Algorithmen und Datenstrukturen 1

Prof. Dr. Carsten Lecon

Sortieralgorithmen

Bildquelle:: http://www.juergenkreitner.de

Sortieralgorithmen

- Anwendungsgebiete
- Kategorisierung von Sortierverfahren
- Sortieralgorithmen
  - Bubblesort
  - Selectionsort
  - Insertionsort (Wdh.)
  - Shellsort
  - Heapsort
  - Quicksort
  - Mergesort (Wdh.)
  - Weitere Sortierverfahren

Sortieralgorithmen

Typische Anwendungsgebiete

- Listenerzeugung
  - Bsp. Telefon-, Wörterbuch; Tabellenkalkulation, …
- Duplikat-Ermittlung
- Prioritätswarteschlangen
- Computergrafik
  - Umwandlung 3D→2D: z-Wert zeigt Überdeckung an
- Spieleprogrammierung
  - – Auswahl der besten Lösung

Sortieralgorithmen

- Sortierung von mehr oder weniger komplexen Datenobjekten
- Die zu sortierenden Datenobjekte haben oft mehrere Komponenten
- Für Sortierung: ein repräsentatives Attribut

Sortieralgorithmen

Kategorisierung

- Stabilität
- Intern vs. Extern
- In-place vs. out-of-place

Sortieralgorithmen

Stabilität

- Relative Ordnung gleichrangiger Objekte wird nicht verändert.
- Interessant an Vorsortierung
- Beispiel:
  - Notenliste:
    - Liste ist nach Namen sortiert
    - Soll nach Sortierung nach Note auch nach Namen sortiert bleiben

Sortieralgorithmen

Intern vs. extern

- Intern: Gesamte Datenbestand befindet sich während des Sortierens im Hauptspeicher
- Extern: Überwiegender Teil des Datenbestandes befindet sich während des Sortierens überwiegend auf Hintergrundspeichern
  - Zurückführen auf interne Suche:
    - Aufteilung und Zusammenmischung (→ Mergesort)

Sortieralgorithmen

In-place vs. out-of-place

- In-Placement:
- Sortierung direkt auf Eingabestruktur ohne Verwendung zusätzlichen Speicherplatzes
  - Beispiele: Heapsort, Quicksort, Bubblesort, …

Sortieralgorithmen

- Anwendungsgebiete
- Kategorisierung von Sortierverfahren
- Sortieralgorithmen
  - **Bubblesort**
  - Selectionsort
  - Insertionsort (Wdh.)
  - Shellsort
  - Heapsort
  - Quicksort
  - Mergesort (Wdh.)
  - Weitere Sortierverfahren

Sortieralgorithmen

Bubblesort

- Sortieren durch Vertauschen
- Eines der bekanntesten Sortierverfahren
  - Selbst Barack Obama kennt es ;-)
- Größtes Element „blubbert" nach oben
- Doppelschleife
  - Verbesserung: Nur solange weiterlaufen, bis keine Vertauschung mehr erforderlich ist

Sortieralgorithmen

http://www.youtube.com/watch?v=k4RRi_ntQc8

Sortieralgorithmen

```
procedure bubblesort1(A)
  for i=1 to length(A)-1 do
    for j=i+1 to length(A) do
      if A[i] > A[j] then
        swap A[i] ↔ A[j]
      end if
    end for
  end for
end procedure
```

Sortieralgorithmen

```
procedure bubblesort2(A)
  do
    for i=1 to length(A)-1
      if A[i] > A[i+1] then
        swap A[i] ↔ A[i+1]
      end if
    end for
  until keine Vertauschung mehr
end procedure
```

https://www.youtube.com/watch?v=lyZQPjUT5B4

Sortieralgorithmen

- Anwendungsgebiete
- Kategorisierung von Sortierverfahren
- Sortieralgorithmen
  - Bubblesort
  - **Selectionsort**
  - Insertionsort (Wdh.)
  - Shellsort
  - Heapsort
  - Quicksort
  - Mergesort (Wdh.)
  - Weitere Sortierverfahren

Sortieralgorithmen

- ***Selectionsort*** : Sortieren durch Selektion
- Idee:
  - Kleinstes Element suchen und an Anfang stellen (Vertauschen)
  - Ab zweitem Element genauso verfahren
  - Etc.

