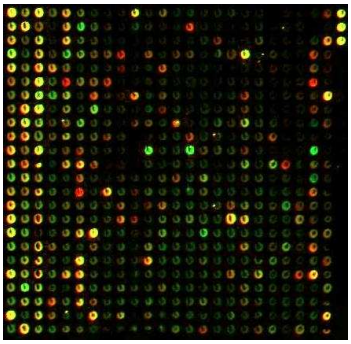
DNA-Array oder DNA-Chip



- Jeder Punkt des Feldes repräsentiert ein Gen g_i des Menschen.
- Ein Gen ist der Bauplan eines molekularen Bausteins unseres Körpers.
- Mittels eines DNA-Chips kann man gleichzeitig für viele Gene (n) messen, in welchem Maße bestimmte Bausteine unseres Körpers zu einem bestimmten Zeitpunkt produziert werden (welche Gene in welchem Maße exprimiert werden).

- Mittels dieser Chips kann man zum Beispiel die Gen-Expression verschiedener Zellkulturen (Tumorzellen mit normalen Zellen) miteinander vergleichen:
  Gene, die stärker in der ersten Zellart exprimiert werden, leuchten intensiv rot.
  Gene, die stärker in der zweiten Zellart exprimiert werden, leuchten intensiv grün.
  Die Differenz a[i] der Gen-Expression eines Gens g_i in den beiden Zellarten kann man in Form einer reellen Zahl angeben: $a[i] \in [-1,1]$ für alle i = 0, 1, ... , n-1.

Aufgabe: Sortiere die Gene aufsteigend nach den Werten der Differenzen a[i].

Gegeben ein Feld a der Größe n mit reellen Zahlen a[i], i = 0, 1, ... ,n.
Sortiere die Zahlen in aufsteigender Reihenfolge.

Nach dem Sortieren gilt:

$$a[0] \le a[1] \le a[2] \le ... \le a[n-1]$$

Allgemeine Formulierung des Sortier-Problems:

Gegeben eine Menge A von Elementen aus einem Universum U, auf dem es eine Ordnungsrelation „≤“ gibt. Sortiere die Elemente der Menge A in aufsteigender (fallender) Reihenfolge bezüglich der Ordnungsrelation.

Beispiel 1: Das Universum könnte zum Beispiel die Menge aller englischen Wörter sein und die Ordnungsrelation könnte die lexikographische Ordnung für „Strings“ sein.

Beispiel 2: Das Universum U ist die Menge aller Studenten der Universität. Diese werden nach ihrer Matrikelnummer oder ihren Geburtstagen sortiert.

Wir werden im folgenden Abschnitt verschiedene Sortier-Verfahren kennenlernen und dabei einige wichtige Problemlösestrategien diskutieren.

Ein triviales Sortierverfahren

k	10	0	2	6	9	3	1	4	5	7	8	10

```
int index_of_minimum(int [ ] a, int min_index, int max_index) {
  int index = min_index;
  for (int i = min_index; i < max_index ; i++)
                          if (a[index] > a[i])  index = i;
  return index;
}
```

Die obige Funktion sucht im Teilfeld von Feld „a“ zwischen dem Index „min_index“ und „max_index“ nach der Feldkomponente, die die kleinste Zahl enthält, und gibt deren Index zurück. Mit Hilfe dieser Funktion können wir „a“ wie folgt sortieren:

```
void trivial_sort(int [ ] a, int size_of_a) {
  for (int i = 0; i < size_of_a - 1 ; i++) {
     int  j  = index_of_minimum(a, i+1, size_of_a); // Suche nächstes Min.
     int  k = a[ i ];                               // Zwischenspeicherung a[i]
     if ( k > a[ j ] ) { a[ i ]  = a[ j ];          // die nächste sortierte Zahl
                         a[ j ]  = k;  }
   }
}
```

Ein triviales Sortierverfahren

10	10	9	8	7	6	5	4	3	2	1	0

k

Der Zeitaufwand für die for-Schleife im i-ten Schritt ist gleich

$$c(n-i)$$

wobei c > 0 eine Konstante ist und n (size_of_a) die Größe des Feldes a.

Die Gesamtlaufzeit von trivial_sort ist daher:

$$c\sum_{i=0}^{n-1}(n-i) = c(\sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i) = c(n^2 - \frac{n^2-n}{2}) = c\frac{n^2+n}{2} = \Theta(n^2)$$

Satz [1]:

Die (worst-case) Laufzeit von „trivial_sort“ ist in $\Theta(n^2)$, wobei n die Größe der Eingabe (Größe des Feldes) ist.

