Lambda-Funktionen

Lambdas sind wie folgt aufgebaut:

$$\underbrace{[value]}_{Capture}\ \underbrace{(int\ x)}_{Parameter}\ \underbrace{->bool}_{return\ type}\ \underbrace{\{return\ x < value;\}}_{Anweisung}$$

- **Caputre**: Variablen, die die Funktion aus dem Kontext des übergeordneten Kontextes nimmt.
  - [x]: Zugriff auf kopierten Wert von x
  - [&x]: Zugriff auf Referenz von x
  - [&]: Referenz auf alle Objekte des Kontextes.
  - [=]: Werte-Zugriff auf alle Objekte im Kontext.
  - [=, &x]: x als Referenz, den Rest als Kopie.
- **Parameter**: Parameter, die der Funktion beim Aufruf übergeben werden. Kann weggelassen werden, wenn die Funktion keine annimmt.
- **Rückgabewert**: spezifiziert welcher Wert zurückgegeben werden soll. Meistens kann er weggelassen werden, da der Compiler anhand der *return* Anweisung selbst darauf schliessen kann.
- **Anweisungs Block**: Wie ein Funktionsrumpf. Hier stehen die Anweisungen, die die Lambda-Funktion ausführen soll und was sie zurückgibt. Man kann auf alle Werte im Capture und auf alle Parameter zugreifen.

Sortieren, mal anders

```
// pre: i >= 0
// post: returns sum of digits of i
int q(int i){
    int res =0;
    for(;i>0;i/=10)
        res += i % 10;
    return res;
}

std::vector<int> v {10,12,9,7,28,22,14};
std::sort (v.begin(), v.end(),
  [] (int i, int j) { return q(i) < q(j);}
);
```

Jetzt $v = 10, 12, 22, 14, 7, 9, 28$ (sortiert nach Quersumme)

Templates

Funktionentemplates

```
template <typename T>
void swap(T& x, T&y){
    T temp = x;
    x = y;
    y = temp;
}
```

Auto

```
#include <iostream>
#include <vector>

int main(){
  std::vector<int> v(10); // Vector of length 10

  for (auto& x: v)
    std::cin >> x;

  for (const auto x: v)
    std::cout << x << " ";
}
```

Typen als Template Parameter

```
template <typename ElementType>
class Vector{
    std::size_t size;
    ElementType* elem;
public:
    ...
    Vector(std::size_t s):
        size{s},
        elem{new ElementType[s]}{}
    ...
    ElementType& operator[](std::size_t pos){
        return elem[pos];
    }
    ...
}
```

Funktoren

Ein simpler Ausgabefilter

```
template <typename T, typename Function>
void filter(const T& collection, Function f){
    for (const auto& x: collection)
        if (f(x)) std::cout << x << " ";
    std::cout << "\n";
}
```

Funktor: Objekt mit überladenem Operator ()

```
class GreaterThan{
  int value; // state
  public:
  GreaterThan(int x):value{x}{}

  bool operator() (int par) const {
    return par > value;
  }
};
```

Ein **Funktor** ist ein aufrufbares Objekt. Kann verstanden werden als Funktion mit Zustand.

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int value=8;
filter(a,GreaterThan(value)); // 9,11,16,19
```

Asymptotik / Komplexität

Asymptotisches Verhalten beschreibt, wie ein Algorithmus sich mit $n \to \infty$ verhält. Die drei Typen sind:

- Obere Grenze

$$\mathcal{O}(g) := \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}, \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$$

  Ab einer gewissen Datengrösse n_0 liegt f unterhalb von g.

- Untere Grenze

$$\Omega(g) := \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}, \forall n \geq n_0 : f(n) \geq c \cdot g(n)\}$$

  Ab einer gewissen Datengrösse n_0 liegt f oberhalb von g.

- Enge Grenze

$$\begin{aligned}\Theta(g) &:= \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N}, \forall n \geq n_0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\} \\ &= \Omega(g) \cap \mathcal{O}(g)\end{aligned}$$

  Ab einer gewissen Datengrösse n_0 liegt f zwischen $\mathcal{O}(g)$ und $\Omega(g)$.

↳ Es gilt:

- Falls $\lim_{x\to\infty} \frac{f}{g} = \infty$, $g \in O(f)$ und $f \in \Omega(g)$;
- Falls $\lim_{x\to\infty} \frac{f}{g} = C \in \mathbb{R}^+ \setminus \{0\}$, $f \in \Theta(g)$ und $g \in \Theta(f)$;
- Falls $\lim_{x\to\infty} \frac{f}{g} = 0$, $f \in O(g)$ und $g \in \Omega(f)$.

- $\sum_{i=0}^{n} i^2 \in \Theta(n^3)$
- $\sum_{i=0}^{n^2} i \in \Theta(n^4)$

$\mathcal{O}(1)$	beschränkt	Array-Zugriff
$\mathcal{O}(\log\log n)$	doppelt logarithmisch	Binäre sortierte Suche interpoliert
$\mathcal{O}(\log n)$	logarithmisch	Binäre sortierte Suche
$\mathcal{O}(\sqrt{n})$	wie die Wurzelfunktion	Primzahltest (naiv)
$\mathcal{O}(n)$	linear	Unsortierte naive Suche
$\mathcal{O}(n \log n)$	superlinear / loglinear	Gute Sortieralgorithmen
$\mathcal{O}(n^2)$	quadratisch	Einfache Sortieralgorithmen
$\mathcal{O}(n^c)$	polynomial	Matrixmultiplikation
$\mathcal{O}(c^n)$	exponentiell	Travelling Salesman Dynamic Programming
$\mathcal{O}(n!)$	faktoriell	Travelling Salesman naiv

$\mathcal{O}(n^?)$

also asymptotische Laufzeit des besten Algorithmus

Komplexität eines Problems P

Minimale (asymptotische) Kosten über alle Algorithmen A, die P lösen.

Komplexität der Elementarmultiplikation zweier Zahlen der Länge n ist $\Omega(n)$ und $\mathcal{O}(n^{\log_3 2})$ (Karatsuba Ofman).

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

$$\sum_{i=0}^{n} i^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6}$$

$$\sum_{i=0}^{n} \rho^i \overset{!}{=} \frac{1-\rho^{n+1}}{1-\rho}$$

Summen und log()-Tricks

$$\log_a y = x \Leftrightarrow a^x = y \quad (a > 0, y > 0)$$

$$\log_a(x \cdot y) = \log_a x + \log_a y \qquad a^x \cdot a^y = a^{x+y}$$

$$\log_a \frac{x}{y} = \log_a x - \log_a y \qquad \frac{a^x}{a^y} = a^{x-y}$$

$$\log_a x^y = y \log_a x \qquad a^{x \cdot y} = (a^x)^y$$

$$\log_a n! = \sum_{i=1}^{n} \log i$$

$$\log_b x = \log_b a \cdot \log_a x \qquad a^{\log_b x} = x^{\log_b a}$$

Begriffsklärung: Idioms und Patterns

Idiom und **(Design) Pattern** sind Begriffe aus dem **Software-Engineering**

- Design Patterns (Entwurfsmuster) sind *allgemeine, in der Regel nicht sprachspezifische*, wiederverwendbare Lösungen für häufig auftretende Entwurfsprobleme. Sie erfassen bewährte Verfahren und beschreiben in der Regel Beziehungen und Interaktionen zwischen Klassen und Objekten. Beispiel: visitor-pattern.
- Idioms sind hingegen *sprachspezifische* Methoden zur Implementierung bestimmter häufig auftretender Aufgaben oder Schritte, z.B. das RAII-Idiom oder das Copy-&-Swap-Idiom in C++. Idioms umfassen üblicherweise auch weniger Code als Patterns.

Rekursionsgleichung

Bsp: Sei

Annahme: $n = 2^k$

$$T(n) = \begin{cases} c & , n = 1 \\ 2T(\frac{n}{2}) + a \cdot n & , n > 1 \end{cases} \longrightarrow \bar{T}(k) := T(2^k) = \begin{cases} c & k = 0 \\ 2\bar{T}(k-1) + a \cdot 2^k & k > 0 \end{cases}$$

$$\begin{aligned}\bar{T}(k) &= 2^k \cdot c + \sum_{i=0}^{k-1} 2^i \cdot a \cdot 2^{k-i} \\ &= c \cdot 2^k + a \cdot k \cdot 2^k \\ &= \Theta(k \cdot 2^k)\end{aligned} \longrightarrow T(n) = \Theta(n \log n)$$

Karatsuba Ofman

Idee: $ab \cdot cd = (10 \cdot a + b) \cdot (10 \cdot c + d)$ (a, b als Ziffern)

$$\begin{aligned}&= 100 \cdot a \cdot c + 10 \cdot a \cdot c \\ &\quad + 10 \cdot \underline{b \cdot d} + \underline{b \cdot d} \\ &\quad + 10 \cdot (a-b) \cdot (d-c)\end{aligned}$$

nur 3 Multiplikationen!

$n^{\log_3 2} \approx n^{1.58}$ Multiplikationen, also $< n^2$

Algo:

Input: Zwei n-stellige ($n > 0$) ganze positive Zahlen x und y mit dezimalen Ziffern $(x_i)_{1 \leq i \leq n}$ und $(y_i)_{1 \leq i \leq n}$
Output: Produkt $x \cdot y$
if $n = 1$ **then**
  return $x_1 \cdot y_1$
else
  Sei $m := \lfloor \frac{n}{2} \rfloor$
  Unterteile $a := (x_1, \ldots, x_m)$, $b := (x_{m+1}, \ldots, x_n)$, $c := (y_1, \ldots, y_m)$, $d := (y_{m+1}, \ldots, y_n)$
  Berechne rekursiv $A := a \cdot c$, $B := b \cdot d$, $C := (a-b) \cdot (d-c)$
  Berechne $R := 10^n \cdot A + 10^m \cdot A + 10^m \cdot B + B + 10^m \cdot C$
  return R

C++

```
for (range-declaration : range-expression)
    statement;
```

```
std::vector<double> v(5);
for (double x: v) std::cout << x; // 00000
for (int x: {1,2,5}) std::cout << x; // 125
for (double& x: v) x=5;
```

copy-swap-idiom

```
  // Assignment operator
  Vector& operator= (const Vector&v){
    Vector cpy(v);
    swap(cpy);
    return *this;
  }
private:
  // helper function
  void swap(Vector& v){
    std::swap(sz, v.sz);
    std::swap(elem, v.elem);
  }
}
```

copy-and-swap idiom: alle Felder von `*this` tauschen mit den Daten von `cpy`. Beim Verlassen von `operator=` wird `cpy` aufgeräumt (dekonstruiert), während die Kopie der Daten von `v` in `*this` verbleiben.

template Bsp.