  - Geht auch andersherum (hinten beginnen mit dem größten Element)
    - Beispiel Tafel: 5 – 1 – 8 – 3 – 9 – 2

Sortieralgorithmen

```
procedure selectionsort(A)
  left=1
  do
    min = left
    for i=left+1 to length(A) do
      if A[i] < A[min] then
        min=i
      end if
    end for
    swap A[min] ↔ A[left]
    left = left+1
  while left < length(A)
end procedure
```

Sortieralgorithmen

```
procedure selectionsort(A)
  right=length(A)
  do
    max = right
    for i=1 to right-1 do
      if A[i] > A[max] then
        max=i
      end if
    end for
    swap A[max] ↔ A[right]
    right = right-1
  while right > 0
end procedure
```

Sortieralgorithmen

- Anwendungsgebiete
- Kategorisierung von Sortierverfahren
- Sortieralgorithmen
  - Bubblesort
  - Selectionsort
  - **Insertionsort (Wdh.)**
  - Shellsort
  - Heapsort
  - Quicksort
  - Mergesort (Wdh.)
  - Weitere Sortierverfahren

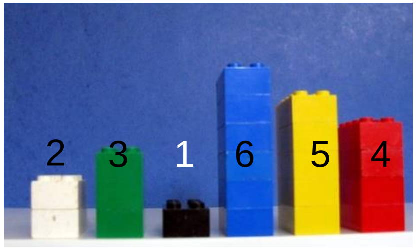
Anwendungen des Sortierproblems

Insertionsort

- Sortieren durch Einfügen
- Einfach, wenig effizient; aber
  - leicht zu implementieren
  - gute Ergebnisse bei vorsortierten oder kleinen Folgen

Bildquelle: http://mathbits.com

- Entspricht bspw. dem Kartenaufnehmen in Kartenspielen
- Beispiel



Algorithmus (in Pseudocode)

```
function insertionsort(A)
  for i=2 to length(A)
    x := A[i]
    j := i
    while( (j>1) and (A[j-1] > x) )
      A[j] := A[j-1]
      j := j-1
    end while
    A[j] = x
  end for
end function
```

Beispielfolge 5,1,8,3,9,2

Sortieralgorithmen

- Anwendungsgebiete
- Kategorisierung von Sortierverfahren
- Sortieralgorithmen
  - Bubblesort
  - Selectionsort
  - Insertionsort (Wdh.)
  - **Shellsort**
  - Heapsort
  - Quicksort
  - Mergesort (Wdh.)
  - Weitere Sortierverfahren

Beispiel Insertionsort

5	**8**	2	1	7	4	6	0	3
5	8	**2**	1	7	4	6	0	3
2	5	8	**1**	7	4	6	0	3
1	2	5	8	**7**	4	6	0	3
1	2	5	7	8	**4**	6	0	3
1	2	4	5	7	8	**6**	0	3
1	2	4	5	6	7	8	**0**	3
0	1	2	4	5	6	7	8	**3**
0	1	2	3	4	5	6	7	8

Shellsort

- Weiterentwicklung von Insertionsort
- Problem bei Insertionsort: Bei weit entfernten Elementen erfolgen viele Verschiebungen
- **Shellsort**: Vorsortierung mit Insertionsort
- Aufteilung des Arrays in 2^n Teile

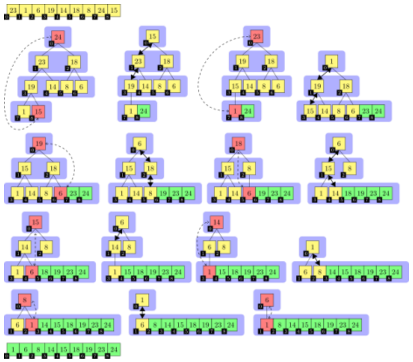
Shellsort

5	8	2	1	7	4	6	0	3	
									Aufteilen in 4 Teilbereiche
5	8	2	1	7	4	6	0	3	↓
									Sortieren
3	4	2	0	5	8	6	1	7	↓
									Aufteilen in 2 Teilbereiche
3	4	2	0	5	8	6	1	7	↓
									Sortieren
2	0	3	1	5	4	6	8	7	↓
									Aufteilen in 1 Teilbereich
2	0	3	1	5	4	6	8	7	↓
									Sortieren
0	1	2	3	4	5	6	7	8	