Divide & Conquer

Problem T(n) = Konkatenation (Problem B(n) , Problem T(n-1))

B = Berechne das Minimum von n Zahlen

$$T(n) = \begin{cases} d & n = 1 \\ c_1 + c_2 n + T(n-1) & n > 1 \end{cases}$$

Divide & Conquer

Idee:

Zerlege das ursprüngliche Problem der Größe n in kleinere Teilprobleme. Man löse die Teilprobleme und setze die Gesamtlösung aus den Lösungen der Teilprobleme zusammen.

Zum Beispiel könnten wir das Problem der Größe n in zwei Teilprobleme der Größe n/2 zerlegen.

T(n) = ? (T(n/2), T(n/2))

MergeSort

T(n) = Merge (T(n/2), T(n/2))

Frage 1: Wie können wir das Sortierproblem zerlegen?

10	2	6	9	3	1	4	5	7	8	0

10	2	6	9	3	1	4	5	7	8	0

Wir zerlegen das Feld in zwei Teilfelder der Länge (ungefähr) n/2!

Frage 2: Nehmen wir an, dass die beiden Teilfelder sortiert sind?

1	2	3	6	9	10	0	4	5	7	8

Wie können wir aus den Teillösungen die Gesamtlösung effizient berechnen?

Man mische die sortierten Halbfelder zu dem sortierten Feld!

1	2	3	6	9	10	0	4	5	7	8

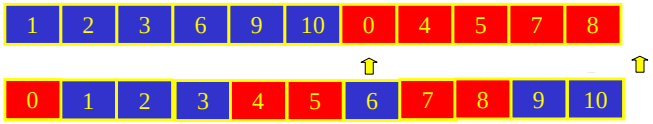
⇧ ⇧

0	1	2	3	4	5	6	7	8	9	10

MergeSort


```
void merge(int [ ] a, int s1, int e1, int s2, int e2) {
  int b = new int[e2-s1+1];                  // Speicherplatz für sortierte Folge
  int i = s1, j = s2, k = 0;                 // Indexzähler
  for (  ; i < e1+1 && j < e2+1 ; k++ ) {
     if (a[i] <= a[j]) {  b[k] = a[i];  i++; }
     else              {  b[k] = a[j]; j++; }
  }
  for (   ;  i < e1+1; k++,i++)   b[k] = a[i];  //  Leere den Rest des erstes Teilfelds
  for (   ;  j < e2+1; k++,j++)   b[k] = a[j];  //  Leere den Rest des zweiten Teilfelds
  for ( k = 0, i = s1 ;  i < e2+1; i++,k++)   a[i] = b[k];   // Kopiere sortierte Folge
}
```

Man mische die sortierten Halbfelder zu dem sortierten Feld!



MergeSort


```
void merge(int [] a, int s1, int e1, int s2, int e2) {
  int b = new int[e2-s1+1];                  // Speicherplatz für sortierte Folge
  int i = s1, j = s2, k = 0;                 // Indexzähler
  for (  ; i < e1+1 && j < e2+1 ; k++ ) {
     if (a[i] <= a[j]) {  b[k] = a[i];  i++; }
     else              {  b[k] = a[j]; j++; }
  }
  for (   ;  i < e1+1; k++,i++)   b[k] = a[i];  //  Leere den Rest des erstes Teilfelds
  for (   ;  j < e2+1; k++,j++)   b[k] = a[j];  //  Leere den Rest des zweiten Teilfelds
  for ( k = 0, i = s1 ;  i < e2+1; i++,k++)   a[i] = b[k];   // Kopiere sortierte Folge
}
```

Unter der Annahme, dass die beiden Teilfelder a[s1,e1] und a[s2,e2] bereits (korrekt) sortiert sind, folgt die Korrektheit des Algorithmus „merge“ aus der Tatsache, dass die Indizes i und j jeweils die kleinsten noch nicht in „b“ einsortierten Elemente der beiden Teilfelder repräsentieren und dass das Minimum dieser beiden Zahlen folglich die kleinste noch nicht verarbeitete (Zahl des Rests sein muss.

Die Laufzeit von „merge“ für zwei Teilfelder der Länge n/2 ist $\Theta(n)$.

MergeSort


```
void merge_sort(int [ ] a, int min, int max) {
  if (max-min > 0) {                                  // Gibt es was zu tun ?
     merge_sort(a, min, min + ((max-min)/2));         // Sortiere das erste Teilfeld
     merge_sort(a, min +((max-min)/2) +1, max); // Sortiere das zweite Teilfeld
     merge(a, min, min + ((max-min)/2), min + ((max-min)/2) +1, max);
   }
}
```

Laufzeit des Algorithmus „merge_sort“ für ein Feld der Größe n:

$$T(1) = c$$

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

$$T(n) = 4T(\frac{n}{4}) + dn + dn$$

$$T(n) = 2^{\log n} T(1) + dn \log n \; = cn + dn \log n \in \Theta(n \log n)$$

Satz [2]:

Die Laufzeit des MergeSort-Algorithmus für ein Feld der Größe n ist Θ(n log n).