```
template <typename T> // square number
T sq(T x){
    return x*x;
}
template <typename Container, typename F>
void apply(Container& c, F f){ // x <- f(x) forall x in c
    for(auto& x: c)
        x = f(x);
}
int main(){
  std::vector<int> v={1,2,3};
  apply(v,sq<int>);
  output(v); // 1 4 9
}
```

Maximum Subarray

Gegeben: ein Array von n reellen Zahlen $(a_1, \ldots, a_n)$.
Gesucht: Teilstück $[i, j]$, $1 \leq i \leq j \leq n$ mit maximaler positiver Summe $\sum_{k=i}^{j} a_k$.

Naiver Maximum Subarray Algorithmus

Input: Eine Folge von n Zahlen $(a_1, a_2, \ldots, a_n)$
Output: I, J mit $\sum_{k=I}^{J} a_k$ maximal.

$M \leftarrow 0$; $I \leftarrow 1$; $J \leftarrow 0$
for $i \in \{1, \ldots, n\}$ **do**
  for $j \in \{i, \ldots, n\}$ **do**
    $m = \sum_{k=i}^{j} a_k$
    if $m > M$ **then**
      $M \leftarrow m$; $I \leftarrow i$; $J \leftarrow j$
return I, J

Laufzeit: $\sum_{i=1}^{n} \sum_{j=i}^{n} \overbrace{\sum_{l=i}^{j} 1}^{(j-i+1)} \overbrace{\quad}^{\Theta(n^2)} = \Theta(n^3)$

Maximum Subarray Algorithmus mit Präfixsummen

Input: Eine Folge von n Zahlen $(a_1, a_2, \ldots, a_n)$
Output: I, J mit $\sum_{k=J}^{J} a_k$ maximal.

$\mathcal{S}_0 \leftarrow 0$
for $i \in \{1, \ldots, n\}$ **do** // Präfixsumme
  $\mathcal{S}_i \leftarrow \mathcal{S}_{i-1} + a_i$
$M \leftarrow 0$; $I \leftarrow 1$; $J \leftarrow 0$
for $i \in \{1, \ldots, n\}$ **do**
  for $j \in \{i, \ldots, n\}$ **do**
    $m = \mathcal{S}_j - \mathcal{S}_{i-1}$
    if $m > M$ **then**
      $M \leftarrow m$; $I \leftarrow i$; $J \leftarrow j$

Induktiver Maximum Subarray Algorithmus

Input: Eine Folge von n Zahlen $(a_1, a_2, \ldots, a_n)$.
Output: $\max\{0, \max_{i,j} \sum_{k=i}^{j} a_k\}$.

$M \leftarrow 0$
$R \leftarrow 0$
for $i = 1 \ldots n$ **do**
  $R \leftarrow R + a_i$
  if $R < 0$ **then**
    $R \leftarrow 0$
  if $R > M$ **then**
    $M \leftarrow R$
return M;

Theorem 6

Der induktive Algorithmus für das Maximum Subarray Sum Problem führt $\Theta(n)$ viele Additionen und Vergleiche durch.

Theorem 7

Das Maximum Subarray Sum Problem hat Komplexität $\Theta(n)$.

Beweis: Induktiver Algorithmus mit asymptotischer Laufzeit $\mathcal{O}(n)$.
Jeder Algorithmus hat Laufzeit $\Omega(n)$.
Somit ist die Komplexität $\Omega(n) \cap \mathcal{O}(n) = \Theta(n)$. ■

Suchalgorithmen

5.1.1 Lineare Suche

Die lineare Suche durchläuft das Array vom ersten bis zum letzten Element.

- Bestenfalls: 1 Vergleich
- Schlimmstenfalls: n Vergleiche
- Erwartungswert bei Gleichverteilung: $\frac{n+1}{2}$

Theorem 10

Jeder vergleichsbasierte Algorithmus zur Suche in unsortierten Daten der Länge n benötigt im schlechtesten Fall $\Omega(n)$ Vergleichsschritte.

5.1.2 Binäre Suche

Voraussetzung für die Binäre Suche ist ein sortiertes Array. Der Algorithmus beginnt in der Mitte, vergleicht ob das Element grösser oder kleiner als der gesuchte Key ist und wiederholt dieses Vorgehen in der linken bzw. rechten Hälfte. Dieses Verfahen nennt sich Divide & Conquer. Besser als ein Vektor eignet sich hier ein Suchbaum oder AVL-Baum, zu denen wir später kommen werden.
Die Laufzeit beträgt $\Theta(\log n)$

Binärer Suchalgorithmus BSearch(A, l, r, b)

Input: Sortiertes Array A von n Schlüsseln. Schlüssel b. Bereichsgrenzen $1 \leq l, r \leq n$ mit $l \leq r$ oder $l = r + 1$.
Output: Index $m \in [l, \ldots, r+1]$, so dass $A[i] \leq b$ für alle $l \leq i < m$ und $A[i] \geq b$ für alle $m < i \leq r$.

$m \leftarrow \lfloor (l + r)/2 \rfloor$
if $l > r$ **then** // erfolglose Suche
  return l
else if $b = A[m]$ **then**// gefunden
  return m
else if $b < A[m]$ **then**// Element liegt links
  return BSearch$(A, l, m-1, b)$
else // $b > A[m]$: Element liegt rechts
  return BSearch$(A, m+1, r, b)$

Analyse (schlechtester Fall)

Rekurrenz $(n = 2^k)$

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

$$T(n) = T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c = \ldots$$
$$= T\left(\frac{n}{2^i}\right) + i \cdot c$$
$$= T\left(\frac{n}{n}\right) + \log_2 n \cdot c = d + c \cdot \log_2 n \in \Theta(\log n)$$

Iterativer binärer Suchalgorithmus

Input: Sortiertes Array A von n Schlüsseln. Schlüssel b.
Output: Index des gefundenen Elements. 0, wenn erfolglos.
$l \leftarrow 1$; $r \leftarrow n$
while $l \leq r$ **do**
  $m \leftarrow \lfloor (l + r)/2 \rfloor$
  if $A[m] = b$ **then**
    return m
  else if $A[m] < b$ **then**
    $l \leftarrow m + 1$
  else
    $r \leftarrow m - 1$
return *NotFound*;

Algorithmus bricht nur ab, falls $A[l..r]$ leer oder b gefunden.
Invariante: Falls b in A, dann im Bereich $A[l..r]$
Beweis durch Induktion
- Induktionsanfang: $b \in A[1..n]$ (oder nicht)
- Hypothese: Invariante gilt nach i Schritten
- Schritt: $b < A[m] \Rightarrow b \in A[l..m-1]$, $b > A[m] \Rightarrow b \in A[m+1..r]$

Auswahlalgorithmen

Eingabe

- Unsortiertes Array $A = (A_1, \ldots, A_n)$ paarweise verschiedener Werte
- Zahl $1 \leq k \leq n$.

Ausgabe: $A[i]$ mit $|\{j : A[j] < A[i]\}| = k - 1$

Spezialfälle

$k = 1$: Minimum: Algorithmus mit n Vergleichsoperationen trivial.
$k = n$: Maximum: Algorithmus mit n Vergleichsoperationen trivial.
$k = \lfloor n/2 \rfloor$: Median.

Algorithmus `Partition`(A, l, r, p)

Input: Array A, welches den Pivot p in $A[l, \ldots, r]$ mindestens einmal enthält.
Output: Array A partitioniert in $A[l, \ldots, r]$ um p. Rückgabe der Position von p.
while $l \leq r$ **do**
  while $A[l] < p$ **do**
    $l \leftarrow l + 1$
  while $A[r] > p$ **do**
    $r \leftarrow r - 1$
  swap$(A[l], A[r])$
  if $A[l] = A[r]$ **then**
    $l \leftarrow l + 1$
return l-1

Algorithmus `Quickselect` (A, l, r, k)

Input: Array A der Länge n. Indizes $1 \leq l \leq k \leq r \leq n$, so dass für alle $x \in A[l..r] : |\{j | A[j] \leq x\}| \geq l$ und $|\{j | A[j] \leq x\}| \leq r$.
Output: Wert $x \in A[l..r]$ mit $|\{j | A[j] \leq x\}| \geq k$ und $|\{j | x \leq A[j]\}| \geq n - k + 1$
if l=r **then**
  return $A[l]$;
$x \leftarrow$ `RandomPivot`(A, l, r)
$m \leftarrow$ `Partition`(A, l, r, x)
if $k < m$ **then**
  return `QuickSelect`$(A, l, m-1, k)$
else if $k > m$ **then**
  return `QuickSelect`$(A, m+1, r, k)$
else
  return $A[k]$

Überblick

1.	Wiederholt Minimum finden	$\mathcal{O}(n^2)$
2.	Sortieren und $A[i]$ ausgeben	$\mathcal{O}(n \log n)$
3.	Quickselect mit zufälligem Pivot	$\mathcal{O}(n)$ im Mittel
4.	Median of Medians (Blum)	$\mathcal{O}(n)$ im schlimmsten Fall

Rekursive Problemlösestrategien

Brute Force Enumeration	Backtracking	Divide and Conquer	Dynamic Programming	Greedy
Rekursive Aufzählbarkeit	Prüfbare Randbedingung, Partielle Validierung	Optimale Substruktur	Optimale Substruktur, Überlappende Teilprobleme	Optimale Substruktur, Gierige Auswahl Eigenschaft
DFS, BFS, Alle Permutationen, Baumtraversieren	n Damen, Sudoku, m-Färbung, SAT-Solving, naiver TSP	Binäre Suche, Mergesort, Quicksort, Türme von Hanoi, FFT	Bellman Ford, Warshall, Rod-Cutting, LAT, Editierdistanz, Knapsack Problem DP	Dijkstra, Kruskal, Huffmann Coding

Geometrische Algorithmen

· Konvexe Hülle:

Jarvis Marsch / Gift Wrapping Algorithmus

1. Starte mit Extrempunkt (z.B. unterster Punkt) $p = p_0$
2. Suche Punkt q, so dass $\overline{pq}$ am weitesten rechts liegende Gerade, d.h. jeder andere Punkt liegt links von der Geraden $\overline{pq}$ (oder auf der Geraden näher bei p).
3. Fahre mit $p \leftarrow q$ bei (2) weiter, bis $p = p_0$.

- Sei h die Anzahl Eckpunkte der konvexen Hülle.
- Laufzeit des Algorithmus $\mathcal{O}(h \cdot n)$.

Algorithmus Graham-Scan

Input: Menge von Punkten Q
Output: Stack S von Punkten der konvexen Hülle von Q
p_0: Punkt mit minimaler y- (gegebenenfalls zusätzlich minimaler x-) Koordinate
$(p_1, \ldots, p_m)$ restlichen Punkte sortiert nach Polarwinkel gegen Uhrzeigersinn relativ zu p_0; Wenn Punkte mit gleichem Polarwinkel vorhanden, verwerfe alle ausser dem mit maximalen Abstand von p_0
$S \leftarrow \emptyset$
