D&A - Fynn Krebser
fkrebser@student.ethz.ch

Effizienz

Def. (Gross-O-Notation) $f \in \mathcal{O}(g)$, wenn $\exists c > 0$, $n_0 \in \mathbb{N}$, sodass

$$\forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n).$$

Def. (Gross-Ω-Notation) $f \in \Omega(g)$, wenn $\exists c > 0$, $n_0 \in \mathbb{N}$, sodass

$$\forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n).$$

Def. (Gross-Θ-Notation) $\Theta(g) = O(g) \cap \Omega(g)$.

Satz (Master Theorem) Für $a \geq 1, b > 1$ gilt für $T(n) = a * T(\frac{n}{b}) + f(n)$:

1. $f(n) \in \mathcal{O}(n^{\log_b a - \epsilon}) \Rightarrow T(n) \in \Theta(n^{\log_b a})$
2. $f(n) \in \Theta(n^{\log_b a}) \Rightarrow T(n) \in \Theta(n^{\log_b a} \log n)$
3. $f(n) \in \Omega(n^{\log_b a + \epsilon})$ und $af(\frac{n}{b}) \leq cf(n)$ für $c < 1$ $\Rightarrow T(n) \in \Theta(f(n))$

Beispiele

Alg. (Ägyptische Multiplikation) Multiplikation von a und b durch sukzessives Halbieren von a und Verdoppeln von b.

```
int mul(a, b):
  int result = 0;
  while (a > 0){
    if (a % 2 == 1) result += b;
    a /= 2; b *= 2;
  }
  return result
```

Alg. (Karatsuba-Offmann) Schnelleres Multiplizieren von zwei n-stelligen Zahlen x und y durch Aufteilung in zwei $\frac{n}{2}$-stellige Zahlen a, b und c, d und Berechnung von drei Produkten statt vier.

```
int karatsuba(x, y):
  if (x < 10 || y < 10) return x * y;
  n = max(len(x), len(y));
  a, b = div,mod(x, 10**(n/2));
  c, d = div,mod(y, 10**(n/2));
  ac = karatsuba(a, c);
  bd = karatsuba(b, d);
  abcd = karatsuba(a + b, c + d);
  return ac * 10**n + (abcd - ac - bd) * 10**(n/2) + bd;
```

Alg. (Maximum Subarray Problem) Finden des zusammenhängenden Teilarrays mit der größten Summe in einem Array von Zahlen. Verwenden von Sliding Window Technik, um die Summe effizient zu berechnen.

```
M=0; R=0;
for (i=0; i<n; i++):
  R = max(R + A[i], A[i]);
  M = max(M, R);
return M;
```

Suchen

Alg. (Binäre Suche) Suche nach einem Element x in einem **sortierten** Array A. Laufzeit: $\Theta(\log n)$

```
int binarySearch(int A[], int n, int x) {
  int left = 0, right = n - 1;
  while (left <= right) {
    int mid = left + (right - left) / 2;
    if (A[mid] == x) return mid;
    else if (A[mid] < x) left = mid + 1;
    else right = mid - 1;
    }
    return -1; // Element nicht gefunden
```

Auswählen

Finde das i-te kleinste Element in einem Array A der Länge n.

Alg. (Median of Medians) Teilt das Array in Gruppen von 5 Elementen, findet den Median jeder Gruppe, und verwendet den Median der Mediane als Pivot, um das Array zu partitionieren. Laufzeit: $\Theta(n)$

Alg. (Pivotieren) Partitioniert ein Array A um ein Pivot-Element p, so dass alle Elemente kleiner als p links von p und alle Elemente größer als p rechts von p liegen. Gibt die endgültige Position von p zurück.

```
l = 1; r=n; lhat = none; rhat = none;
loop{
  while (A[l] < p) l++;
  while (A[r] > p) r--;
  if (l >= r){
    if (lhat != none) swap(A[lhat], A[r]); return r;
    else if (rhat != none) swap(A[rhat], A[l]); return l;
    return r;
  }
  swap(A[l], A[r]);
  if (A[l] == p) lhat = l;
  if (A[r] == p) rhat = r;
  l++; r--;
}
```

Laufzeiten für Auswahlalgorithmen:

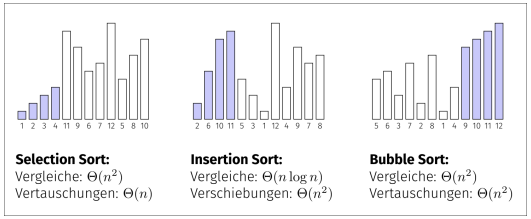
Repeatedly find min	$\Theta(n^2)$
Sort and select	$\Theta(n \log n)$
Randomized selection	$\Theta(n)$ (expected)
Median of medians	$\Theta(n)$ (worst-case)

Sortieren

Alg. (Selection Sort) Sortiert ein Array A der Länge n durch wiederholtes Finden des kleinsten Elements und Verschieben an den Anfang des Arrays. Laufzeit: $\Theta(n^2)$

Alg. (Insertion Sort) Sortiert ein Array A der Länge n durch schrittweises Einfügen von Elementen an die richtige Position in einem bereits sortierten Teil des Arrays. Laufzeit: $\Theta(n^2)$

Alg. (Bubble Sort) Sortiert ein Array A der Länge n durch wiederholtes Vertauschen benachbarter Elemente, wenn sie in der falschen Reihenfolge sind. Laufzeit: $\Theta(n^2)$



Alg. (Merge Sort) Sortiert ein Array A der Länge n durch rekursive Aufteilung in kleinere Subarrays, Sortieren dieser Subarrays und anschließendes Zusammenführen der sortierten Subarrays. Zusammenführen kann mit zwei Zeigern erfolgen, um die Elemente effizient zu vergleichen und zu kopieren. Laufzeit: $\Theta(n \log n)$

Alg. (Quick Sort) Sortiert ein Array A der Länge n durch Auswahl eines Pivot-Elements, Partitionierung des Arrays um das Pivot und rekursives Sortieren der Partitionen. Die Wahl des Pivots kann die Leistung beeinflussen, worst case $\Theta(n^2)$ Vergleiche. Swaps in $\Theta(n \log n)$. Zufälliger Pivot führt zu erwarteter Laufzeit von $\Theta(n \log n)$.

Satz Vergleichbasiertes sortieren von n Elementen durch Vergleiche erfordert im worst case $\Omega(n \log n)$ Vergleiche.

Alg. (Radix Exchange Sort) Sortiert ein Array A der Länge n von d-stelligen Zahlen durch Sortieren der Ziffern von der niedrigsten zur höchsten Stelle. Verwendet einen stabilen Sortieralgorithmus (z.B. Counting Sort) für jede Ziffer. Laufzeit: $\Theta(d \cdot n \cdot k)$, wobei k die Anzahl der möglichen Werte pro Ziffer ist.

Alg. (Bucket Sort) Sortiert ein Array A der Länge n von Zahlen im Bereich $[0, 1)$ durch Aufteilung in n gleich große Intervalle (Buckets), Einfügen der Elemente in die entsprechenden Buckets, Sortieren jedes Buckets (z.B. mit Insertion Sort) und anschließendes Zusammenführen der sortierten Buckets. Laufzeit: $\Theta(l(n + r))$, wobei l die Länge der Zahlen und r die Radix der Zahlen ist.

Datenstrukturen

Def. (Abstrakter Datentyp) Ein abstrakter Datentyp (ADT) ist eine mathematische Modellierung einer Datenstruktur, die eine Menge von Operationen definiert, ohne die Implementierung dieser Operationen zu spezifizieren. Beispiel: Liste, Set, Dictionary, Stack, Queue, Graph, Baum

Def. (Datenstruktur) Eine Datenstruktur ist eine konkrete Implementierung eines abstrakten Datentyps, die die Operationen des ADTs effizient unterstützt. Beispiel: Array, verkettete Liste, Hash-Tabelle, Binärbaum, Heap

Bsp. (Stack) Ein Stack ist ein ADT, der die Last-In-First-Out (LIFO) Semantik unterstützt. Er bietet die Operationen `push` (Element hinzufügen), `pop` (Element entfernen).Implementierung: Linked List.

Bsp. (Queue) Eine Queue ist ein ADT, der die First-In-First-Out (FIFO) Semantik unterstützt. Er bietet die Operationen `enqueue` (Element hinzufügen), `dequeue` (Element entfernen). Implementierung: (Doubly) Linked List.