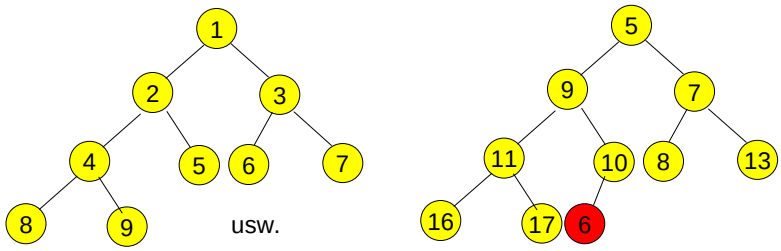
HeapSort

Heap-Sort:

Ein Heap ist ein binärer Baum mit den folgenden Eigenschaften:

[1] Für jeden Knoten v des Heaps mit einem Vater v.parent (!= NULL) gilt:
v.key ≥ v.parent->key

[2] Der Heap wird beginnend mit der Wurzel in einer bestimmten Reihenfolge (möglichst weit oben und soweit links wie möglich) Schicht für Schicht aufgefüllt (siehe Abbildung unten links, die Zahlen geben die Reihenfolge an):



Durch das Einführen eines neuen Elements (an der richtigen Position) kann die Heap-Eigenschaft verletzt werden.

HeapSort

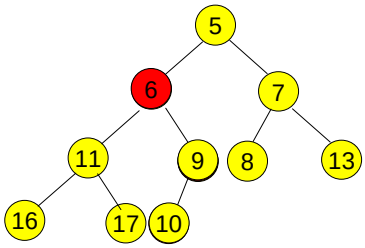
Wir nehmen an, dass v ein Zeiger auf einen (den eingefügten) Knoten ist.

Die Reparatur erfolgt durch Hochschieben des neuen Knotens v, d.h., solange

v->key < v->parent->key

vertauscht man Vater und Sohn:

```
int help = v->key;
v->key = v->parent->key;
v->parent->key = help;
v = v->parent;
```



HeapSort

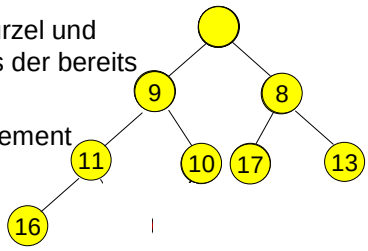
Aufgrund der Reihenfolge der Einfügungen hat ein Heap mit n Elementen Höhe log(n). Da man bei jeder Einfügung höchstens den Pfad vom Einfüge-Knoten bis zur Wurzel wandert und dabei pro Knoten nur konstante Zeit benötigt, hat jede Einfüge-Operation im schlimmsten Fall Laufzeit O(log n). Ein Heap mit n Elementen kann daher in Zeit O(n log n) aufgebaut werden.

Das kleinste Element in einem Heap ist in der Wurzel gespeichert.

Mit Hilfe eines Heaps kann man nun die n Elemente wie folgt sortieren:

Solange der Heap nicht leer ist, führt man die folgenden Operationen aus:

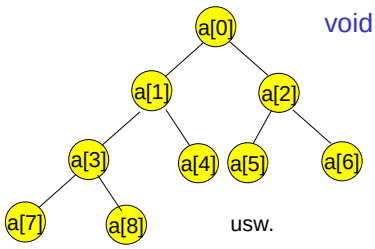
- Entnehme das Element in der Wurzel und speichere es am Ende des Feldes der bereits sortierten Elemente.
- Bewege das zuletzt eingefügte Element in die Wurzel.
- Schiebe das Element solange nach unten, bis die Heap-Eigenschaft wiederhergestellt ist.



5 6 7 usw.

Wie bewegen wir die in der Wurzel eingefügten Elemente auf ihren Platz?

Wir simulieren den Heap mit einem Feld a[0,n-1]:

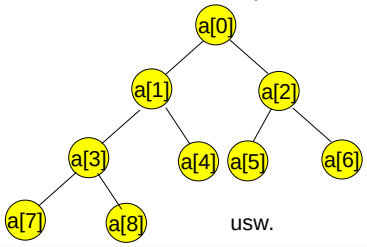


Das linke Kind des Knotens mit Index i hat den Index 2i + 1.
Das rechte Kind des Knotens mit Index i hat den Index 2i + 2.

```
void shift_down( int [ ] a, int i, int last) {
    bool finished = FALSE;
    int k ;
    while (finished != TRUE) {
        if ((2*i+2) < last) {
            if (a[2*i+2] < a[2*i+1]) k = 2*i+2;
            else                     k = 2*i+1;
        }  else if ((2*i+1) < last)   k = 2*i+1;
                else  return;
        if (a[ i ] > a[ k ]) {
            int help = a[ i ];
            a[ i ] = a[ k ];
            a[ k ] = help;
            i = k;
        }  else   finished = TRUE;
    }
}
```


Wie machen wir aus dem Feld einen Heap?