if $m < 2$ **then return** S
Push(S, p_0); Push(S, p_1); Push(S, p_2)
for $i \leftarrow 3$ **to** m **do**
  while Winkel (NextToTop(S), Top(S), p_i) nicht nach links gerichtet **do**
    Pop(S);
  Push(S, p_i)
return S

Laufzeit des Algorithmus Graham-Scan
- Sortieren $\mathcal{O}(n \log n)$
- n Iterationen der For-Schleife
- Amortisierte Analyse des Multipop beim Stapel: amortisiert konstante Laufzeit des Multipop, ebenso hier: amortisiert konstante While-Schleife.

Insgesamt $\mathcal{O}(n \log n)$

Algorithmus Any-Segments-Intersect(S)

Input: Liste von n Strecken S
Output: Rückgabe ob S schneidende Strecken enthält
$T \leftarrow \emptyset$
Sortiere Endpunkte der Strecken in S von links nach rechts (links vor rechts und unten vor oben)
for Sortierte Endpunkte p **do**
  if p linker Endpunkt einer Strecke s **then**
    Insert(T, s)
    if Above$(T, s) \cap s \neq \emptyset \ \vee$ Below$(T, s) \cap s \neq \emptyset$ **then return** true
  if p rechter Endpunkt einer Strecke s **then**
    if Above$(T, s) \cap$ Below$(T, s) \neq \emptyset$ **then return** true
    Delete(T, s)
return false;

Laufzeit des Algorithmus Any-Segments-Intersect
- Sortieren $\mathcal{O}(n \log n)$
- $2n$ Iterationen der For-Schleife. Jede Operation auf dem balancierten Baum $\mathcal{O}(\log n)$

Insgesamt $\mathcal{O}(n \log n)$

Sortieralgorithmen

5.2.1 Bubblesort

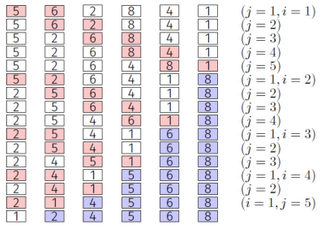
Bubblesort geht das Array (bis zu) n Mal von links nach rechts durch und betrachtet immer zwei benachbarte Elemente. Sind diese nicht sortiert, werden sie vertauscht. Wird in einem Durchlauf nichts vertauscht, ist das Array sortiert. Dies resultiert in

- $\Theta(n^2)$ Vergleiche
- $\Theta(n)$ Vertauschungen

Das Worst-case Szenario ist, das Ausgangsarray absteigend sortiert zu haben.

Algorithmus: `Bubblesort`

Input: Array $A = (A[1], \dots, A[n])$, $n \geq 0$.
Output: Sortiertes Array A
for $i \leftarrow 1$ **to** $n-1$ **do**
  for $j \leftarrow 1$ **to** $n-i$ **do**
    if $A[j] > A[j+1]$ **then**
      swap$(A[j], A[j+1])$;



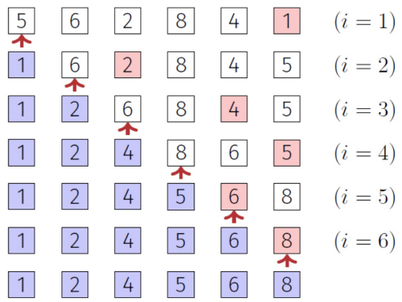
5.2.2 Selectionsort

Das kleinste Element wird an die erste Stelle des Arrays getauscht. Das zweit kleinste kommt an die zweite Stelle usw. Bei der Implementation wird der Pointer für den linken Rand des Arrays in jedem Schritt ein nach rechts verschoben.

- $\Theta(n^2)$ Vergleiche
- $\Theta(n)$ Vertauschungen

Algorithm: Selection Sort

Input: Array $A = (A[1], \dots, A[n])$, $n \geq 0$.
Output: Sorted Array A
for $i \leftarrow 1$ **to** $n-1$ **do**
  $p \leftarrow i$
  for $j \leftarrow i+1$ **to** n **do**
    if $A[j] < A[p]$ **then**
      $p \leftarrow j$;
  swap$(A[i], A[p])$



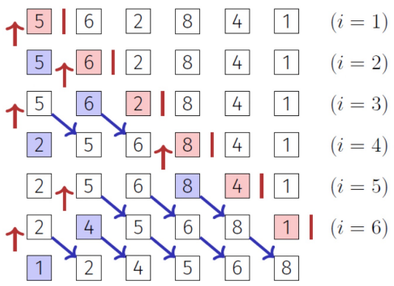
5.2.3 Insertionsort

Insertionsort funktioniert so, wie wir Spielkarten in unserer Hand sortieren. Die schelife geht von $i = 1$ bis n. Man wählt das i-te Element des Arrays arr und fügt es mittels binärer Suche in $arr[0..i-1]$ ein.
Wie bei Selectionsort hat man während dem Algorithmus einen sortierten und einen unsortierten Teil des Arrays.

- $\Theta(n \log n)$ Vergleiche
- $\Theta(n^2)$ Vertauschungen

Algorithm: Insertion Sort

Input: Array $A = (A[1], \dots, A[n])$, $n \geq 0$.
Output: Sorted Array A
for $i \leftarrow 2$ **to** n **do**
  $x \leftarrow A[i]$
  $p \leftarrow$ `BinarySearch`$(A, 1, i-1, x)$;
  for $j \leftarrow i-1$ **downto** p **do**
    $A[j+1] \leftarrow A[j]$
  $A[p] \leftarrow x$



5.2.4 Shellsort

Shellsort arbeitet mit Insertion sort auf Teilfolgen der Eingabe. Abängig von der Eingabegrösse wählt man ein k und betrachtet in jedem Durchgang nur jedes k-te Element. Dies bietet den Vorteil, dass Elemente nicht nur einen Schritt sondern gleich mehrere nach vorne oder hinten wandern.
Z.B. kann man die Folge $1, 3, 7, 15, 2^k - 1$ wählen. Man benutzt Insertion sort zuerst mit jedem $2^k - 1$-ten Element, mit jedem $2^{k-1} - 1$-ten, ..., 15., 7., 3. und schliesslich jedem ELement. Bei dieser Folge erreicht man dann eine Lauftzeit von $\mathcal{O}(n^{3/2})$.

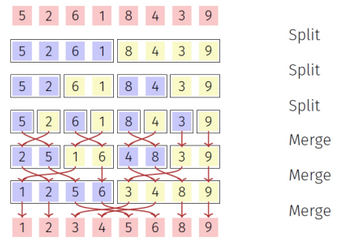
9	8	7	6	5	4	3	2	1	0	
2	8	7	6	5	4	3	9	1	0	insertion sort, $k = 7$
2	1	7	6	5	4	3	9	8	0	
2	1	0	6	5	4	3	9	8	7	
2	1	0	3	5	4	6	9	8	7	insertion sort, $k = 3$
2	1	0	3	5	4	6	9	8	7	
2	1	0	3	5	4	6	9	8	7	
0	1	2	3	4	5	6	7	8	9	insertion sort, $k = 1$

5.2.6 Mergesort

Hier wird erneut Rekursion angewandt. Der Input wird in zwei teile aufgeteilt, mit welchen Merge sort erneut aufgerufen wird. Nachdem zwei Teile sortiert zurückkommen, werden sie (linear in Zeit) aufsteigend zusammengefügt.
Der Algorithmus benötigt $\Theta(n \log n)$ Vergleiche und Vertauschungen.

Algorithmus `Merge`(A, l, m, r)

Input: Array A der Länge n, Indizes $1 \leq l \leq m \leq r \leq n$. $A[l, \dots, m]$, $A[m+1, \dots, r]$ sortiert
Output: $A[l, \dots, r]$ sortiert
$B \leftarrow$ new Array$(r - l + 1)$
$i \leftarrow l$; $j \leftarrow m+1$; $k \leftarrow 1$
while $i \leq m$ and $j \leq r$ **do**
  if $A[i] \leq A[j]$ **then** $B[k] \leftarrow A[i]$; $i \leftarrow i+1$
  else $B[k] \leftarrow A[j]$; $j \leftarrow j+1$
  $k \leftarrow k+1$;
while $i \leq m$ **do** $B[k] \leftarrow A[i]$; $i \leftarrow i+1$; $k \leftarrow k+1$
while $j \leq r$ **do** $B[k] \leftarrow A[j]$; $j \leftarrow j+1$; $k \leftarrow k+1$
for $k \leftarrow l$ **to** r **do** $A[k] \leftarrow B[k-l+1]$



Algorithmus (Rekursives 2-Wege) `Mergesort`(A, l, r)

Input: Array A der Länge n. $1 \leq l \leq r \leq n$
Output: $A[l, \dots, r]$ sortiert.
if $l < r$ **then**
  $m \leftarrow \lfloor (l+r)/2 \rfloor$ // Mittlere Position
  `Mergesort`(A, l, m) // Sortiere vordere Hälfte
  `Mergesort`$(A, m+1, r)$ // Sortiere hintere Hälfte
  `Merge`(A, l, m, r) // Verschmelzen der Teilfolgen

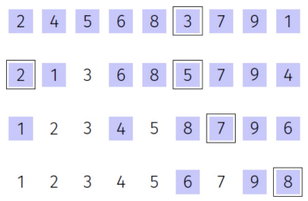
Was ist der Nachteil von Mergesort?
Benötigt zusätzlich $\Theta(n)$ Speicherplatz für das Verschmelzen.
Wie könnte man das Verschmelzen einsparen?
Sorge dafür, dass jedes Element im linken Teil kleiner ist als im rechten Teil.
Wie?
Pivotieren und Aufteilen!

5.2.5 Quicksort

Dieser Algorithmus arbeitet Rekursiv. Es wird ein Pivot Element gewählt und anhand dessen zwei Teilarrays S^- und S^+ erstellt, in denen alle kleineren bzw. grösseren Elemente sind. Die Funktion ruft sich dann erneut auf diesen Arrays auf und fügt sie schliesslich mit dem Pivot in der Mitte zusammen.
Die Laufzeit beträgt im schlechtesten fall $\Theta(n^2)$, was aber sehr selten vorkommt.

Algorithmus `Partition`(A, l, r, p)

Input: Array A, welches den Pivot p in $A[l, \dots, r]$ mindestens einmal enthält.
Output: Array A partitioniert in $A[l, \dots, r]$ um p. Rückgabe der Position von p.
while $l \leq r$ **do**
  while $A[l] < p$ **do**
    $l \leftarrow l+1$
  while $A[r] > p$ **do**
    $r \leftarrow r-1$
  swap$(A[l], A[r])$
  if $A[l] = A[r]$ **then**
    $l \leftarrow l+1$
return l-1



Algorithmus `Quicksort`(A, l, r)

Input: Array A der Länge n. $1 \leq l \leq r \leq n$.
Output: Array A, sortiert in $A[l, \dots, r]$.
if $l < r$ **then**
  Wähle Pivot $p \in A[l, \dots, r]$
  $k \leftarrow$ `Partition`(A, l, r, p)
  `Quicksort`$(A, l, k-1)$
  `Quicksort`$(A, k+1, r)$

Quicksort wird trotz $\Theta(n^2)$ Laufzeit im schlechtesten Fall oft eingesetzt.
Grund: Quadratische Laufzeit unwahrscheinlich, sofern die Wahl des Pivots und die Vorsortierung nicht eine ungünstige Konstellation aufweisen.
Vermeidung: Zufälliges Ziehen eines Pivots. Mit gleicher Wahrscheinlichkeit aus $[l, r]$.

Theorem 13
Im Mittel benötigt randomisiertes Quicksort $\mathcal{O}(n \log n)$ Vergleiche.

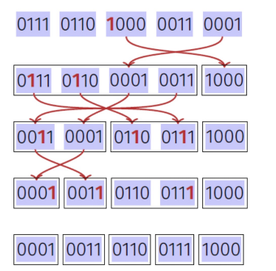
Theorem 15
Vergleichsbasierte Sortierverfahren benötigen im schlechtesten Fall und im Mittel mindestens $\Omega(n \log n)$ Schlüsselvergleiche.

5.2.8 Radixsort

Dieser Algorithmus sortiert die Eingabe Ziffer für Ziffer, beginnend bei der signifikantesten. Im schlechtesten Fall erreicht Radixsort $\mathcal{O}(p \cdot n)$, wobei p die Anzahl der Ziffern ist.

