n-1]$ sieht dies folgendermassen aus:

- $A[0]$ ist die Wurzel des Heaps, also das Minimum (bzw. Maximum bei einem Max-Heap)
- Für jedes Element $A[i]$ ist $A[2i+1]$ resp. $A[2i+2]$ das linke resp. rechte Kind. Alternativ sind bei 1 beginnenden Arrays die Kinder an positionen $2i$ und $2i+1$.

Algorithmus (mit Max-Heap)

1. Heap erstellen (in $\mathcal{O}(n)$ möglich!)
2. Maximum (das erste Element) mit dem letzten Element vertauschen. Der Heap wird danach um ein Element kleiner
3. Neues erstes Element versickern, so dass die Heapbedingung wieder hergestellt ist ($\mathcal{O}(n \log n)$)
4. Wiederholen bis nur noch ein Element im Heap übrig bleibt



Die Gesamtlaufzeit für Heap Sort: $\mathcal{O}(n \log n)$

```
HeapSort(A)
1  Build-Heap(A) // Heap erstellen
2  for i = length[A] downto 2
3       exchange A[1] ↔ A[i]
4       heapsize[A] = heapsize[A] − 1
5       versickern(A, 1)
```

- *Vergleiche*: $\mathcal{O}(n \log n)$
- *in place?*: Ja
- *stable?*: Nein

Aufgabe Sortieren Sie das Array [H,E,A,P,S,O,R,T,M,E] aufsteigend mit Heapsort. Geben Sie dabei nach jedem Zwischenschritt, bei dem ein Schlüssel an die endgültige Position wandert, den Inhalt des Arrays an

Aufgabe Gegeben ist ein Array A von n Zahlen, bestimme die k grössten Zahlen (in $\mathcal{O}(k \log n + n)$)

Natürliches 2-Wege Merge Sort

Idee: Vorsortiertheit ausnutzen

Dieses Verfahren ist eine Optimierung des normalen Merge Sorts. Erstens wird dabei keine Rekursion benötigt und zweitens werden schon vorsortierte Teilfolgen (*runs*) ausgenutzt.

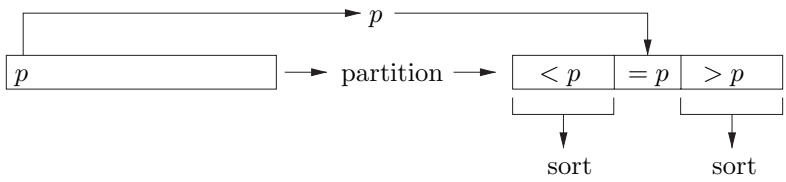
Eine ausführliche Darstellung findet sich im Buch *Algorithmen und Datenstrukturen* von Widmayer und Ottman.

Der Algorithmus geht durch die Liste und verschmilzt alle benachbarten Runs. Dabei wird die Anzahl der Runs in jedem Durchgang halbiert. Da zu Beginn zwischen minimal 0 und maximal n Runs vorliegen ist die Laufzeit (*think about it!*) $\mathcal{O}(n \log k)$, wobei n die Anzahl Elemente ist und k die Anzahl der anfänglich vorsortierten Folgen. Insbesondere erreichen wir so eine lineare Laufzeit für bereits sortierte Eingaben.

Quick Sort

Idee: Divide & Conquer

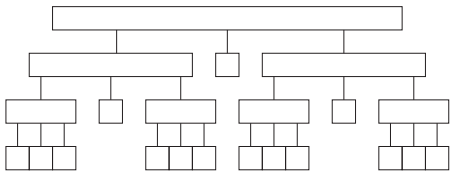
- *Divide*: Partitioniere mittels des Pivot in einen linken ($<$ Pivot) und einen rechten ($>$ Pivot) Teil
- *Conquer*: Führe den Algorithmus rekursiv auf beiden Teilen aus



Best Case für Quick Sort

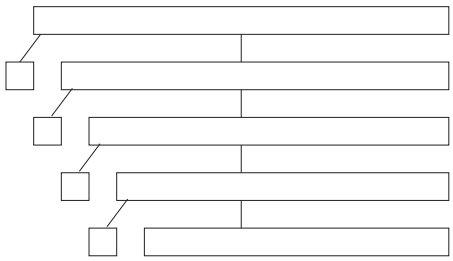
Der Best Case für *Divide & Conquer* kommt vor, wenn wir jedesmal den Input möglichst in der Mitte halbieren können, d.h. jedes Teilproblem ist von der Grösse $n/2$.