Wir simulieren den Heap mit einem Feld a[0,n-1]:

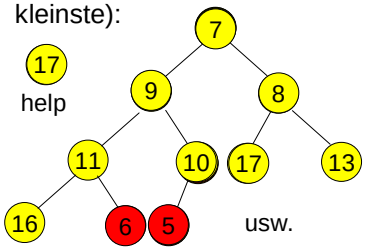



```
void heapify( int [ ] a, int size_of_a) {   // macht einen Heap aus Feld a
   for (int i = size_of_a - 1; i >= 0;  i--) shift_down(a, i, size_of_a);
}
```

Die obige Funktion baut einen Heap, in dem sie Baumschicht für Baumschicht die Heap-Eigenschaft herstellt. Hierbei beginnen wir mit der untersten Schicht. Die Elemente in der untersten Baumschicht können nicht nach unten durchgereicht werden. Daher kann man die obige „for“-Schleife durch eine effizientere ersetzen, in der erst nur die Elemente ab der zweiten Schicht getestet werden.

Wie sortieren wir mit einem Heap (Feld)?

O.B.d.A sortieren wir die Zahlen im Feld fallend (das letzte Element ist das kleinste):




```
void heap_sort( int [ ] a, int size_of_a) {   // Sortiert das Feld a
        heapify(a, size_of_a);                 // Baue den Heap
        for (int i = size_of_a - 1; i > 0 ; i--) {
            int help = a[i];                   // Rette den Schlüssel von a[i]
            a[ i ] = a[0];                     // ein weiteres Element sortiert
            a[0] = help;                       // neuer Schlüssel für die Wurzel
            shift_down(a, 0 , i);              // Reparatur der Heap-Eigensch.
        }
    }
```


Satz [3]:

Heap-Sort sortiert ein Feld mit n Elementen in Zeit O(n log n).

Beweis:

Der Aufbau des Heaps „heapify“ hat eine Laufzeit von O(n log n).

Jede Iteration i der „for“-Schleife hat im „worst-case“ Laufzeit O(log n), wobei wir in jedem Durchgang ein Element einsortieren. ■

Quick-Sort

Idee: Man wähle zufällig ein Element e aus den n Elementen der Menge E und man vergleiche alle anderen Elemente mit e.
Mit Hilfe dieser Vergleiche zerlegt man E in zwei Teilmengen:

Die Menge $E_{\leq e}$ mit Schlüssel kleiner gleich dem Schlüssel von e.
Die Menge $E_{>e}$ mit Schlüssel größer dem Schlüssel von e.

Man führe rekursiv die gleiche Prozedur für $E_{>e}$ und $E_{\leq e}$ aus.

$$QS(E) = Konkatenation(QS(E_{\leq e}), e, QS(E_{>e}))$$

Die Zahl der Elemente in den beiden Mengen $E_{\leq e}$, $E_{>e}$ hängt natürlich von e ab. Wenn man (ständig) Pech hat (im worst case), ist eine der Mengen immer leer, z.B. $E_{>e}$, und der Algorithmus hat die gleiche Komplexität ($O(n^2)$) wie „trivial_sort".

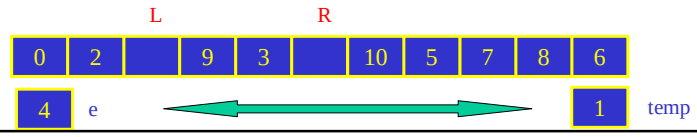
Falls man immer durch die Wahl von e zwei ungefähr gleich mächtige Mengen produziert, dann erhält man als Laufzeit:

$$QS(n) = 2QS(n/2) + cn = O(n \log n)$$

Quick-Sort

QuickSort


```
void quick_sort(int [ ] a, int start, int end) {
   int length = end - start +1;
   if( length > 1) {
      int  L = start;
      int  R = end;
      int  e = a[L];                          // wir wählen e als die erste Zahl im Feld
      int temp = a[R];
      while (L < R) {
          if (temp < e) {  a[L] = temp; L++; temp = a[L];  }
          else           {  a[R] = temp; R-- ; temp = a[R]  }
      }
      a[L] = e;
      if ( (L-1) > start)   quick_sort(a, start, L-1);
      if ((R+1) < end)   quick_sort(a, R+1, end);
   }
}
```