Bsp. (Linked List) Singly Linked mit pointer zum Kopf: `std::forward_list` in C++. Doubly Linked mit pointer zum Kopf und Ende: `std::list` in C++.

Amortisierte Analyse

Konzept (Aggregierte Analyse) Berechne eine obere Schranke für die Gesamtkosten einer Sequenz von n Operationen und teile durch n, um die amortisierten Kosten pro Operation zu erhalten.

Konzept (Kontomethode) Jede elementare Operation kostet eine Münze, die von einem zu beginn leeren Konto kommen. Für jede Operation kommen a_k Münzen auf das Konto. Mit diesen bezahlen wir die wahren Kosten t_k der Operation. Wenn das Konto nie leer ist, so sind die amortisierten Kosten von op_k höchstens a_k.

Konzept (Potentialmethode) Definere Potential Φ_i assoziert mit dem Zustand der Datenstruktur zur Zeit i. Das Potential sollte ansteigen mit billigen und sinken mit teuren Operationen. Wir verlangen $\Phi_i > \Phi_0$. Die amortisierten Kosten sind dann $a_i := t_i + \Phi_i - \Phi_{i-1}$.

Bsp. (Stack) Wähle als Potential die Anzahl der Elemente im Stack. Dann haben `push` Operationen Kosten 1 und erhöhen das Potential um 1, also amortisierte Kosten 2. `pop` Operationen kosten 1 und verringern das Potential um 1, also amortisierte Kosten 0. Somit haben beide Operationen amortisierte Kosten von $\mathcal{O}(1)$.

Bsp. Vector: pushback, popback konstant, Deque: pushfront, popfront, pushback, popback konstant, Dictionary: insert $\log n$, find $\log^2 n$.

Suchbäume

Def. (Binärer Baum) Entweder ein Blatt oder ein innerer Knoten, mit einem Binary Tree als linkes und einem Binary Tree als rechtes Kind.

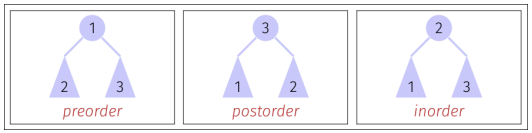
Def. (Binärer Suchbaum) Ein binärer Baum, bei dem für jeden inneren Knoten gilt: Alle Werte im linken Teilbaum sind kleiner als der Wert des Knotens, und alle Werte im rechten Teilbaum sind größer als der Wert des Knotens.

Alg. (Suchen im bin. Suchbaum) Suche nach einem Schlüssel *key* in einem binären Suchbaum mit Wurzel *root*. Laufzeit: $\Theta(h)$, wobei h die Höhe des Baums ist.

```
Node* search(Node* root, int key) {
  if (root == nullptr || root->value == key) return root;
    if (key < root->value) return search(root->left, key);
    else return search(root->right, key);
}
```

Konzept (Löschen im bin. Suchbaum) 1. Knoten ist Blatt: Einfach entfernen. 2. Knoten hat ein Kind: Ersetze den Knoten durch sein Kind. 3. Knoten hat zwei Kinder: Finde den Inorder-Nachfolger (kleinster Knoten im rechten Teilbaum) oder Inorder-Vorgänger (größter Knoten im linken Teilbaum), ersetze den Wert des zu löschenden Knotens durch den Wert des Nachfolgers/Vorgängers, und lösche dann den Nachfolger/Vorgänger-Knoten.

Laufzeit: $\Theta(h)$, wobei h die Höhe des Baums ist.



Traversierungsarten eines binären Baums

Heaps

Def. (Max Heap) Binärer Baum, der die Max-Heap-Eigenschaft erfüllt: Jeder Knoten ist grösser oder gleich seinen Kindern. Zusätzlich ist der Baum vollständig, d.h. alle Ebenen ausser möglicherweise der letzten sind vollständig gefüllt, und die letzte Ebene ist von links nach rechts gefüllt.

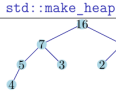
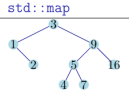
Das Maximum ist immer in der Wurzel!

Konzept Ein Heap kann effizient in einem Array dargestellt werden, wobei die Kinder eines Knotens bei Index i an den Indizes $2i+1$ und $2i+2$ liegen, und der Elternknoten bei Index $\lfloor (i-1)/2 \rfloor$ liegt.