Sortieralgorithmen

- Anwendungsgebiete
- Kategorisierung von Sortierverfahren
- Sortieralgorithmen
  - Bubblesort
  - Selectionsort
  - Insertionsort (Wdh.)
  - Shellsort
  - **Heapsort**
  - Quicksort
  - Mergesort (Wdh.)
  - Weitere Sortierverfahren

Heapsort



Bildquelle: http://upload.wikimedia.org/wikipedia/commons/thumb/d/dc/Heapsort.svg/600px-Heapsort.svg.png

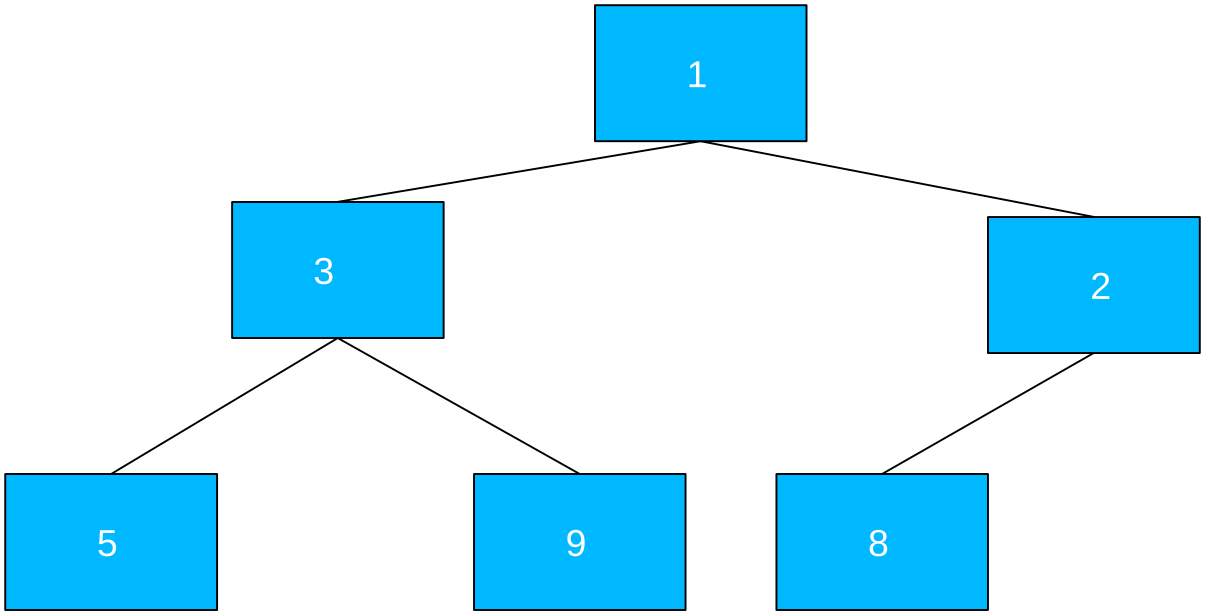
Heap

- Übersicht
- Einführung
- Aufrechterhaltung der Heap-Eigenschaft
- Aufbau eines Heaps