Algorithmus RadixExchangeSort(A, l, r, b)

Input: Array A der Länge n, linke und rechte Grenze $1 \leq l \leq r \leq n$, Bitposition b
Output: Array A, im Bereich $[l, r]$ nach Bits $[0, \dots, b]$ sortiert.
if $l < r$ **and** $b \geq 0$ **then**
  $i \leftarrow l-1$
  $j \leftarrow r+1$
  repeat
    repeat $i \leftarrow i+1$ **until** $z_2(b, A[i]) = 1$ **or** $i \geq j$
    repeat $j \leftarrow j-1$ **until** $z_2(b, A[j]) = 0$ **or** $i \geq j$
    if $i < j$ **then** swap$(A[i], A[j])$
  until $i \geq j$
  RadixExchangeSort$(A, l, i-1, b-1)$
  RadixExchangeSort$(A, i, r, b-1)$



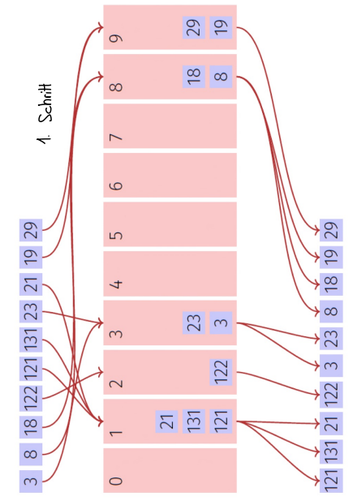
5.2.9 Bucketsort

Hier werden wiederholt Buckets erstellt und deren Inhalte in der richtigen Reihenfolge zusammengefügt. Der erste Bucket sortiert die Eingabe anhand der letzten Ziffer. Nach dem zusammenfügen wird werden die Zahlen anhand der zweitletzten Ziffer sortiert usw. Die Laufzeit beträgt $\mathcal{O}(n)$.
Damit sind Radix sort und Bucket sort die einzigen hier behandelten Sortieralgorithmen, die in linearer Zeit arbeiten. Die anderen haben alle mindesten $\Omega(n \log n)$.

Bucket Sort – Andere Voraussetzung

Annahme: gleichmässig verteilte Daten, z.B. aus $[0, 1)$
Input: Array A der Länge n, $A_i \in [0, 1)$, Konstante $M \in \mathbb{N}^+$
Output: Sortiertes Array
$k \leftarrow \lceil n/M \rceil$
$B \leftarrow$ new array of k empty lists
for $i \leftarrow 1$ **to** n **do**
  $B[\lfloor A_i \cdot k \rfloor]$.append$(A[i])$
for $i \leftarrow 1$ **to** k **do**
  sort $B[i]$ // z.B. insertion sort, mit Laufzeit $\mathcal{O}(M^2)$
return $B[0] \circ B[1] \circ \cdots \circ B[k]$ // konkateniert

Erwartete asymptotische Laufzeit $\mathcal{O}(n)$ (Beweis in Cormen et al, Kap. 8.4)



5.2.7 Heapsort

Alle Elemente werden in einen Max-Heap eingefügt (dazu mehr in 6.4.4). Nun vertauscht man die Wurzel mit dem letzten Knoten (1. und letztes Element im Array), löscht die ehemalige Wurzel und stellt erneut die Max-Heap Eigenschaften her. Dieser Vorgang wird wiederholt, bis der ganze Heap aufgelöst wurde. Bewegt man dabei immer die Elemente im Array, wird dieses sortiert.

- Einfügen: $\mathcal{O}(\log n)$
- Löschen: $\mathcal{O}(\log n)$
- Sortieren: $\mathcal{O}(n \log n)$

Algorithmus HeapSort(A, n)

Input: Array A der Länge n.
Output: A sortiert.
// Heap aufbauen
for $i \leftarrow n/2$ **downto** 1 **do**
  Versickere(A, i, n)
// Nun ist A ein Heap
for $i \leftarrow n$ **downto** 2 **do**
  Vertausche$(A[1], A[i])$
  Versickere$(A, 1, i-1)$
// Nun ist A sortiert.

Algorithmus Versickern(A, i, m)

Input: Array A mit Heapstruktur für die Kinder von i. Letztes Element m.
Output: Array A mit Heapstruktur für i mit letztem Element m.
while $2i \leq m$ **do**
  $j \leftarrow 2i$; // j linkes Kind
  if $j < m$ and $A[j] < A[j+1]$ **then**
    $j \leftarrow j+1$; // j rechtes Kind mit grösserem Schlüssel
  if $A[i] < A[j]$ **then**
    Vertausche$(A[i], A[j])$
    $i \leftarrow j$; // weiter versickern
  else
    $i \leftarrow m$; // versickern beendet

Heapsort: $\mathcal{O}(n \log n)$ Vergleiche und Bewegungen.

Nachteile von Heapsort?

- Wenig Lokalität: per Definition springt Heapsort im sortierten Array umher (Negativer Cache-Effekt).
- Zwei Vergleiche vor jeder benötigten Bewegung.

stable and in-situ

A comparison-based sort algorithm is called **stable**, if the relative order of two identical elements is not changed by the algorithm. A sort algorithm is called **in-situ**, if it does, apart from the input sequence, only require a constant amount of memory.

insertionSort and *bubbleSort* are stable in their naive implementation. *mergeSort* is stable if at every merge the first element is preferred. *selectionSort*, *quickSort* and *heapSort* cannot be made stable easily.
selectionSort, *insertionSort*, *bubbleSort* and *heapSort* work directly on the input array and are therefore **in-situ**. *quickSort* uses between $\Omega(\log n)$ and $\mathcal{O}(n)$ extra space to keep track of the recursive calls. This space is not used for element storage, therefore *quickSort* is **in-situ**. *mergeSort* has to merge repeatedly parts of the array. There are complicated modifications to make *mergeSort* in-situ, but none that can be achieved by simple modifications of the standard algorithm.

Stability of a sorting algorithm only refers to the elements with the same value. Attribute each element with its original position and sort by value plus position for elements with equal values. Maximally one more comparison per element (factor of 2).

6.1 Stack

Beim Stack gilt Last in, First out. Ein Stack unterstützt die Operationen:

- **push(x,S)**: Legt Element x auf Stapel S
- **pop(S)**: Entfernt oberstes Element von S und gibt es zurück (oder *nullptr*)
- **top(S)**: Gibt oberstes Element (oder *nullptr*) zurück, ohne es zu entfernen.
- **isEmpty(S)**: *true* wenn Stack leer ist, sonst *false*.
- **emptyStack()**: Gibt leeren Stack zurück.

} $\mathcal{O}(1)$

6.2 Queue

Bei einer Queue gilt First in, First out. Eine Queue unterstützt die Operationen:

- **enqueue(x,Q)**: fügt x am Ende der Schlange an.
- **dequeue(Q)**: Entfernt das vorderste Element der Schlange und gibt es zurück.
- **head(Q)**: Gibt das Element am vorderen Ende der Schlange zurück, ohne es zu entfernen.
- **isEmpty(Q)**: *true* wenn Queue leer ist, sonst *false*.
- **emptyQueue()**: Gibt leere Queue zurück.

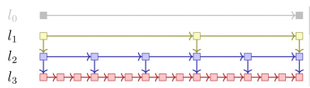
} $\mathcal{O}(1)$

	enqueue	delete	search	concat
(A)	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
(B)	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
(C)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
(D)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$

(A) = Einfach verkettet
(B) = Einfach verkettet, mit Dummyelement am Anfang und Ende
(C) = Einfach verkettet, mit einfach indirekter Elementaddressierung
(D) = Doppelt verkettet

6.3.1 Skiplisten

Sortierte verkettete Liste mit mehreren Ebenen. Bei der Suche startet man in der obersten Ebene mit den wenigsten Einträgen. Hat das nächste Element einen kleineren Wert, geht man weiter. Wenn nicht, geht man eine Ebene tiefer. So bewegt man sich schneller durch die Liste, was sich in einer erwarteten Laufzeit von $\mathcal{O}(\log n)$ für Suchen, Löschen und meist auch Einfügen bemerkbar macht.
Meist wird per Zufall entschieden, wie viele Ebenen ein Wert bekommt. Dabei liegt die Wahrscheinlichkeit für 2 bei $\frac{1}{2}$, für 3 bei $\frac{1}{4}$, ..., für k bei $\frac{1}{2^{k-1}}$.



Theorem 16

Die Anzahl an erwarteten Elementaroperationen für Suchen, Einfügen und Löschen eines Elements in einer randomisierten Skipliste ist $\mathcal{O}(\log n)$.

6.3.2 Hashing

- **Lineares Sondieren:**

$$s(k,j) = h(k) + j$$
$$S(k) = \big(h(k), h(k)+1, ..., h(k)+m-1\big) \mod m$$

Bei $\alpha = 0.95$ betrachtet die erfolglose Suche hier im Durchschnitt 200 Einträge.

- **Quadratisches Sondieren:**

$$s(k,j) = h(k) + \lceil j/2 \rceil (-1)^{j+1}$$
$$S(k) = \big(h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, ...\big) \mod m$$

Bei $\alpha = 0.95$ betrachtet die erfolglose Suche hier im Durchschnitt 22 Einträge.

- **Double Hashing** benutzt zwei Hashfunktionen, $h(k)$ und $h'(k)$.

$$s(k,j) = h(k) + j \cdot h'(k)$$
$$S(k) = \big(h(k), h(k)+h'(k), h(k)+2h'(k), ..., h(k)+(m-1)h'(k)\big) \mod m$$

Bei $\alpha = 0.95$ betrachtet die erfolglose Suche hier im Durchschnitt 20 Einträge (Formel im Vorlesungsskript S.395).

Beispiele für die beiden Hashfunktionen sind $h(k) = k \mod m$ und $h'(k) = 1 + k \mod m$.

Steht in der Prüfung zum Beispiel "Lineare Sondierung nach links", muss das + durch ein − ausgetauscht werden.

Divisionsmethode

$$h(k) = k \bmod m$$

Ideal: m Primzahl, nicht zu nahe bei Potenzen von 2 oder 10