Das Partitionieren ist jeweils $\mathcal{O}(n)$ und wir Teilen $\log_2 n$ mal. $\rightsquigarrow$ $\mathcal{O}(n \log n)$ (sehr "sloppy" Begründung)



Worst Case für Quick Sort

Das Pivot Element teilt das Array ungleichmässig, so dass wir in einem Teil kein Element mehr haben, d.h. das Pivot ist jedesmal das grösste oder kleinste Element im Array. Jetzt teilen wir n mal, anstatt $\log_2 n$ mal, $\rightsquigarrow$ Worst Case Time von $\mathcal{O}(n^2)$, da die ersten $\frac{n}{2}$ "Ebenen" jeweils mehr als $\frac{n}{2}$ Elemente zu teilen haben.



Die Laufzeit hängt also stark von der konkreten Wahl des Pivotelements p ab und kann im schlechtesten Fall quadratisch werden.

- *Randomisiertes Quick Sort*: Das Pivotelement wird zufällig (uniform) ausgewählt. Randomisiertes Quick Sort läuft in $\mathcal{O}(n \log n)$ auf jedem Input mit hoher Wahrscheinlichkeit.
- *Median of Three*: Man betrachtet das erste, letzte und ungefähr das mittlere Element und wählt den Median dieser drei Elemente.

War Story: TimSort

Für Interessierte hier eine aktuelle Geschichte zum Sortierverfahren, das in den Standardbibliotheken von Java, Android und Python verwendet wird: *TimSort*. Als spezielle Variante des natürlichen Mergesorts hat es, korrekt implementiert, eine Laufzeit von $\mathcal{O}(n \log n)$. Kürzlich

haben Sicherheitsforscher bemerkt, dass die gängigen Implementierungen in den Standardbibliotheken nicht korrekt sind und haben gezeigt, dass sich eine Eingabe finden lässt auf denen die Implementationen in quadratischer Zeit laufen. Die genaue Beschreibung des Algorithmus, den Bug in der Implementierung, sowie die Herleitung der Gegenbeispiele sind in diesem Artikel zu finden: http://goo.gl/TgDrSk.

Skuril aber einfach: Gnome Sort (aka Stupid Sort)

Ein auf den ersten Blick verblüffendes Sortierverfahren ist Gnome Sort. Es beschreibt eine Anleitung für einen Gartenzwerg, der bei A_1 startet und Schritt für Schritt durch die Element wandert. Es sei i die aktuelle Position des Gartenzwerges. Der Zwerg befolgt nun diese vier Regeln:

- Ist $a_i > a_{i+1}$, so tausche die Elemente und mache einen Schritt nach links ($i := i - 1$)
- Ist $a_i \leqslant a_{i+1}$, gehe einen Schritt nach rechts ($i := i + 1$).
- Wenn wir bei $i = 0$ über den linken Rand hinausfahren, gehen wir zurück zu $i = 1$.
- Erreichen wir den rechten Rand ($i = n$), so sind wir fertig und der Array ist sortiert.

Schaut man etwas genauer hin, fällt auf, dass diese Beschreibung nur ein umformuliertes Sortieren durch Einfügen erklärt. Der Gartenzwerg bringt Schlüssel für Schlüssel an den korrekten Platz in einem bereits sortierten Präfix (think about it!). Anstatt die korrekte Position im sortierten Teilarray mittels binärer Suche zu finden, schiebt der Zwerg den Schlüssel solange nach links, bis er an der korrekten Stelle angelangt ist. Das Platzieren des neuen Schlüssels an der korrekten Stelle im Präfix dauert also linear viele Schritte, genau wie bei Insertion Sort. Somit läuft auch Gnome Sort in $\mathcal{O}(n^2)$ hat aber den Vorteil auf bereits sortierten Eingaben nach bereits $\mathcal{O}(n)$ vielen Schritten fertig zu sein.

Lower Bound

All dies ist ausführlich in *Algorithmen und Datenstrukturen*, T. Ottmann & P. Widmayer, Kapitel 2.8 S. 138ff dargestellt oder auch in *Introduction to Algorithms*, Cormen et al. Kapitel 8.1.