Heap

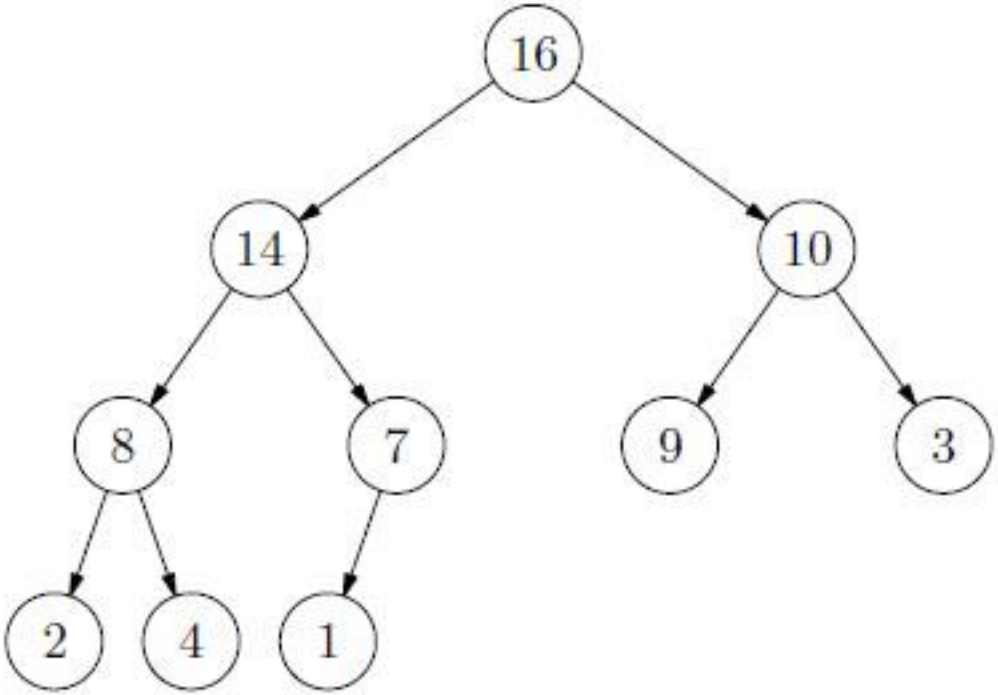
- Lernziele
  - Kennenlernen der neuen Datenstruktur „Heap“
  - Verstehen der Eigenschaften von Heaps
  - Kennen der Algorithmen zur Herstellen dieser Eigenschaften
  - Kennen der Anwendungen von Heaps

Heap: Baum und Array



1	3	2	5	9	8
[1]	[2]	[3]	[4]	[5]	[6]

Heap: Baum und Array, Beispiel 2



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Heap

- Eigentlich kaum neu: Wir kennen (binäre) Heaps bereits: Implementierung von Binärbäumen durch Arrays
- Ein binärer Heap ist eine Arrayobjekt, das als vollständiger Baum angesehen kann.
- Jeder Knoten entspricht einem Array-Element.
- Ein Array A, das einen Heap repräsentiert, hat 2 Attribute:
  - **length(A)**: Anzahl der Elemente im Array
  - **heap-size(A)**: Anzahl der Elemente, die auch tatsächlich zum Heap gehören, also heap-size(A) ≤ length(A)
- Gültige Elemente im Heap: A[1]…A[heap-size(A)]
- Wurzel des Baums ist A[1].

Heap

- Berechnung von Kind- und Elternknoten:
- **parent(i)** : i/2 (nach unten abgerundet)
- **left(i)**: 2i
- **right(i)**: 2i+1

Heap

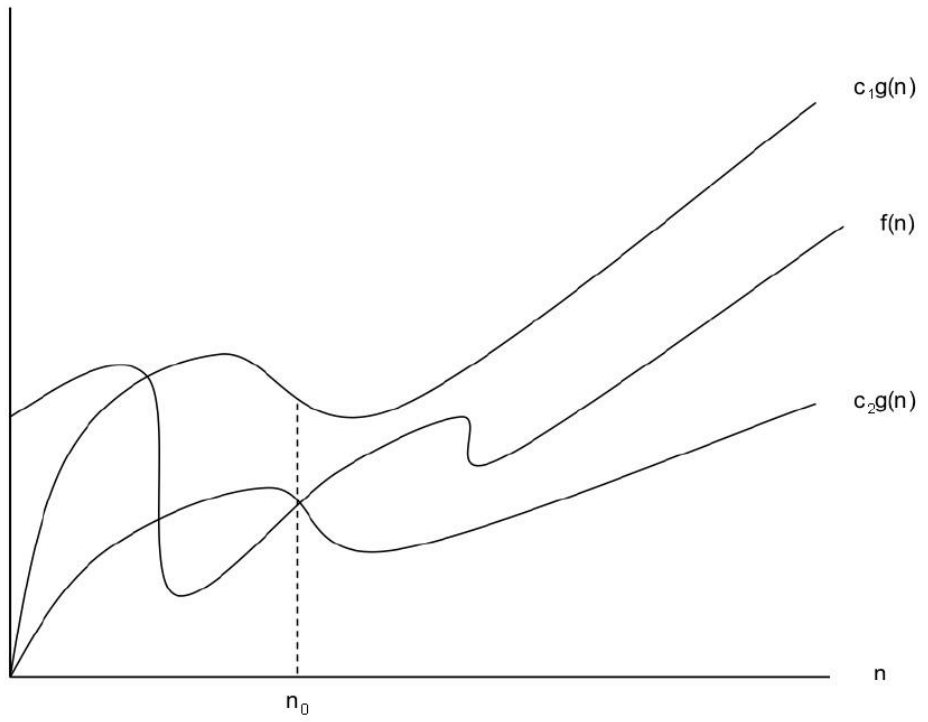
- Anwendungen:
  - Prioritätswarteschlangen (in Betriebssystemen, etc.)
  - Greedy-Algorithmen
  - Heapsort