Aber oft: $m = 2^r - 1$ ($r \in \mathbb{N}$), da Tabellenwachstum per Verdoppelung

Anzahl Knoten und Höhe von Bäumen

Die Höhe $h(T)$ eines binären Baumes T mit Wurzel r ist gegeben als

$$h(r) = \begin{cases} 0 & \text{falls } r = \textbf{null} \\ 1 + \max\{h(r.\text{left}), h(r.\text{right})\} & \text{sonst.} \end{cases}$$

Baum mit 1 Knoten hat Höhe 1

binärer Baum der Höhe h

$h \le n \le 2^h - 1$

AVL-Baum der Höhe h

$N(0) = 0, \quad N(1) = 1$
$N(h-1) + N(h-2) + 1 \le n \le 2^h - 1$

Wege / einfache Beobachtungen

- **Weg**: Sequenz von Knoten $\langle v_1, \dots, v_{k+1} \rangle$ so dass für jedes $i \in \{1 \dots k\}$ eine Kante von v_i nach v_{i+1} existiert.
- **Länge** des Weges: Anzahl enthaltene Kanten k.
- **Pfad** (auch: einfacher Pfad): Weg der keinen Knoten mehrfach verwendet.

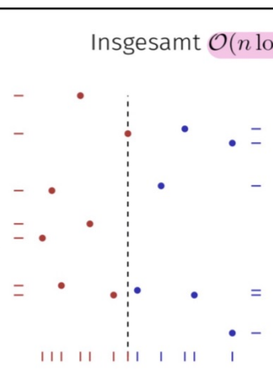
- Allgemein: $0 \le |E| \in \mathcal{O}(|V|^2)$
- Zusammenhängender Graph: $|E| \in \Omega(|V|)$
- Vollständiger Graph: $|E| = \frac{|V| \cdot (|V|-1)}{2}$ (ungerichtet)
- Maximal $|E| = |V|^2$ (gerichtet),$|E| = \frac{|V| \cdot (|V|+1)}{2}$ (ungerichtet)

Dichtestes Punktepaar

Insgesamt $\mathcal{O}(n \log n)$

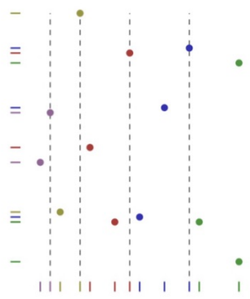
Divide

- Punktmenge P, zu Beginn $P \leftarrow Q$
- Arrays X und Y, welche die Punkte aus P enthalten, sortiert nach x- bzw. nach y-Koordinate.
- Teile Punktmenge ein in zwei (annähernd) gleich grosse Mengen P_L und P_R, getrennt durch vertikale Gerade durch einen Punkt von P.
- Teile Arrays X und Y entsprechend in X_L, X_R. Y_L und Y_R.



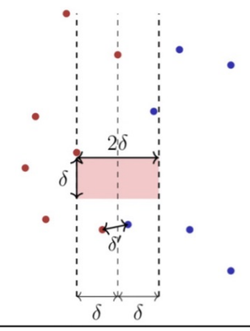
Conquer

- Rekursiver Aufruf jeweils mit P_L, X_L, Y_L und P_R, X_R, Y_R. Erhalte minimale Abstände δ_L, δ_R.
- (Wenn nur noch $k \le 3$ Punkte: berechne direkt minimalen Abstand)
- Nach reukrsivem Aufruf $\delta = \min(\delta_L, \delta_R)$. Kombiniere (nächste Folie) und gib bestes Resultat zurück.

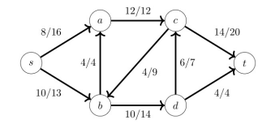


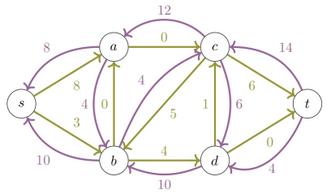
Kombiniere

- Erzeuge Array Y' mit y-sortierten Punkten aus Y, die innerhalb des 2δ Streifens um die Trennlinie befinden
- Betrachte für jeden Punkt $p \in Y'$ die maximal sieben auf p folgenden Punkte mit y-Koordinaten-Abstand kleiner δ. Berechne minimale Distanz δ'.
- Wenn $\delta' < \delta$, dann noch dichteres Paar in P als in P_L und P_R gefunden. Rückgabe der minimalen Distanz.

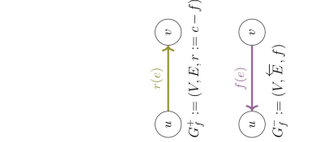


Flüsse





Restnetzwerk: $G_f := G_f^+ \cup G_f^- = (V, E_f, c_f)$



Idee: Erhöhung und Verringerung

- **Erhöhung:** Fluss durch e kann um höchstens $r(e) := c(e) - f(e)$ erhöht werden
- **Verringerung:** Fluss durch e kann um höchstens $f(e)$ verringert werden
  $\Rightarrow$ Fluss durch $\overleftarrow{e}$ kann um höchstens $f(e)$ erhöht werden

Algorithmus Ford-Fulkerson(G, s, t)

Input: Flussnetzwerk $G = (V, E, c)$, Quelle s, Senke t
Output: Maximaler Fluss f

for $e \in E$ **do**
  $f(e) \leftarrow 0$
while existiert positiver Weg P: $s \rightsquigarrow t$ im Restnetzwerk $G_f = (V, E_f, c_f)$ **do**
  $d \leftarrow \min_{e \in P} c_f(e)$
  foreach $e \in P$ **do**
    if $e \in E$ **then**
      $f(e) \leftarrow f(e) + d$
    else
      $f(\overleftarrow{e}) \leftarrow f(\overleftarrow{e}) - d$

Laufzeit-Analyse von Ford-Fulkerson

Zeit pro Iteration: Suche eines Erweiterungswegs $s \rightsquigarrow t$
$\Rightarrow$ Tiefensuche oder Breitensuche: $\mathcal{O}(|V| + |E|) = \mathcal{O}(|E|)$
($|V| \le |E|$, da man alle nicht erreichbaren Knoten ignorieren kann.)

Anzahl Iterationen:
In jedem Schritt erhöht sich der Grösse des Flusses $|f|$ um $d > 0$.
ganzzahlige Kapazitäten $\Rightarrow$ Erhöhung um ≥ 1 $\Rightarrow$ höchstens $|f_{\max}|$ Iterationen

$\Rightarrow \mathcal{O}(|f_{\max}| \cdot |E|)$ für Flussnetzwerke $G = (V, E, c)$ mit $c\colon E \to \mathbb{N}^{\ge 1}$

Edmonds-Karp Algorithmus: (Variante von Ford-Fulkerson)
kürzester Erweiterungsweg (Anzahl Kanten) $\Rightarrow \mathcal{O}(|V| \cdot |E|^2)$ (ohne Herleitung)

Max-Flow Min-Cut Theorem

Für einen Fluss f in einem Flussnetzwerk $G = (V, E, c)$ mit Quelle s und Senke t sind die folgende Aussagen äquivalent:

1. f ist ein maximaler Fluss in G
2. Das Restnetzwerk G_f enthält keine Erweiterungswege
3. $|f| = c(S,T)$ für einen Schnitt (S,T) von G. $\longrightarrow$ Max Flow = Min Cut

- $V = S \sqcup T$
- $c(S,T) = \sum_{\substack{s \in S \\ t \in T \\ (s,t) \in E}} c(s,t)$

6.4 Bäume

Ein Binärer Baum besteht aus Knoten mit einem Wert/Schlüssel und zwei Zeigern; je einer zum linken und zum rechten Teilbaum. Die meisten Operationen verlaufen in $\mathcal{O}(\log n)$.

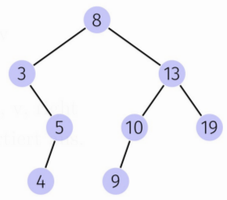
6.4.1 Binärer Suchbaum

Die Suchbaumeigenschaften sind:

- Jeder Knoten **v** speichert einen Schlüssel
- Schlüssel im linken Teilbaum **v.left** sind kleiner als **v.key**
- Schlüssel im rechten Teilbaum **v.right** sind grösser als **v.key**

6.4.2 Traversierungsarten

- *Hauptreihenfolge* (*preorder*):
  v, dann $T_{\text{left}}(v)$, dann $T_{\text{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19
- *Nebenreihenfolge* (*postorder*):
  $T_{\text{left}}(v)$, dann $T_{\text{right}}(v)$, dann v.
  4, 5, 3, 9, 10, 19, 13, 8
- *Symmetrische Reihenfolge* (*inorder*):
  $T_{\text{left}}(v)$, dann v, dann $T_{\text{right}}(v)$.
  3, 4, 5, 8, 9, 10, 13, 19

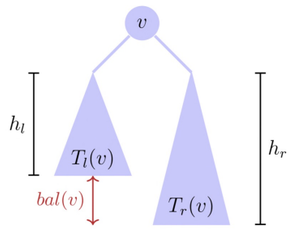


6.4.3 AVL Bäume

Dies sind spezielle Suchbäume, bei denen Degeneration durch Balancieren verhindert wird.

Die *Balance* eines Knotens v ist definiert als die Höhendifferenz seiner beiden Teilbäume $T_l(v)$ und $T_r(v)$

$$\text{bal}(v) := h(T_r(v)) - h(T_l(v))$$



AVL-Bedingung: für jeden Knoten v eines Baumes gilt $\text{bal}(v) \in \{-1, 0, 1\}$

Rotationen

RR **Rechts Rotation:** $bal(pp) = 2,\ bal(p) = 1$

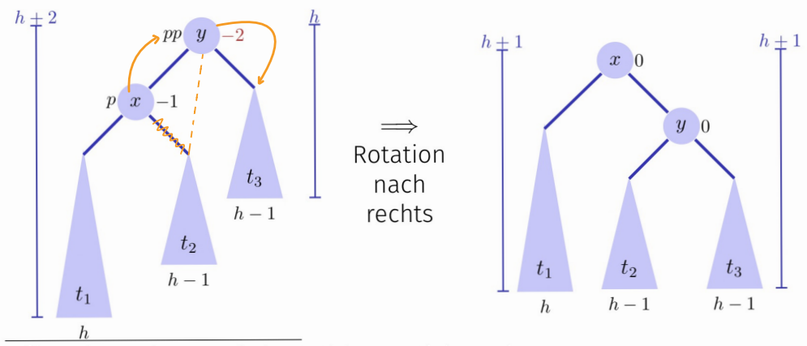
LL **Links Rotation:** $bal(pp) = -2,\ bal(p) = -1$

LR **Links-Rechts Rotation:** $bal(pp) = -2,\ bal(p) = 1$

RL **Rechts-Links Rotation:** $bal(pp) = 2,\ bal(p) = -1$

Rotationen

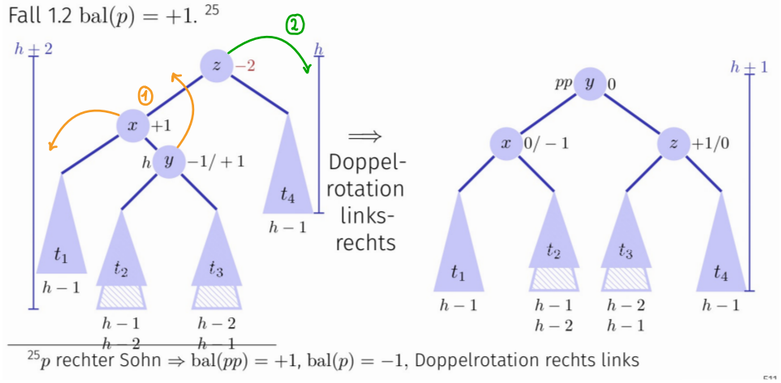
Fall 1.1 $\text{bal}(p) = -1$. [24]



[24] p rechter Sohn $\Rightarrow \text{bal}(pp) = \text{bal}(p) = +1$, Linksrotation

Rotationen

Fall 1.2 $\text{bal}(p) = +1$. [25]



[25] p rechter Sohn $\Rightarrow \text{bal}(pp) = +1$, $\text{bal}(p) = -1$, Doppelrotation rechts links

Das Löschen hat 3 Fälle:

1. Knoten n hat zwei Blätter als Kinder: Löschen und rebalancieren.
2. n hat einen Knoten k als Kind: n durch k ersetzen., rebalancieren.
3. n hat zwei Kinder: n durch symmetrischen Nachfolger ersetzen (und diesen wie in 1 oder 2 löschen), rebalancieren.

6.4.4 Heaps

[Max-]Heap

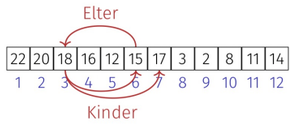
Binärer Baum mit folgenden Eigenschaften

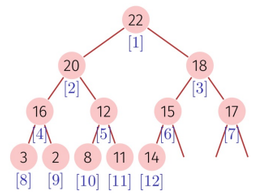
1. vollständig, bis auf die letzte Ebene
2. Lücken des Baumes in der letzten Ebene höchstens rechts.
3. **Heap-Bedingung:**
   Max-(Min-)Heap: Schlüssel eines Kindes kleiner (grösser) als der des Elternknotens

Heap als Array

Baum → Array:

- Kinder$(i) = \{2i, 2i+1\}$
- Elter$(i) = \lfloor i/2 \rfloor$



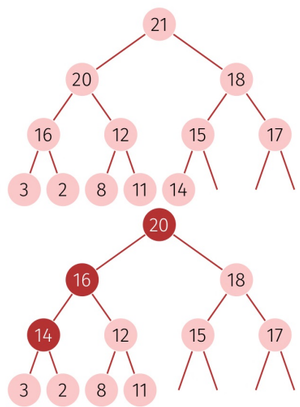


Abhängig vom Startindex![19]

[19] Für Arrays, die bei 0 beginnen: $\{2i, 2i+1\} \to \{2i+1, 2i+2\}$, $\lfloor i/2 \rfloor \to \lfloor (i-1)/2 \rfloor$

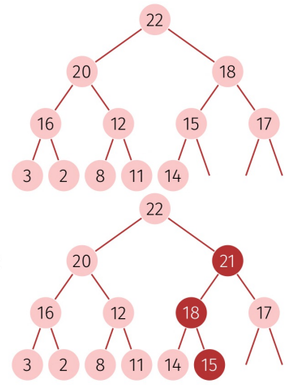
Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: Sukzessives Absinken (in Richtung des grösseren Kindes).
- Anzahl Operationen im schlechtesten Fall: $\mathcal{O}(\log n)$



Einfügen

- Füge neuen Schlüssel an erster freien Stelle ein. Verletzt Heap-Eigenschaft potenziell.
- Stelle Heap-Eigenschaft wieder her: Sukzessives Aufsteigen
- Anzahl Operationen im schlechtesten Fall: $\mathcal{O}(\log n)$



Algorithmus Aufsteigen(A, m)

Input: Array A mit mindestens m Schlüsseln und Heapstruktur auf $A[1, \ldots, m-1]$
Output: Array A mit Heapstruktur auf $A[1, \ldots, m]$