Konzept (Einfügen) Füge ein neues Element am Ende des Heaps (letzte Position im Array) hinzu, und führe dann "Heapify Up"durch: Vergleiche das neue Element mit seinem Elternknoten und tausche sie, wenn das neue Element größer ist. Wiederhole diesen Vorgang, bis die Max-Heap-Eigenschaft wiederhergestellt ist. Laufzeit: $\Theta(\log n)$

Konzept (Entfernen des Maximums) Entferne das Maximum (Wurzel) und ersetze es durch das letzte Element im Heap (letzte Position im Array). Führe dann "Heapify Down"durch: Vergleiche das neue Wurzelelement mit seinen Kindern und tausche es mit dem größeren Kind, wenn es kleiner ist. Wiederhole diesen Vorgang, bis die Max-Heap-Eigenschaft wiederhergestellt ist. Laufzeit: $\Theta(\log n)$

Konzept (Heapify) Um einen unsortierten Array in einen Max Heap zu verwandeln, rufe "Heapify Downffür alle Nicht-Blatt-Knoten von der letzten Nicht-Blatt-Ebene bis zur Wurzel auf. Laufzeit: $\Theta(n)$

	Suchbäume	Heaps Min- / Max- Heap	Balancierte Bäume AVL, Rot-Schwarz
in C++:		`std::make_heap`	`std::map`
			
Einfügen	$\Theta(h(T))$	$\Theta(\log n)$	$\Theta(\log n)$
Suchen	$\Theta(h(T))$	$\Theta(n)$ (!!)	$\Theta(\log n)$
Löschen	$\Theta(h(T))$	Suchen + $\Theta(\log n)$	$\Theta(\log n)$
Min/Max	$\Theta(h(T))$	$\Theta(1)$ / Suchen	$\Theta(\log n)$

Alg. (Heapsort) Sortiert ein Array A der Länge n durch Umwandlung in einen Max Heap und anschließendes wiederholtes Entfernen des Maximums, um die Elemente in aufsteigender Reihenfolge zu sortieren. Laufzeit: $\Theta(n \log n)$

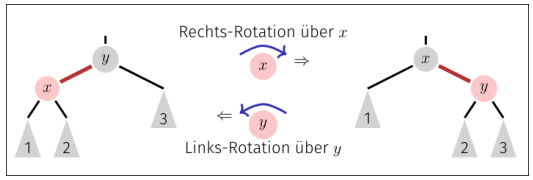
Balancierte Suchbäume

Def. (2-3-Baum) Ein 2-3-Baum ist ein selbstbalancierender Suchbaum, bei dem jeder Knoten entweder 2 oder 3 Kinder hat und alle Blätter die gleiche Tiefe haben.

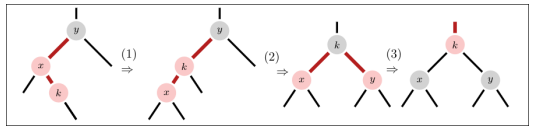
Alg. (Einfügen in 2-3-Baum) Füge ein neues Element in einen 2-3-Baum ein, indem du es in das passende Blatt einfügst. Wenn das Blatt bereits 2 Elemente enthält, teile es in zwei Blätter auf und verschiebe das mittlere Element nach oben. Wiederhole diesen Vorgang, bis die Baumstruktur wiederhergestellt ist. Die Höhe des Baums erhöht sich nur, wenn die Wurzel geteilt wird. Laufzeit: $\Theta(\log n)$

Def. (Rot-Schwarz-Baum) Ein Rot-Schwarz-Baum ist ein selbstbalancierender binärer Suchbaum, bei dem jeder Knoten entweder rot oder schwarz ist und die folgenden Eigenschaften erfüllt: 1. Die Wurzel ist schwarz. 2. Alle Blätter sind schwarz. 3. Rote Knoten haben keine roten Kinder (keine zwei roten Knoten dürfen direkt verbunden sein). 4. Jeder Pfad von einem Knoten zu seinen Blättern enthält die gleiche Anzahl schwarzer Knoten. Ausserdem verlangen wir, dass der Baum left-leaning ist, d.h. dass alle roten Knoten linke Kinder sind.

Alg. (Suchen) Analog zur Suche im binären Suchbaum.