Heap

- Bemerkungen
  - Da man Heaps als vollständige Binärbaume interpretieren kann, können alle Begrifflichkeiten übernommen werden.
  - Die Höhe eines Knotens ist die Anzahl der Kanten, über die der Knoten auf dem längsten einfachen Weg zu einem Blatt verbunden wird.
  - Die Höhe eines Heaps ist also $\theta(\log n)$.
  - Die Basisalgorithmen sind proportional zur Höhe h und haben damit eine Laufzeit $O(\log n)$.

Wachstum der Laufzeit von Algorithmen

- Θ-Notation



Heap

- **Definition Min-Heap**
- Ein **Min-Heap** ist ein Heap, für dessen Elemente gilt:

$$A[parent(i)] \leq A[i],\ i>1$$

- Die geforderte Eigenschaft heißt **Min-Heap-Eigenschaft**.

  → Alle Knoten des Teilbaums von A[i]
  sind größer/gleich A[i].

Heap

- **Definition Max-Heap**
- Ein **Max-Heap** ist ein Heap, für dessen Elemente gilt:

$$A[parent(i)] \geq A[i], i>1$$

- Die geforderte Eigenschaft heißt **Max-Heap-Eigenschaft**.

  → Alle Knoten des Teilbaums von A[i]
  sind kleiner/gleich A[i].

Heap

- Voraussetzung für Operationen (z.B. Sortieren) auf Heaps:
  - Array muss in binären Heap umgewandelt werden.
- Algorithmus:
  - Beginn in der Mitte des Arrays
  - „Versickern“ der davorliegenden Knoten (bis zum ersten Element)
    - Austausch mit Nachfolgeknoten, falls dieser größer/gleich dem aktuellen Knoten ist
      - Fortführung, bis kein größerer Nachfolgeknoten gefunden wird

Heap

```
function max-heapify(A, i)
  l = left(i)
  r = right(i)
  if l ≤ A.heap-size(A) and A[l] > A[i] then largest = l
  else                                       largest = i
  end if
  if r ≤ A.heap-size(A) and A[r] > A[largest] then largest = r
  end if
  if largest ≠ i then
    Vertausche A[i] und A[largest]
    max-heapify(A, largest)
  end if
end function
```

Heap

- Laufzeit von `max-heapify`:
  - Bis auf rekursiven Aufruf: konstante Laufzeit
  - Je Aufruf ein weiterer Aufruf mit Element einer höheren Ebene
    - →Maximal so viel Aufrufe wie Höhe
    - →$T(n) = O(h) = O(\log n)$

Heap

Aufbau Heap:

```
function build-max-heap(A)
  for i=A.heap-size(A)/2 downto 1 do
    max-heapify(A, i)
  end for
end function
```

Heap

Beispiele:

- Folge (Quelle: Wikipedia)
  - H E A P S O R T

    überführen in einen Max-Heap

- Folge
  - 1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 – 9 – 10

    überführen in einen Max-Heap
    (am Beispiel des Programms)

Heap

- HÖRSAAL-ÜBUNG
- Illustrieren Sie die Operation `build-max-heap` am Beispiel des Arrays
  - 5 – 3 – 17 – 10 – 84 – 19 – 6 – 22 – 9
    - Zeichnen Sie den entsprechenden binären Baum vor und nach der Erstellung des Heaps.

Heap

- Min-Heap:
  - Was muss im Code geändert werden?