```
v ← A[m] // Neuer Schlüssel
c ← m // Index derzeitiger Knoten (child)
p ← ⌊c/2⌋ // Index Elternknoten (parent)
while c > 1 and v > A[p] do
    A[c] ← A[p] // Schlüssel Elternknoten → Schlüssel derzeitiger Knoten
    c ← p // Elternknoten → derzeitiger Knoten
    p ← ⌊c/2⌋
A[c] ← v // Neuen Schlüssel platzieren
```

Algorithmus Huffman(C)

Input: Codewörter $c \in C$
Output: Wurzel eines optimalen Codebaums

```
n ← |C|
Q ← C
for i = 1 to n − 1 do
    Alloziere neuen Knoten z
    z.left ← ExtractMin(Q)          // Extrahiere Wort mit minimaler Häufigkeit.
    z.right ← ExtractMin(Q)
    z.freq ← z.left.freq + z.right.freq
    Insert(Q, z)
return ExtractMin(Q)
```

Graphen

Handschlag-Lemma:

In jedem Graphen $G = (V, E)$ gilt

1. $\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|$, falls G gerichtet
2. $\sum_{v \in V} \deg(v) = 2|E|$, falls G ungerichtet.

Allg. gilt:

- Allgemein: $0 \leq |E| \in \mathcal{O}(|V|^2)$
- Zusammenhängender Graph: $|E| \in \Omega(|V|)$
- Vollständiger Graph: $|E| = \frac{|V| \cdot (|V|-1)}{2}$ (ungerichtet)
- Maximal $|E| = |V|^2$ (gerichtet),$|E| = \frac{|V| \cdot (|V|+1)}{2}$ (ungerichtet)

Algorithmus Tiefensuche DFS-Visit(G)

Input: Graph $G = (V, E)$

foreach $v \in V$ **do**
  $v.color \leftarrow$ white

foreach $v \in V$ **do**
  if $v.color =$ white **then**
    DFS-Visit(G,v)

Tiefensuche für alle Knoten eines Graphen. Laufzeit $\Theta(|V| + \sum_{v \in V}(\deg^+(v) + 1)) = \Theta(|V| + |E|)$.

Algorithmus Tiefensuche DFS-Visit(G, v)

Input: Graph $G = (V, E)$, Knoten v.

$v.color \leftarrow$ grey
// besuche v
foreach $w \in N^+(v)$ **do**
  if $w.color =$ white **then**
    DFS-Visit(G, w)
$v.color \leftarrow$ black

Tiefensuche ab Knoten v. Laufzeit (ohne Rekursion): $\Theta(\deg^+ v)$

Konzeptuelle Färbung der Knoten
- **Weiss:** Knoten wurde noch nicht entdeckt.
- **Grau:** Knoten wurde entdeckt und zur Traversierung vorgemerkt / in Bearbeitung.
- **Schwarz:** Knoten wurde entdeckt und vollständig bearbeitet

Algorithmus Topological-Sort(G)

Input: Graph $G = (V, E)$.
Output: Topologische Sortierung ord

Stack $S \leftarrow \emptyset$
foreach $v \in V$ **do** $A[v] \leftarrow 0$
foreach $(v, w) \in E$ **do** $A[w] \leftarrow A[w] + 1$ // Eingangsgrade berechnen
foreach $v \in V$ with $A[v] = 0$ **do** push(S, v) // Merke Nodes mit Eingangsgrad 0
$i \leftarrow 1$
while $S \neq \emptyset$ **do**
  $v \leftarrow$ pop(S); ord$[v] \leftarrow i$; $i \leftarrow i + 1$ // Wähle Knoten mit Eingangsgrad 0
  foreach $(v, w) \in E$ **do** // Verringere Eingangsgrad der Nachfolger
    $A[w] \leftarrow A[w] - 1$
    if $A[w] = 0$ **then** push(S, w)

if $i = |V| + 1$ **then return** ord **else return** "Cycle Detected"

Theorem 21

Ein gerichteter Graph $G = (V, E)$ besitzt genau dann eine topologische Sortierung, wenn er kreisfrei ist

(Iteratives) BFS-Visit(G, v)

Input: Graph $G = (V, E)$

Queue $Q \leftarrow \emptyset$
enqueue(Q, v)
v.visited $\leftarrow$ true
while $Q \neq \emptyset$ **do**
  $w \leftarrow$ dequeue(Q)
  // besuche w
  foreach $c \in N^+(w)$ **do**
    if c.visited $=$ false **then**
      c.visited $\leftarrow$ true
      enqueue(Q, c)

$\Theta(|V| + |E|)$

Algorithmus kommt mit $\mathcal{O}(|V|)$ Extraplatz aus.

A*-Algorithmus($G, s, t, \hat{h}$)

Input: Positiv gewichteter Graph $G = (V, E, c)$, Startpunkt $s \in V$, Endpunkt $t \in V$, Schätzung $\hat{h}(v) \leq \delta(v, t)$
Output: Existenz und Wert eines kürzesten Pfades von s nach t

foreach $u \in V$ **do**
  $d[u] \leftarrow \infty$; $\hat{f}[u] \leftarrow \infty$; $\pi[u] \leftarrow$ null
$d[s] \leftarrow 0$; $\hat{f}[s] \leftarrow \hat{h}(s)$; $N \leftarrow \{s\}$; $K \leftarrow \{\}$
while $N \neq \emptyset$ **do**
  $u \leftarrow$ ExtractMin$_{\hat{f}}(N)$; $K \leftarrow K \cup \{u\}$
  if $u = t$ **then return** success
  foreach $v \in N^+(u)$ with $d[v] > d[u] + c(u, v)$ **do**
    $d[v] \leftarrow d[u] + c(u, v)$; $\hat{f}[v] \leftarrow d[v] + \hat{h}(v)$; $\pi[v] \leftarrow u$
    $N \leftarrow N \cup \{v\}$; $K \leftarrow K - \{v\}$
return failure

Der A*-Algorithmus ist eine Erweiterung des Dijkstra-Algorithmus um eine Abstandsheuristik $\hat{h}$.
- A* = Dijkstra wenn $\hat{h} \equiv 0$.
- Wenn $\hat{h}$ ausserdem monoton ist, dann ist der Algorithmus auch effizient.
- Wenn $\hat{h}$ den Abstand unterschätzt, funktioniert der Algorithmus korrekt.
- In der Praxis (z.B. Routing) ist die Wahl von $\hat{h}$ oft intuitiv klar und führt zu deutlich verbessertem Verhalten im Vergleich zu Dijkstra.

Union-Find Algorithmus MST-Kruskal(G)

Input: Gewichteter Graph $G = (V, E, c)$
Output: Minimaler Spannbaum mit Kanten A.

Sortiere Kanten nach Gewicht $c(e_1) \leq ... \leq c(e_m)$
$A \leftarrow \emptyset$
for $k = 1$ **to** $|V|$ **do**
  MakeSet(k)
for $k = 1$ **to** m **do**
  $(u, v) \leftarrow e_k$
  if Find$(u) \neq$ Find(v) **then**
    Union(Find(u), Find(v))
    $A \leftarrow A \cup e_k$
  else
return (V, A, c)

$\mathcal{O}(|E| \cdot \log|V|)$ // konzeptuell:

Theorem 27

Sei $G = (V, E)$ ein Graph und $k \in \mathbb{N}$. Dann gibt das Element $a_{i,j}^{(k)}$ der Matrix $(a_{i,j}^{(k)})_{1 \leq i,j \leq n} = (A_G)^k$ die Anzahl der Wege mit Länge k von v_i nach v_j an.

Algorithmus Floyd-Warshall(G)

Input: Graph $G = (V, E, c)$ ohne Tyklen mit negativem Gewicht.
Output: Minimale Gewichte aller Pfade d
$d^0 \leftarrow c$
for $k \leftarrow 1$ **to** $|V|$ **do**
  for $i \leftarrow 1$ **to** $|V|$ **do**
    for $j \leftarrow 1$ **to** $|V|$ **do**
      $d^k(v_i, v_j) = \min\{d^{k-1}(v_i, v_j), d^{k-1}(v_i, v_k) + d^{k-1}(v_k, v_j)\}$

Laufzeit: $\Theta(|V|^3)$
Bemerkung: Der Algorithmus kann auf einer einzigen Matrix d (in place) ausgeführt werden.

- Sortieren der Kanten: $\Theta(|E| \log |E|) = \Theta(|E| \log |V|)$. [46]
- Initialisieren der Union-Find Datenstruktur $\Theta(|V|)$
- $|E| \times$ Union(Find(x),Find(y)): $\mathcal{O}(|E| \log |E|) = \mathcal{O}(|E| \log |V|)$.

Insgesamt $\Theta(|E| \log |V|)$.

Algorithmus: Dijkstra(G, s)

Input: Positiv gewichteter Graph $G = (V, E, c)$, Startpunkt $s \in V$
Output: Länge d_s der kürzesten Pfade von s und Vorgänger π_s für jeden Knoten

foreach $u \in V$ **do**
  $d_s[u] \leftarrow \infty$; $\pi_s[u] \leftarrow$ null
$d_s[s] \leftarrow 0$; $N \leftarrow \{s\}$
while $N \neq \emptyset$ **do**
  $u \leftarrow \arg\min_{u \in N} d_s[u]$; $N \leftarrow N \setminus \{u\}$
  foreach $v \in N^+(u)$ **do**
    if $d_s[u] + c(u, v) < d_s[v]$ **then**
      $d_s[v] \leftarrow d_s[u] + c(u, v)$
      $\pi_s[v] \leftarrow u$
      $N \leftarrow N \cup \{v\}$

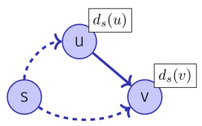
- $|V| \times$ ExtractMin: $\mathcal{O}(|V| \log |V|)$
- $|E| \times$ Insert oder DecreaseKey: $\mathcal{O}(|E| \log |V|)$
- $1 \times$ Init: $\mathcal{O}(|V|)$
- Insgesamt[41] [42] :

$$\mathcal{O}((|V| + |E|) \log |V|)$$

Allgemeiner Algorithmus

Laufzeit: $\Theta(|V| \cdot |E|)$

1. Initialisiere d_s und π_s: $d_s[v] = \infty$, $\pi_s[v] =$ null für alle $v \in V$
2. Setze $d_s[s] \leftarrow 0$
3. Wähle eine Kante $(u, v) \in E$
   Relax(u, v):
   if $d_s[u] + c(u, v) < d_s[v]$ **then**
     $d_s[v] \leftarrow d_s[u] + c(u, v)$
     $\pi_s[v] \leftarrow u$
     return true
   return false



4. Wiederhole 3 bis nichts mehr relaxiert werden kann. (bis $d_s[v] \leq d_s[u] + c(u, v) \quad \forall (u, v) \in E$)

Algorithmus Bellman-Ford(G, s)

Input: Graph $G = (V, E, c)$, Startpunkt $s \in V$
Output: Wenn Rückgabe true, Minimale Gewichte d der kürzesten Pfade zu jedem Knoten, sonst kein kürzester Pfad.

foreach $u \in V$ **do**
  $d_s[u] \leftarrow \infty$; $\pi_s[u] \leftarrow$ null
$d_s[s] \leftarrow 0$;
for $i \leftarrow 1$ **to** $|V|$ **do**
  $f \leftarrow$ false
  foreach $(u, v) \in E$ **do**
    $f \leftarrow f \vee$ Relax(u, v)
  if $f =$ false **then return** true
return false;

Laufzeit $\mathcal{O}(|E| \cdot |V|)$.

Implementation und Laufzeit

Implementation wie bei Dijkstra's Kürzeste Wege Algorithmus. Einziger Unterschied:

Kürzeste Wege
Relaxiere (u, v):
  if $d_s[v] > d[u] + c(u, v)$ then
    $d_s[v] \leftarrow d_s[u] + c(u, v)$
    $\pi_s[v] \leftarrow u$

$\Rightarrow$

Minimaler Spannbaum
Relaxiere (u, v):
  if $d_s[v] > c(u, v)$ then
    $d_s[v] \leftarrow c(u, v)$
    $\pi_s[v] \leftarrow u$

- Mit Min-Heap, Kosten $\mathcal{O}(|E| \cdot \log |V|)$:
  - Initialisierung (Knotenfärbung) $\mathcal{O}(|V|)$
  - $|V| \times$ ExtractMin = $\mathcal{O}(|V| \log |V|)$,
  - $|E| \times$ Insert oder DecreaseKey: $\mathcal{O}(|E| \log |V|)$,
- Mit Fibonacci-Heap: $\mathcal{O}(|E| + |V| \cdot \log |V|)$.

Algorithmus von Jarnik (1930), Prim, Dijkstra (1959)

Idee: Starte mit einem $v \in V$ und lasse von dort einen Spannbaum wachsen:

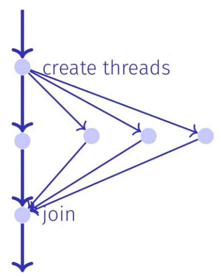
$A \leftarrow \emptyset$
$S \leftarrow \{v_0\}$
for $i \leftarrow 1$ **to** $|V|$ **do**
  Wähle billigste (u, v) mit $u \in S$, $v \notin S$
  $A \leftarrow A \cup \{(u, v)\}$
  $S \leftarrow S \cup \{v\}$ // (Färbung)

Parallel Programming

C++11 Threads