QuickSort


```
void quick_sort(int [ ] a, int start, int end) {
   int length = end - start +1;
   if( length > 1) {
      int  L = start;
      int  R = end;
      int  e = a[L];                          // wir wählen e als die erste Zahl im Feld
      int temp = a[R];
      while (L < R) {
          if (temp < e) {  a[L] = temp; L++; temp = a[L];  }
          else           {  a[R] = temp; R-- ; temp = a[R]  }
      }
      a[L] = e;
      if ( (L-1) > start)   quick_sort(a, start, L-1);
      if ((R+1) < end)   quick_sort(a, R+1, end);
   }
}
```

Die Laufzeit von QuickSort ist proportional zur Zahl der Vergleiche zweier Feldelemente, die in der while-Schleife von allen rekursiven Aufrufen ausgeführt werden. Man beachte, dass jedes Paar von Feldelementen höchstens einmal verglichen wird.

QuickSort

21

Der Einfachheit halber benennen wir die Feldelemente in a um:

$z_1 < z_2 < ... < z_n$ so dass z_i das i-t kleinste Element ist.

Ferner bezeichnen wir mit Z_{ij} die folgende Menge von Feldelementen:

$Z_{ij} = \{ z_i , z_{i+1} , ... , z_j \}$

Unsere Analyse basiert wieder auf der Untersuchung von Zufallsvariablen:

$$X_{ij} = \begin{cases} 1 & \text{falls } z_i \text{ und } z_j \text{ miteinander verglichen werden} \\ 0 & \text{sonst} \end{cases}$$

Die Zahl der Vergleiche X ist gleich

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$$

Daher ist die erwartete Zahl E[X] von Vergleichen (Laufzeit) gleich

$$E[X] = E\left[\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij} \right] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} prob(\{X_{ij} = 1\})$$

QuickSort

22

prob($\{X_{ij} = 1\}$) = prob(z_i wird mit z_j verglichen)

Da $z_1 < z_2 < ... < z_n$ gilt: Sei x das erste Pivot-Element das aus der Menge $Z_{ij} = \{ z_i , z_{i+1} , ... , z_j \}$ gewählt wird:

Ist $x = z_i$ oder z_j , so werden z_i oder z_j miteinander verglichen.

Ist x aus der Menge $Z_{i+1, j-1}$, dann gilt für die entstehenden Teilmengen.

Eine der beiden Teilmengen, die durch den Vergleich mit dem Pivot-Element x gebildet werden, enthält z_i und die andere z_j , d.h., z_i und z_j werden nicht miteinander verglichen.

prob($\{X_{ij} = 1\}$) = prob(z_i wird mit z_j verglichen)
= prob(z_i oder z_j wird als erstes Pivot-Elem. aus Z_{ij} gewählt)
= prob(z_i wird als erstes Pivot-Element aus Z_{ij} gewählt) +
prob(z_j wird als erstes Pivot-Element aus Z_{ij} gewählt)

Jedes Element von Z_{ij} wird mit gleicher WSK als erstes gezogen!

$$prob(\{X_{ij} = 1\}) = \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$$

$$E[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} prob(\{X_{ij} = 1\}) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Wir führen einen neuen Index k = j - i in die obige Gleichung ein:

$$E[X] = \sum_{i=1}^{n} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{2}{k} < \sum_{i=1}^{n} c \log n$$

$$= c \sum_{i=1}^{n} \log n \in O(n \log n)$$

Satz [4]:

Die erwartete Laufzeit von QuickSort für n Elemente ist O(n log n).

Beweis: (siehe oben). ■

Bucket-Sort

Falls die Eingabemenge gewisse Eigenschaften besitzt, kann man diese Mengen unter bestimmten Voraussetzungen in linearer Zeit sortieren (siehe Übungsblatt 5). Sind die n Schlüssel zum Beispiel ganze Zahlen zwischen 0 und n, so kann mit Counting-Sort die Schlüssel in linearer Zeit sortieren.

Wir betrachten jetzt den Fall, dass alle Schlüssel aus einem bestimmten Intervall [a,b) der reellen Zahlen stammen und uniform verteilt sind.