- Folge (Quelle: Saake, Sattler: „Algorithmen…“)
  - 5 – 1 – 8 – 3 – 0 – 2

    überführen in einen Min-Heap

Heap

```
function min-heapify(A, i)
  l = left(i)
  r = right(i)
  if l ≤ A.heap-size(A) and A[l] < A[i] then smallest = l
  else                                       smallest = i
  end if
  if r ≤ A.heap-size(A) and A[r] < A[smallest] then smallest = r
  end if
  if smallest ≠ i then
    Vertausche A[i] und A[smallest]
    min-heapify(A, smallest)
  end if
end function
```

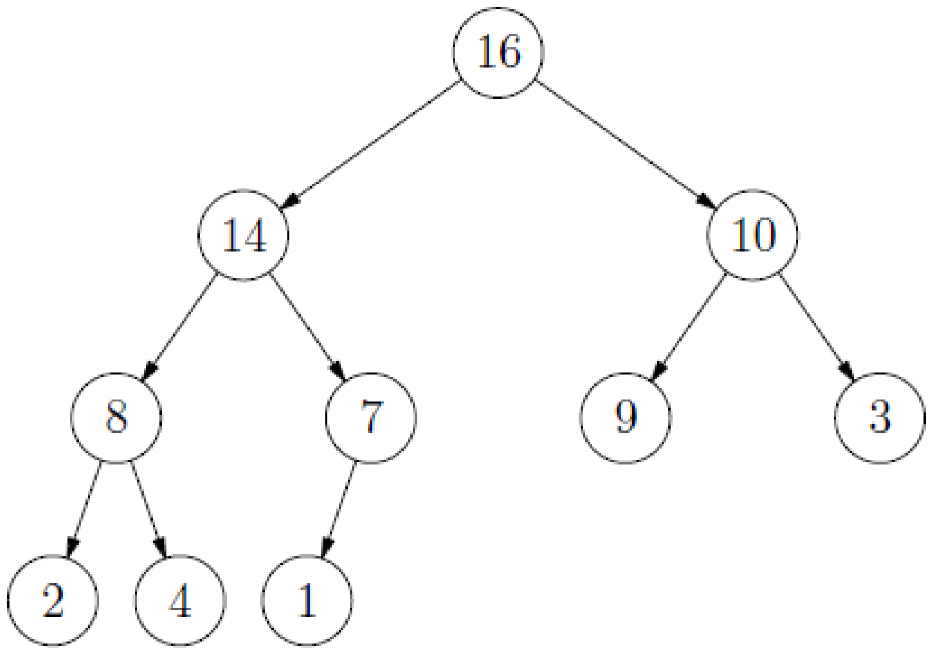
Heap

- Heapsort
- Wie kann man die Datenstruktur „Heap“ zur Sortierung nutzen?
  - A muss zu Beginn Max-Heap sein → **`build-max-heap`**
    - → Größtes Element steht an Position A[1].
  - Aufsteigendes Sortieren durch Verschieben von A[1] nach A[n]
  - Wiederherstellen der Max-Heap-Eigenschaft für A[1]..A[n-1]
  - Wiederholen bis zum trivialen einelementigen Heap A[1]

Heapsort

```
procedure heapsort(A)
  build-max-heap(A)
  for i=heap-size(A) downto 2 do
    exchange A[1] ↔ A[i]
    heap-size(A) = heap-size(A)-1
    max-heapify(A,i)
  end for
end procedure
```