Vergleichsbasierte Sortierverfahren können abstrakt als *decision trees* (dt. Entscheidungsbäume) gesehen werden. Ein Entscheidungsbaum ist ein voller binärer Baum der Vergleiche zwischen Elementen repräsentieren die ausgeführt werden durch ein bestimmtes Sortierverfahren auf einer Eingabe einer gegebenen Länge.

Jedes Blatt ist mit einer Permutation π der Eingabesequenz gekennzeichnet. Die Ausführung eines Sortieralgorithmus entspricht dem Ablaufen eines Pfades von der Wurzel zu einer Permutation bei einem Blatt. Bei jedem inneren Knoten wird ein Vergleich $a[i] \leqslant a[j]$ gemacht. Angekommen bei einem Blatt hat der Sortieralgorithmus die Reihenfolge $a[\pi(1)] \leqslant \ldots \leqslant a[\pi(n)]$ erhalten.

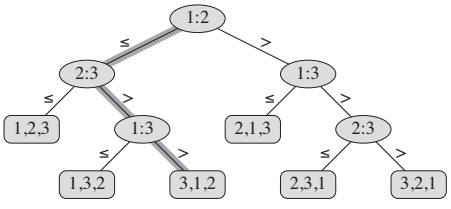


Abb.: Entscheidungsbaum für Insertion Sort für 3 Elemente

Da jeder korrekte Sortieralgorithmus fähig sein muss jede Permutation seines Inputs zu erzeugen (think about it!) ist eine notwendige Vorraussetzung für die Korrektheit, dass der Baum alle möglichen Permutationen als Blätter hat. Insbesondere hat er also mindestens $n!$ Blätter.

Die Länge des *längsten* Pfades von der Wurzel zu einem Blatt representiert die Worst Case Anzahl von notwendigen Vergleichen. Das heisst, die Worst Case Anzahl von Vergleichen für ein gegebenes Sortierverfahren ist gleich der Höhe seines Entscheidungsbaumes. Eine untere Schranke der Höhe für alle möglichen Entscheidungsbäume bei denen jede Permutation als Blatt erreichbar ist, ist deshalb eine untere Schranke für das vergleichsbasierte Sortieren.

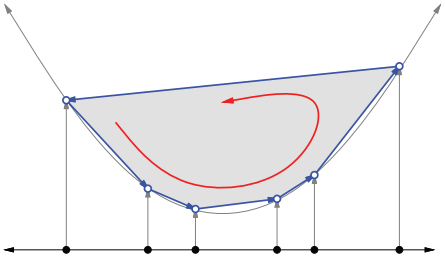
Da ein binärer Baum der Höhe h nicht mehr als 2^h Blätter hat, gilt: $n! \leqslant 2^h$ und deshalb $h \geqslant \log(n!) = \Omega(n \log n)$.

Daraus folgt, dass *jeder vergleichsbasierte Sortieralgorithmus* $\Omega(n \log n)$ *Vergleiche im Worst Case benötigt*. Heap Sort und Merge Sort sind deshalb asymptotisch optimale Sortierverfahren.

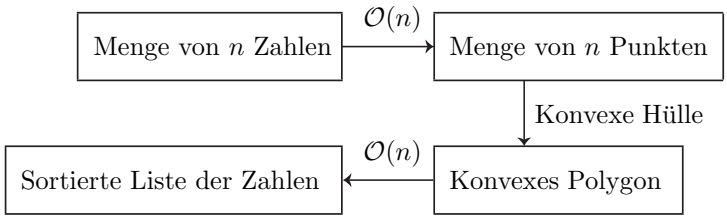
Reduktion

Betrachte das Problem der Bestimmung der *Convex Hull* (dt. Konvexe Hülle) einer Menge von Punkten in der Ebene. Wir beweisen eine untere Schranke für dieses Problem durch Reduktion des Sortierproblems auf dieses.

Um eine Liste von n Zahlen $\{a, b, c, \ldots\}$ zu sortieren, transformieren wir diese in eine Menge von Punkten $\{(a, a^2), (b, b^2), \ldots\}$ Man kann sich vorstellen das die ursprünglichen Zahlen auf einer reellen Zahlenline sind und die Transformierten dann auf der Parabel $y = x^2$ liegen.



Wir berechnen dann die konvexe Hülle dieser Punkte (alle werden auf der Hülle liegen!). Um die sortierte Reihenfolge der ursprünglichen Zahlen zu erhalten, gehen wir im Gegenurzeigersinn die Hülle durch.