Uniforme Verteilung:

Für jede Zahl x∈[a,b) aus der vorgegebenen Menge gilt: Die WSK, dass x aus einem bestimmten Teilintervall [c,d) von [a,b) ist, ist

$$prob(x \in [c,d]) = \frac{d-c}{b-a}$$

Der Einfachheit halber nehmen wir o.B.d.A. an, dass die Zahlen aus dem Intervall [0,1) stammen. Dann gilt für die obige Wahrscheinlichkeit:

$$prob(x \in [c,d)) = d - c$$

Für c = i/n und d = (i+1)/n folgt also:

$$prob(x \in \left[\frac{i}{n}, \frac{i+1}{n}\right)) = \frac{1}{n}$$

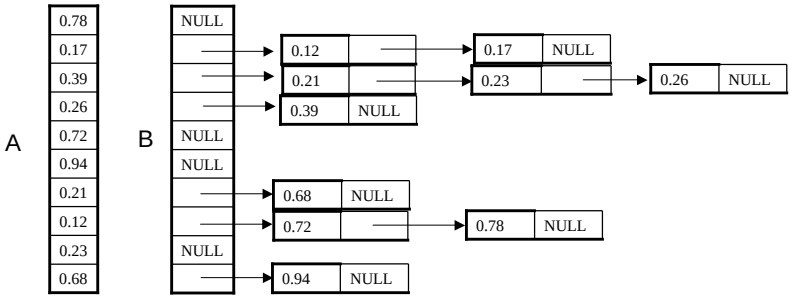
für alle i ∈ {0, 1, ... , n-1}.

Idee von Bucket-Sort

Man teile das Intervall [0,1) in n gleich große Intervalle, die **Buckets**.
Dann ordne man die n Schlüssel ihren Buckets zu und sortiere die Elemente in den Buckets mit „trivial_sort“. Schließlich geht man in der entsprechenden Reihenfolge durch alle Buckets und hängt die sortierten Listen zusammen.

Aufgrund der uniformen Verteilung sollten die Elemente gleichmäßig über alle Buckets verteilt sein.

Eine geeignete Datenstruktur für Bucket-Sort wäre zum Beispiel ein Feld B der Größe n, wobei jedes Feldelement einen Zeiger auf eine Liste von Elementen enthält:



Sei n_i die Zahl der im Bucket i gespeicherten Elemente. Die Laufzeit von Bucket-Sort wird durch die folgende Rekursionsgleichung gegeben (n = size_of_a):

$$T(n) = cn + \sum_{i=0}^{n-1} dn_i^2$$

Die erwartete Laufzeit ist folglich:

$$E[T(n)] = E\left[cn + \sum_{i=0}^{n-1} dn_i^2\right] = E[cn] + d\sum_{i=0}^{n-1} E[n_i^2] = cn + d\sum_{i=0}^{n-1} E[n_i^2]$$

Auf der folgenden Folie zeigen wir, dass $E[n_i^2] = 2 - 1/n$ ist. Hieraus folgt:

$$E[T(n)] = cn + 2dn - d \in O(n)$$

Wir führen die folgenden Zufallsvariablen (Indikatorvariablen) ein:

$$X_{ij} = \begin{cases} 1 & \text{falls a[j] Bucket b[i] zugeordnet wird} \\ 0 & \text{sonst} \end{cases}$$ für i = 0, 1, ... , n-1 und j = 0, 1, ... , n-1

Es gilt:

$$n_i = \sum_{j=0}^{n-1} X_{ij}$$

Hieraus folgt:

$$E[n_i^2] = E\left[\left(\sum_{j=0}^{n-1} X_{ij}\right)^2\right] = E\left[\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} X_{ij}X_{ik}\right] = E\left[\sum_{j=0}^{n-1} X_{ij}^2 + \sum_{j=0}^{n-1}\sum_{\substack{k=0 \\ k\neq j}}^{n-1} X_{ij}X_{ik}\right]$$

$$= \sum_{j=0}^{n-1} E[X_{ij}^2] + \sum_{j=0}^{n-1}\sum_{\substack{k=0 \\ k\neq j}}^{n-1} E[X_{ij}X_{ik}] = \sum_{j=0}^{n-1}\frac{1}{n} + \sum_{j=0}^{n-1}\sum_{\substack{k=0 \\ k\neq j}}^{n-1} E[X_{ij}X_{ik}]$$

$$E[X_{ij}^2] = 1*\frac{1}{n} + 0*(1-\frac{1}{n}) = \frac{1}{n}$$

nächste Seite

$$E[n_i^2] = \sum_{j=0}^{n-1}\frac{1}{n} + \sum_{j=0}^{n-1}\sum_{\substack{k=0 \\ k\neq j}}^{n-1} E[X_{ij}X_{ik}] = \sum_{j=0}^{n-1}\frac{1}{n} + \sum_{j=0}^{n-1}\sum_{\substack{k=0 \\ k\neq j}}^{n-1} E[X_{ij}]E[X_{ik}]$$