Alg. (Einfügen in Rot-Schwarz-Baum) Füge ein neues Element als roten Knoten ein. Überprüfe die Rot-Schwarz Eigenschaften und führe notwendige Rotationen und Farbänderungen durch, um die Eigenschaften wiederherzustellen. Laufzeit: $\Theta(\log n)$



Hash Tabellen

Konzept (Hashing) Ziel ist eine assoziative Datenstruktur, die Schlüssel-Wert-Paare speichert und effiziente Operationen für Einfügen, Suchen und Löschen bietet. Eine Hash-Tabelle verwendet eine Hash-Funktion, um einen Schlüssel in einen Index eines Arrays zu transformieren, wo der entsprechende Wert gespeichert wird.

C++: `std::unordered_map`

Konzept (Prehashing) $ph : \mathcal{K} \to \mathbb{N}$ bildet alle Schlüssel auf natürliche Zahlen ab. Hat sehr grosse Zielmenge.

Konzept (Hash-Funktion) $h : \mathbb{N} \to \{0, 1, \ldots, m-1\}$ bildet natürliche Zahlen auf die Indizes eines Arrays der Länge m ab. Ziel ist es, Kollisionen zu minimieren, d.h. verschiedene Schlüssel sollten möglichst unterschiedliche Indizes erhalten.

Alg. (Division Method) $h(k) = k \mod m$. Einfach, aber kann zu vielen Kollisionen führen, wenn die Schlüssel Muster aufweisen.

Alg. (Multiplication Method) $h(k) = \lfloor (a \cdot k \mod 2^w)/2^{w-r} \rfloor \mod m$, mit a irrational, w der Wortgröße und r der Anzahl der Bits, die für die Indizes benötigt werden.

Konzept (Open Hashing) Jeder Index im Array ist der Startpunkt einer verketteten Liste. Worst Case, $\Theta(n)$, wenn alle Schlüssel den gleichen Index erhalten. Belegungsfaktor grösser als 1 möglich.

Konzept (Closed Hashing) Alle Schlüssel werden direkt im Array gespeichert. Bei Kollisionen wird eine alternative Position gesucht, z.B. durch lineares Sondieren (nächster freier Slot), quadratisches Sondieren oder doppelte Hashing. Belegungsfaktor muss kleiner als 1 sein, um gute Leistung zu gewährleisten.

Alg. (Double Hashing) Bei Kollisionen wird eine zweite Hash-Funktion $h_2(k)$ verwendet, um die Schrittweite für die Suche nach einem freien Slot zu bestimmen: $h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m$. Dies reduziert die Wahrscheinlichkeit von Clustering im Vergleich zum linearen oder quadratischen Sondieren.

$$h_2(k) = 1 + (k \mod (m-2)).$$

Um zu löschen werden die gelöschten Slots als "Belegtmmarkiert, damit die Suche korrekt funktioniert. Beim einfügen können diese Slots wiederverwendet werden.

Konzept (Rehashing) Wenn der Belegungsfaktor einen bestimmten Schwellenwert überschreitet, wird die Größe der Tabelle verdoppelt und alle bestehenden Schlüssel werden mit einer neuen Hash-Funktion neu gehasht und in die neue Tabelle eingefügt. Amortisierte erwartete Laufzeit: $\Theta(1)$ für Einfügen, Suchen und Löschen.

Konzept (Hashing) Worst Case Laufzeit für Einfügen, Suchen und Löschen: $\Theta(n)$, wenn alle Schlüssel den gleichen Index erhalten. Erwartete Laufzeit: $\Theta(1)$, wenn die Hash-Funktion gut ist und die Schlüssel gleichmäßig verteilt sind.

Quadtrees

Def. (Punkt-Quadtree) Ein Punkt-Quadtree ist ein hierarchisches Datenstruktur, bei der jeder Punkt in einem 2-D Raum als Knoten dargestellt wird. Jeder Knoten hat vier Kinder, die die vier Quadranten des Raums repräsentieren, der durch die vertikale und horizontale Linie durch den Punkt definiiert wird. Vergleichsweise viele Knoten.

Def. (k-d-Baum) Ein k-d-Baum ist eine hierarchische Datenstruktur, die Punkte in einem k-dimensionalen Raum organisiert. Jeder Knoten teilt den Raum entlang einer der k Dimensionen, wobei die Dimensionen zyklisch durchlaufen werden. Jeder Knoten hat zwei Kinder, die die beiden Hälften des Raums repräsentieren, die durch die Teilung definiert werden. Vergleichsweise wenige Knoten und nicht balanciert.