Heapsort – Live-Beispiel



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

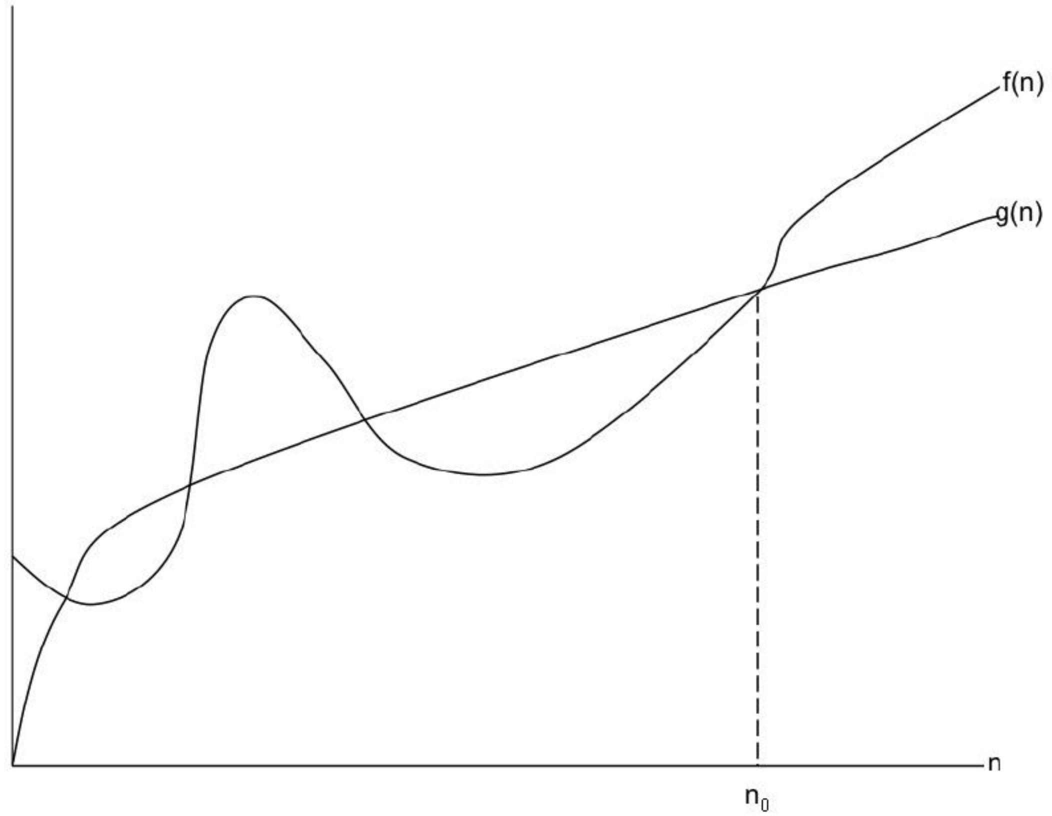
Heapsort

- Mit beschriebenem Verfahren (Max-Heap) wird aufsteigend sortierte Folge erstellt.
  - Mit Min-Heap kann absteigend sortierte Folge erstellt werden
    - Beispiel: Folge 8 – 3 – 2 – 5 – 9

      [Saake, Sattler]

Wachstum der Laufzeit von Algorithmen

- Ω-Notation



Heapsort

- Laufzeit von Heapsort:
- build-max-heap hat Laufzeit von $O(n)$.
- n-1 Aufrufe von max-heapify → $O(n \cdot \log n)$
- Laufzeit ist also $T(n) = O(n \cdot \log n)$.

- Man kann zeigen, dass die Laufzeit im *best case* und *worst case* bei $T(n) = \Omega(n \cdot \log n)$ liegt.

Heap

- Fragen
  - Wie groß ist die maximale und wie groß ist die minimale Anzahl der Elemente in einem Heap der Höhe h?
  - Wo befindet sich in einem Max-Heap das kleinste Element, wenn alle Elemente des Max-Heaps paarweise verschieden sind?
  - Ist ein Feld, das in sortierter Reihenfolge vorliegt, ein Min-Heap?

Sortieralgorithmen

- Anwendungsgebiete
- Kategorisierung von Sortierverfahren
- Sortieralgorithmen
  - Bubblesort
  - Selectionsort
  - Insertionsort (Wdh.)
  - Shellsort
  - Heapsort
  - **Quicksort**
  - Mergesort (Wdh.)
  - Weitere Sortierverfahren

Sortieren

Quicksort

- Prinzip: *Divide & Conquer* (Teile und Herrsche)
- Idee: C.A.R. Hoare: Quicksort. Computer Journal, Vol. 5, 1, pp. 10-15, 1962
- Es gibt verschiedene Varianten…
- Was war 1962?

Sortieren

Exkurs: Das Jahr 1962

- Hardware:
  - Großrechner (Anfänge), Magnetbänder
  - Entwicklung von Chips (vorher Transistoren)
- Programmiersprachen:
  - Algol 60, LISP, COBOL, APL
- Erfindung der Computermaus (X-Y-Indikator)
- Hitliste (Musik): (Hits in 2 Minuten)
  - Tanze mit mir in den Morgen (Gerhard Wendland)
  - Mexiko (Bob Moore)
  - Zwei kleine Italiener (Conny / Jan & Kjeld)
  - Junge, komm bald wieder (Freddy Quinn) (Platz 8)