```
void hello(int id){
  std::cout << "hello from " << id << "\n";
}

int main(){
  std::vector<std::thread> tv(3);
  int id = 0;
  for (auto & t:tv)
    t = std::thread(hello, ++id);
  std::cout << "hello from main \n";
  for (auto & t:tv)
    t.join();
  return 0;
}
```



Locks

```
class BankAccount {
  int balance = 0;
  std::mutex m; // requires #include <mutex>
public:
  ...
  void withdraw(int amount) {
    m.lock();
    int b = getBalance();
    setBalance(b - amount);
    m.unlock();
  }
};
```

RAII Ansatz

```
class BankAccount {
  int balance = 0;
  std::mutex m;
public:
  ...
  void withdraw(int amount) {
    std::lock_guard<std::mutex> guard(m);   // m.lock(); im Konstruktor vom Guard
    int b = getBalance();
    setBalance(b - amount);                 // m.unlock(); im Destruktor vom Guard
  } // Destruction of guard leads to unlocking m
};
```

Greedy Scheduler: teilt zu jeder Zeit so viele Tasks zu Prozessoren zu wie möglich.

Theorem 36

Auf einem idealen Parallelrechner mit p Prozessoren führt ein Greedy-Scheduler eine mehrfädige Berechnung mit Arbeit T_1 und Zeitspanne T_∞ in Zeit

$$T_p \leq T_1/p + T_\infty$$

aus.

3.1.1 Amdahls Gesetz

Amdahl teilt die nötige Rechenarbeit W in einen parallelisierbaren Teil W_p und einen nicht parallelisierbaren, sequentiellen Teil W_s auf. Die sequentielle Ausführungszeit sei T_1, die parallele T_p mit p Prozessoren. Es gilt $T_p \geq T_1/p$. Bei Gleichheit ist Perfektion erreicht, was eigentlich unmöglich ist.

Der Speedup S_p berechnet sich dann aus

$$S_p = \frac{T_1}{T_p} \leq \frac{W_s + W_p}{W_s + \frac{W_p}{p}}$$

Benutzt man den prozentualen Anteil $\lambda : W_s = \lambda W \Rightarrow W_p = (1-\lambda)W$:

$$S_p \leq \frac{1}{\lambda + \frac{1-\lambda}{p}}$$

$$S_\infty \leq \frac{1}{\lambda}$$

Beispiel:
Ein Programm ist zu 80% parallelisierbar, $T_1 = 10$
Berechne den Speedup für 8 und ∞ Prozessoren

$$T_8 = \frac{10 \cdot 0.8}{8} + 10 \cdot 0.2 = 3$$

$$S_8 = \frac{T_1}{T_8} = \frac{10}{3} \approx 3.3 < 8 S_\infty \leq \frac{1}{0.2} = 5$$

Die Gesetze von Amdahl und Gustafson sind Modelle der Laufzeitverbesserung bei Parallelisierung. Amdahl geht von einem festen **relativen** sequentiellen Anteil der Arbeit aus, während Gustafson von einem festen **absoluten** sequentiellen Teil ausgeht (der als Bruchteil der Arbeit W_1 ausgedrückt wird und bei Zunahme der Arbeit nicht wächst). Die beiden Modelle widersprechen sich nicht, sondern beschreiben die Laufzeitverbesserung verschiedener Probleme und Algorithmen.

3.1.2 Gustafsons Gesetz

Gustafson hält die Ausführungszeit fest und variiert die Problemgrösse. Man nimmt an, dass der Sequentielle Teil konstant ist und der parallele Teil grösser wird.
Die Arbeit, die mit p Prozessoren in der Zeit T erledigt werden kann, ist also

$$W_s + p \cdot W_p = \lambda \cdot T + p(1-\lambda)T$$

Der Speedup ist dann

$$S_p = \frac{W_s + p \cdot W_p}{W_s + W_p} = p \cdot (1-\lambda) + \lambda \quad (3)$$

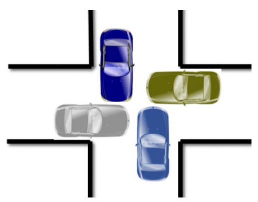
$$= p - \lambda(p-1) \quad (4)$$

Data Race (low-level Race-Conditions) Fehlerhaftes Programmverhalten verursacht durch ungenügend synchronisierten Zugriff zu einer gemeinsam genutzten Resource, z.B. gleichzeitiges Lesen/Schreiben oder Schreiben/Schreiben zum gleichen Speicherbereich.

Bad Interleaving (High Level Race Condition) Fehlerhaftes Programmverhalten verursacht durch eine unglückliche Ausführungsreihenfolge eines Algorithmus mit mehreren Threads, selbst dann wenn die gemeinsam genutzten Resourcen anderweitig gut synchronisiert sind.

} Race Conditions

Deadlock: zwei oder mehr Prozesse sind gegenseitig blockiert, weil jeder Prozess auf einen anderen Prozess warten muss, um fortzufahren.



C++11 Stil