Da k≠j, sind die Zufallsvariablen X_{ij} und X_{ik} unabhängig

$$E[n_i^2] = \sum_{j=0}^{n-1}\frac{1}{n} + \sum_{j=0}^{n-1}\sum_{\substack{k=0 \\ k\neq j}}^{n-1}\frac{1}{n}\frac{1}{n} = \sum_{j=0}^{n-1}\frac{1}{n} + \sum_{j=0}^{n-1}\sum_{\substack{k=0 \\ k\neq j}}^{n-1}\frac{1}{n^2} = n\frac{1}{n} + n(n-1)\frac{1}{n^2}$$

$$= 1 + 1 - \frac{1}{n} = 2 - \frac{1}{n}$$

Satz [5]:

Unter der Annahme, dass die Schlüssel der n zu sortierenden Elemente reelle Zahlen aus einem Intervall [a,b) sind, welche uniform verteilt sind, kann man mit Bucket-Sort die n Elemente in erwarteter Zeit O(n) sortieren.

Untere Schranken für das Sortieren

Die O(n log n) Sortierverfahren, die wir in diesem Abschnitt kennengelernt haben, basieren alle auf Vergleichen von Schlüsseln.

Beim vergleichenden Sortieren verwenden wir nur Vergleiche von Elementen, um die Ordnung der Eingabemenge ($a[0]=a_0$, $a[1]=a_1$, ... , $a[n-1]=a_{n-1}$) zu berechnen: Genauer: Gegeben zwei Elemente (die Schlüssel zweier Elemente) a_i und a_j. Wir führen einen Test $a_i < a_j$, $a_i \le a_j$, $a_i = a_j$, $a_i > a_j$ oder $a_i \ge a_j$ aus.

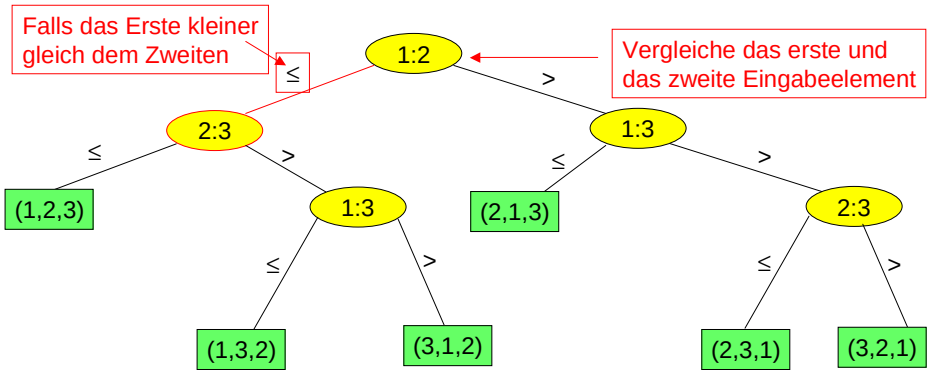
Wir nehmen in diesem Abschnitt an, dass alle Schlüssel verschieden sind, d.h., Vergleiche der Form $a_i = a_j$ sind sinnlos.

Unter dieser Annahme sind Vergleiche der Form $a_i \le a_j$, $a_i \ge a_j$ und $a_i < a_j$, $a_i > a_j$ äquivalent, da sie gleichwertige Informationen liefern.

Das Entscheidungsbaummodell

Der Ablauf von vergleichenden Sortierverfahren kann mittels Entscheidungsbäumen abstrakt dargestellt werden. Ein Entscheidungsbaum ist ein binärer Baum, der alle möglichen Folgen von Vergleichen eines vergleichenden Sortier-Algorithmus für alle möglichen Eingabenmengen einer bestimmten Größe n repräsentiert. Alle anderen Aspekte des Algorithmus werden ignoriert.

Untere Schranken für das Sortieren

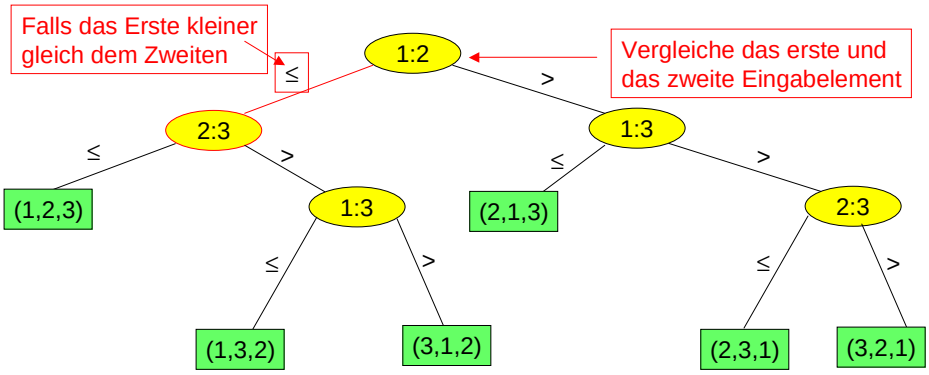


Das Beispiel zeigt einen Entscheidungsbaum für ein triviales Sortierverfahren mit drei Elementen. Die Knotenbeschriftung i:j zeigt an, dass das i-te Element mit dem j-ten Element verglichen werden soll. Je nachdem, wie der Vergleich endet, folgt man der entsprechenden Kante (zum nächsten Vergleich oder zum Resultat des Sortiervorgangs). Die Blätter repräsentieren die sortierten Folgen (Permutationen) der Eingabeelemente:

$(\pi(0), \pi(1), \dots , \pi(n-1))$ so dass $a_{\pi(0)} < a_{\pi(1)} < \dots < a_{\pi(n-1)}$

Untere Schranken für das Sortieren

31

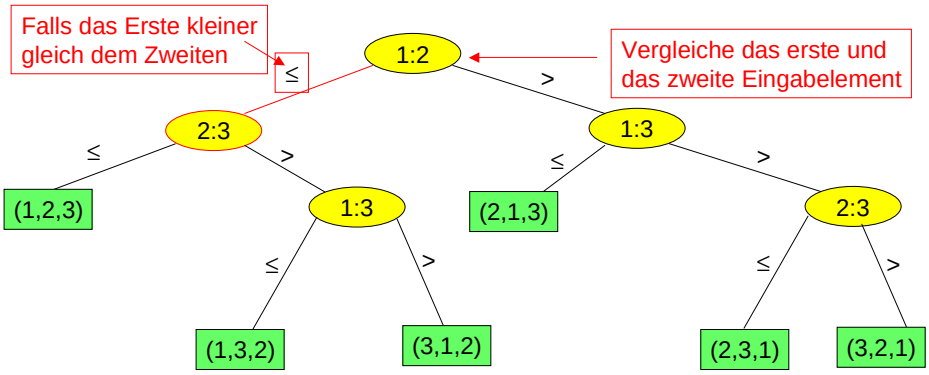


Da jeder korrekte Sortier-Algorithmus in der Lage sein muss, jede Permutation der Eingabeelemente zu produzieren, so muss der Entscheidungsbaum eines solchen Algorithmus für die Problemgröße n über n! Blätter verfügen.

Jedes dieser Blätter muss von der Wurzel aus durch einen entsprechenden Pfad erreichbar sein, der zu einer tatsächlichen Ausführung des Algorithmus auf einer entsprechenden Eingabemenge korrespondiert.

Untere Schranken für das Sortieren

32



Die Länge des längsten Pfades von der Wurzel des Entscheidungsbaumes zu irgend einem Blatt repräsentiert die Zahl der Vergleiche im „worst-case“ (die worst-case Laufzeit) des Sortier-Algorithmus für die entsprechende Problemgröße n. Daher entspricht die Höhe des Baumes der „worse-case“ Performanz des entsprechenden Algorithmus. Folglich ist eine untere Schranke für die Höhe aller möglichen Entscheidungsbäume, in denen jede Permutation als ein Blatt erscheint, auch eine untere Schranke für die Laufzeit von Sortier-Verfahren, die auf Vergleichen basieren.

Untere Schranken für das Sortieren

Satz [6]:

Jeder vergleichende Sortier-Algorithmus benötigt im worst-case $\Omega(n \log n)$ Vergleiche.

Beweis:

Ein binärer Baum der Höhe h hat höchstens 2^h Blätter. Jeder potentielle Entscheidungsbaum für vergleichendes Sortieren von n Elementen hat genau n! Blätter. Folglich gilt für die Höhe jedes Entscheidungsbaumes:

$n! \leq 2^h$ Durch Anwendung des Logarithmus zur Basis 2 erhalten wir:

$$h \geq \log(n!)$$

Aus der Stirling-Formel

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha_n} \quad \text{mit} \quad \frac{1}{12n+1} < \alpha_n < \frac{1}{12n} \quad \text{folgt:}$$

$$\log(n!) = \frac{1}{2}\log(2\pi n) + n\log(\frac{n}{e}) + \log(e)\alpha_n \geq n\log(\frac{n}{e})$$

$$\geq n\log(n) - n\log(e) \geq n\log(n) - \frac{1}{2}n\log(n) = \frac{1}{2}n\log(n)$$

Für alle n mit log(n) > 2 log(e) gilt $h \in \Omega(n\log(n))$ ■

Untere Schranken für das Sortieren

Satz [7]:

Heap-Sort und Merge-Sort sind asymptotisch optimale vergleichende Sortierverfahren.

Beweis:

Die obere Schranke von O(n log n) für die Laufzeit der Algorithmen stimmt überein mit der unteren Schranke von $\Omega(n \log n)$ für vergleichende Sortierverfahren. ■