Sortieren

Quicksort

- Grundidee:
  - A wird in zwei Teilfelder zerlegt, x Wert des Mittelelements (*Pivot*-Element):
    - Alle Werte links von x sind kleiner/gleich x
    - Alle Werte rechts von x sind größer/gleich als x
  - Rekursive Aufteilung der linken und rechten Seite
  - Umsortierung:
    - Von links suchen bis Wert größer/gleich Pivot-Element
    - Von rechts suchen, bis Wert kleiner/gleich Pivot-Element
    - Tauschen der beiden gefundenen Elemente
    - Linken Index erhöhen, rechten Index verringern

Sortieren

- Quicksort: Divide & Conquer
- Divide:
  - Teilen um q: Anordnung der Elemente A[p..r], dass zwei Teil-Arrays mit folgenden Bedingungen entstehen:
    - A[p..q-1] mit $A[i] \leq A[q]$ für alle i aus {p..q-1}
    - A[q+1..r] mit $A[i] \geq A[q]$ für alle i aus {q+1..r}
- Conquer:
  - Sortiere A[p..q-1] und A[p+1..r] durch rekursiven Aufruf
- Merge:
  - Kein Zusatzaufwand, da in-place sortiert wird

Sortieren

```
procedure quicksort(A, left, right)
  if (right ≤ left) then return end if
  i=left
  j=right
  Bestimme index; z.B. x = (left+right)/2
  x =A[index]
  while (i ≤ j) do
    while A[i] < x do i=i+1 end while
    while A[j] > x do j=j-1 end while
    if i ≤ j then
      swap A[i] ↔ A[j]
      i=i+1
      j=j-1
    end if
  end while
```

Sortieren

```
        quicksort(A, left, j)
        quicksort(A, i, right)
    end procedure
```

Sortieren

```
procedure quicksort2(A, left, right)
  if (right ≤ left) then return end if
  i=left
  j=right-1

  x =A[right]
  do
    while A[i] ≤ x and i<right do i=i+1 end while
    while A[j] ≥ x and j>left do j=j-1 end while
    if i < j then
      swap A[i] ↔ A[j]
      // Kleineres Element nach vorne,
      // größeres Element nach hinten
    end if
  while i<j
```

Sortieren

```
        if A[i] > x then
          swap A[i] ↔ A[right]
        end if
        quicksort2(A, left, j)
        quicksort2(A, i+1, right)
      end procedure
```

Sortieralgorithmen

Eigenschaften

Verfahren	Stabilität	Vergleiche (Ø)
SelectionSort	instabil	$\approx n^2/2$
InsertionSort	stabil	$\approx n^2/4$
Bubblesort	stabil	$\approx n^2/2$
Mergesort	stabil	$\approx n \log_2 n$
Quicksort	instabil	$\approx 2n \ln 2 \approx 1.38\, n \log_2 n$

Sortieralgorithmen

SelectionSort

- Jeder Durchlauf: ab `left` n-1 Vergleiche
- → $(n-1) + (n-2) + (n-3) + \ldots + 2 + 1 = n(n-1)/2 \approx n^2/2$
- Anzahl der Vergleiche ist immer gleich

Sortieralgorithmen

InsertionSort

- Gesamte Folge wird durchlaufen
- Gemerktes Element wandert ggf. nach vorne
- Ungünstigster Fall:
  - i-1 Positionen verschieben für i-ten Durchgang
- Günstigster Fall:
  - Keine Verschiebung
- Im Durchschnitt: Halber Weg wird zurückgelegt
  (n-1)/2 Operationen
  - Hälfte der Vergleiche von SelectionSort

Sortieralgorithmen

Bubblesort

- Jeder Durchlauf: Suchen des größten Elements
- Ungünstigster Fall:
  - n Durchläufe
    - n-i Vergleiche im i-ten Durchlauf
- Günstigster Fall (Folge ist sortiert):
  - Ein Durchlauf
- Durchschnittlicher Fall wie SelectionSort, aber:
  - Bubblesort ist stabil

Sortieralgorithmen

Mergesort

- Mischen: linear (Größe) des Teilfeldes
- Vergleiche: max. log n Rekursionslevel
  - → O(n log n)
- Besonderheit:
  - Es wird zusätzlicher Speicher benötigt

Sortieralgorithmen

Quicksort

- Laufzeitbestimmung ähnlich wie Mergesort
  - aber keine Hilfsfelder erforderlich
- Herleitung aufwändig