```
class BankAccount {
  ...
  std::recursive_mutex m;
  using guard = std::lock_guard<std::recursive_mutex>;
public:
  ...
  void transfer(int amount, BankAccount& to){
    std::lock(m,to.m); // lock order done by C++
    // tell the guards that the lock is already taken:
    guard g(m,std::adopt_lock); guard h(to.m,std::adopt_lock);
    withdraw(amount);
    to.deposit(amount);
  }
};
```

Atomic

Hier auch möglich:

```
class C {
  std::atomic_int x{0}; // requires #include <atomic>
  std::atomic_int y{0};
public:
  void f() {
    x = 1;
    y = 1;
  }
  void g() {
    int a = y;
    int b = x;
    assert(b >= a); // cannot fail
  }
};
```

```
#include <mutex>
#include <thread>
#include <condition_variable>
using mutex = std::mutex;
using guard = std::unique_lock<mutex>;
using condition = std::condition_variable;

class ReentrantLock {
  std::thread::id id;
  unsigned count;
  mutex m;
  condition cond;

public:
  ReentrantLock(): count(0) {
  }

  void lock(){
    guard g(m);
    std::thread::id current = std::this_thread::get_id();
    if(count == 0){
      id = current;
    }else if(id != current){
      cond.wait(g, [&]{return count == 0;});
      id = current;
    }
    count++;
  }

  void unlock(){
    guard g(m);
    if(count > 0){
      count--;
      cond.notify_one();
    }
  }
};
```

Bedingungsvariablen

```
class Buffer {
...
public:
    void put(int x){
        guard g(m);
        buf.push(x);
        cond.notify_one();
    }
    int get(){
        guard g(m);
        cond.wait(g, [&]{return !buf.empty();});
        int x = buf.front(); buf.pop();
        return x;
    }
};
```

Konzept von Read-Modify-Write: Der Effekt von Lesen, Verändern und Zurückschreiben, wird zu einem Zeitpunkt sichtbar (geschieht atomar).

Pseudo-Code für CAS – Compare-And-Swap

```
bool CAS(int& variable, int& expected, int desired){
  // atomic:
  if (variable == expected){
    variable = desired;
    return true;
  }
  else{
    expected = variable;
    return false;
  }
}
```

Nachfolgend ist das zu einer Union-Find-Datenstruktur zugehörige Array dargestellt. Geben Sie das Array an, nachdem die Operation Union(Find(3),Find(5)) mit der Optimierung „Small under Large" ausgeführt wurde.

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

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

1P if a[6]=2 (no optimization)

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Tree Sort	O(n log(n))	O(n log(n))	O(n^2)	O(n)
Shell Sort	O(n log(n))	O(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(k)
Cubesort	O(n)	O(n log(n))	O(n log(n))	O(n)

Graph Data Structure Operations

Data Structure	Time Complexity					
	Storage	Add Vertex	Add Edge	Remove Vertex	Remove Edge	Query
Adjacency list	O(|V|+|E|)	O(1)	O(1)	O(|V| + |E|)	O(|E|)	O(|V|)
Incidence list	O(|V|+|E|)	O(1)	O(1)	O(|E|)	O(|E|)	O(|E|)
Adjacency matrix	O(|V|^2)	O(|V|^2)	O(1)	O(|V|^2)	O(1)	O(1)
Incidence matrix	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|E|)

Heap Data Structure Operations

Data Structure	Time Complexity					
	Find Max	Extract Max	Increase Key	Insert	Delete	Merge
Binary Heap	O(1)	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(m+n)
Pairing Heap	O(1)	O(log(n))	O(log(n))	O(1)	O(log(n))	O(1)
Binomial Heap	O(1)	O(log(n))	O(log(n))	O(1)	O(log(n))	O(log(n))
Fibonacci Heap	O(1)	O(log(n))	O(1)	O(1)	O(log(n))	O(1)

Graph Algorithms

Algorithm	Time Complexity		Space Complexity
	Average	Worst	Worst
Dijkstra's algorithm	O(|E| log |V|)	O(|V|^2)	O(|V| + |E|)
A* search algorithm	O(|E|)	O(b^d)	O(b^d)
Prim's algorithm	O(|E| log |V|)	O(|V|^2)	O(|V| + |E|)
Bellman–Ford algorithm	O(|E| · |V|)	O(|E| · |V|)	O(|V|)
Floyd-Warshall algorithm	O(|V|^3)	O(|V|^3)	O(|V|^2)
Topological sort	O(|V| + |E|)	O(|V| + |E|)	O(|V| + |E|)

Subset Sum

Aufgabe: sei $z = \frac{1}{2}\sum_{i=1}^{n} a_i$. Suche Auswahl $I \subset \{1, \ldots, n\}$, so dass $\sum_{i \in I} a_i = z$.

DP-Tabelle: $[0, \ldots, n] \times [0, \ldots, z]$-Tabelle T mit Wahrheitseinträgen. $T[k, s]$ gibt an, ob es eine Auswahl $I_k \subset \{1, \ldots, k\}$ gibt, so dass $\sum_{i \in I_k} a_i = s$.

Initialisierung: $T[0, 0] =$ true. $T[0, s] =$ false für $s > 1$.

Berechnung:

$$T[k, s] \leftarrow \begin{cases} T[k-1, s] & \text{falls } s < a_k \\ T[k-1, s] \vee T[k-1, s-a_k] & \text{falls } s \geq a_k \end{cases}$$

für aufsteigende k und innerhalb k dann s.

Knapsack

Tabelleneintrag $t[i, w]$ enthält statt Wahrheitswerten das jeweils grösste v, das erreichbar ist[25] mit

- den Gegenständen $1, \ldots, i$ $(0 \leq i \leq n)$
- bei höchstem zulässigen Gewicht w $(0 \leq w \leq W)$.

Initial

- $t[0, w] \leftarrow 0$ für alle $w \geq 0$.

Berechnung

$$t[i, w] \leftarrow \begin{cases} t[i-1, w] & \text{falls } w < w_i \\ \max\{t[i-1, w], t[i-1, w-w_i] + v_i\} & \text{sonst.} \end{cases}$$

aufsteigend nach i und für festes i aufsteigend nach w.

Lösung steht in $t[n, w]$

DP-Beispiel **Ausfüllen der Nebendiagonalen**

```
Number max_value_dp(const Expression& expression){
  unsigned n = expression.numbers.size();
  Matrix memo(n,std::vector<Number>(n,Min_Number));
  // fill diagonal
  for (unsigned pos = 0; pos < n ; ++pos){
    memo[pos][pos] = expression.numbers[pos];
  }
  // compute off-diagonals
  for (unsigned dist = 1; dist < n; ++dist){
    for (unsigned from = 0; from+dist < n; ++from){
      unsigned to = from + dist;
      Number max = Min_Number;
      for (unsigned mid = from; mid < to; ++mid){
        Number v1 = memo[from][mid];
        Number v2 = memo[mid+1][to];
        Number val = evaluate(v1, expression.operators[mid],v2);
        if (val > max){
          max = val;
        }
      }
      assert(max != Min_Number);
      memo[from][to] = max;
    }
  }

  print_solution(expression,memo,0,n-1); std::cout << "\n";

  return memo[0][n-1];
}
```

Angenommen jede Operation eines Algorithmus auf einer Datenstruktur hat amortisierte Laufzeit von $O(T(n))$. Dann ist die Laufzeit für eine Folge von n Operationen auf der initial leeren Datenstruktur im schlechtesten Fall $O(n \cdot T(n))$.

☒ Wahr / ○ Falsch

```
int f(int n){
    if(n<=1){
        return 1;
    } else {
        int x = f(n-1)*f(n-2);
        return x;
    }
}
```

Asymptotische Laufzeit von f / *Asymptotic*

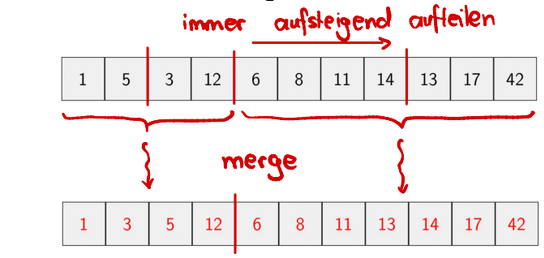
Running time of f

$\Theta((\frac{1+\sqrt{5}}{2})^n)$

$\Theta(\Phi^n)$ also ok

any exponential: 1P

Natürliches 2-Wege Mergesort:



Amortized Analysis

Idee: In Algorithmen können grosse Operationen durch viele kleine Operationen kompensiert, dabei kann eine bessere **amortisierte Laufzeit** erreicht werden. Es können folgende drei Methoden benutzt werden:

Aggregierte Analyse

Direkte Argumentation: berechne eine Schranke für die Gesamtzahl der Elementaroperationen und teile durch die Anzahl der Operationen

Kontomethode

Der Computer basiert auf Münzen: jede Elementaroperation der Maschine kostet eine Münze.

Für jede Operation op_k einer Datenstruktur wird eine bestimme Anzahl Münzen a_k auf eine Konto A eingezahlt: $A_k = A_{k-1} + a_k$

Die Münzen vom Konto A werden verwendet, um die anfallenden echten Kosten t_k zu bezahlen.

Das Konto A muss zu jeder Zeit genügend Münzen aufweisen, um die laufende Operation op_k zu bezahlen: $A_k - t_k \geq 0 \forall k$.

a_k sind die amortisierten Kosten der Operation op_k.

Potentialmethode

Bezeichne t_i die realen Kosten der Operation op_i.
Potentialfunktion $\Phi_i \geq 0$ zur Datenstruktur nach i Operationen.
Voraussetzung: $\Phi_i \geq \Phi_0 \ \forall i$.
Amortisierte Kosten der i-ten Operation:

$$a_i := t_i + \Phi_i - \Phi_{i-1}.$$

Es gilt nämlich

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (t_i + \Phi_i - \Phi_{i-1}) = \left(\sum_{i=1}^{n} t_i\right) + \Phi_n - \Phi_0 \geq \sum_{i=1}^{n} t_i.$$