Die beiden Transformationen können in $\mathcal{O}(n)$ berechnet werden. Da die konvexe Hülle als zirkuläre Liste der Punkte gegeben ist, ist das sortierte Ausgeben der Zahlen ebenfalls in $\mathcal{O}(n)$. D.h. mit dem Algorithmus zur Berechnung der konvexen Hülle kann (mit linearer Extrazeit) sortiert werden. Damit gilt für die Komplexitäten:

$$T_{\text{Sort}}(n) \leqslant T_{\text{Convex Hull}}(n) + \mathcal{O}(n)$$

und deshalb gilt die untere Schranke von $\Omega(n \log n)$ ebenfalls für die Berechnung der konvexen Hülle (again, think about it!).

Einige Algorithmen zur Berechnung der konvexen Hülle werden später in der Vorlesung behandelt.

Sortieren unter $\Omega(n \log n)$

Wie in der Vorlesung gesehen, ist $\Omega(n \log n)$ eine untere Schranke für das vergleichsbasierte Sortieren. Vergleichsbasiert heisst dabei, dass die einzige Voraussetzung an die zu sortierenden Elemente eine existierende $\leqslant$ Relation ist.

Falls jedoch noch andere Bedingungen erfüllt sind, ist es möglich schnellere Sortierverfahren anzugeben. Beispiele:

Radix Exchange Sort (Siehe Implementation)

- *Voraussetzung*: Objekte in Binärrepräsentation mit fester Länge k.
- *Verfahren*: Partitionierung wie Quicksort, jedoch bezüglich einzelner Bits.
- *Komplexität*: $\Theta(nk)$ Worstcase.

Counting Sort

- *Voraussetzung*: Zahlen in konstantem Intervall $[0, k]$.
- *Verfahren*: Zählen wieviele Vorkommen von jeder Zahl existieren.
- *Komplexität*: $\Theta(n + k)$ Worstcase.

Bucket Sort

- *Voraussetzung*: Zahlen in konstantem Intervall $[a, b]$.
- *Verfahren*: Unterteilen von $[a, b]$ in n gleichgrosse Intevalle, einfügen in Intervalle, Intervalle mit Bubblesort (o. Äh.) sortieren.
- *Komplexität*: $\Theta(n)$ Averagecase bei gleichverteiltem Input.

Zusätzliche Aufgaben

Gebe für jedes der folgenden Probleme einen Algorithmus an, der die Lösung in der verlangten Zeit findet.

1. Sei A ein unsortiertes Array von n Zahlen. Finde $i, j, i \neq j$ in A, so dass $|A[i] - A[j]|$ maximal ist (in $\mathcal{O}(n)$).
2. Sei A ein sortiertes Array von n Zahlen. Finde $i, j, i \neq j$ in A, so dass $|A[i] - A[j]|$ maximal ist (in $\mathcal{O}(1)$).
3. Sei A ein unsortiertes Array von n Zahlen. Finde $i, j, i \neq j$ in A, so dass $|A[i] - A[j]|$ minimal ist (in $\mathcal{O}(n \log n)$).
4. Sei A ein sortiertes Array von n Zahlen. Finde $i, j, i \neq j$ in A, so dass $|A[i] - A[j]|$ minimal ist (in $\mathcal{O}(n)$).
5. Gegeben sind n Jasskarten die nach Zahlenwert aufsteigend sortiert sind. Gebe einen $\mathcal{O}(n)$ Algorithmus an, der die Karten nach Farbe sortiert, so dass alle Elemente der gleichen Farbe aufsteigend sortiert sind. Zum Beispiel: 6♢, 7♡, 8♢, 8♣, 10♡, J♠ sollte sortiert werden zu 6♢, 8♢, 7♡, 10♡, 8♣, J♠
6. Der "Mode" einer Menge von Zahlen ist das Element welches am meisten vorkommt. Die Menge $(4, 6, 2, 4, 3, 1)$ hat den Mode 4. Gib einen effizienten Algorithmus an um den Mode von n Zahlen zu berechnen.
7. Gebe einen effizienten Algorithmus an, um ein Array von n Schlüsseln zu rearrangieren, so dass alle negativen Schlüssel vor allen positiven Schlüsseln kommen. (in place, wie schnell?)
8. Entscheide in $\mathcal{O}(n^2)$ ob es einem unsortierten Array, drei Zahlen a, b, c gibt, so dass $a+b = c$ gilt.