Def. (Region-Quadtree) 2-D Raum wird rekursiv in vier Quadranten unterteilt, bis eine bestimmte Auflösung erreicht ist oder alle Punkte in einem Quadranten gleich sind. Jeder Knoten repräsentiert einen Quadranten und hat vier Kinder, die die vier Unterquadranten repräsentieren. Vergleichsweise wenige Knoten.

Alg. (Erstellen eines Quadtreeknoten) Create a new node with given coordinates.

```
QNode* q = new QNode();
q.area = Rectangle(x,y,w,h);
q.ul = q.ur = q.ll = q.lr = nullptr;
q.split = false;
q.points = empty list;
```

Alg. (Insert) Einfügen eines Punktes p in einen Quadtree.

```
if (p in q.area){
  if (q.split){
    Insert(p, q.ul); Insert(p, q.ur); Insert(p, q.ll); Insert(p,
    ↪ q.lr);
  }
  else{
    q.points.append(p);
    if (q.points.size() > threshold){
      Split(q);
      for (Point pt : q.points){
        Insert(pt, q);
      }
      q.points.clear();
    }
  }
}
```

Convex Hull

Def. (Convex Hull) Der konvexe Hülle eines Satzes von Punkten in der Ebene ist die kleinste konvexe Menge, die alle Punkte enthält. Geometrisch entspricht dies der Form eines Gummibands, das um die Punkte gespannt wird.

Alg. (Gift-Wrapping) Beginne mit dem Punkt mit der kleinsten x-Koordinate (und bei Gleichstand der kleinsten y-Koordinate). Wähle diesen Punkt als Startpunkt und füge ihn zum konvexen Hülle hinzu. Füge stets den nächsten Punkt hinzu, der den größten Winkel zum letzten hinzugefügten Punkt und dem vorherigen Punkt bildet. Laufzeit: $\Theta(nh)$, wobei n die Anzahl der Punkte und h die Anzahl der Punkte in der konvexen Hülle ist.

```
list<Point> H = {};
Point p0 = leftmost point in S;
Point q = p0;
do {
  H.push_back(q);
  r = S[0];
  for (auto s:S){
    if ((r-q)x(s-q) < 0) r = s;
  }
  q = r;
} while (q != p0);
```

Alg. (Graham-Scan) Betrachte untere und obere Hülle separat. Sortiere die Punkte nach x-Koordinate

und verbinde sie. Lösche nun alle Punkte, die eine Rechtsdrehung bilden, bis nur noch die Punkte der unteren Hülle übrig sind. Wiederhole den Prozess für die obere Hülle. Verbinde schließlich die beiden Hüllen. Laufzeit: $\Theta(n \log n)$, mit Sortieren, $\Theta(n)$ für die Konstruktion der Hülle.

```
Stack<Point> S = {};
S.push(points[0]);S.push(points[1]);
for (int i=2; i<n; i++){
  while ((S.top() - S.secondTop()) x (points[i] - S.secondTop()) <=
  ↪ 0){
    S.pop();
  }
  S.push(points[i]);
}
```

Lambdas

Konzept (Lambda-Ausdrücke) Syntax in C++: `[capture](parameters) -> return_type {body}`

Note that `[]` und `()` nie weggelassen werden können.

Konzept (Capture) `[=]`: Alle Variablen werden per Wert erfasst (kopiert). `[&]`: Alle Variablen werden per Referenz erfasst. `[this]`: Erfasst den Zeiger auf das aktuelle Objekt. `[x, &y]`: Erfasst die Variable x per Wert und die Variable y per Referenz.

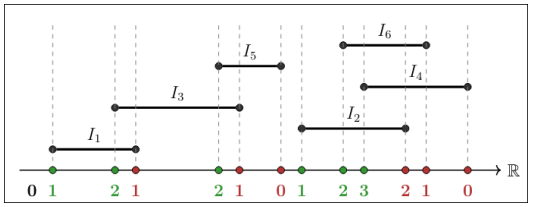
Bsp. Aufsummieren eines Vektors mit foreach

```
std::vector<int> a {1,2,3,4,5};
int sum = 0;
std::for_each(a.begin(), a.end(), [&sum](int x) { sum += x; });
std::cout << "Summe: " << sum << std::endl; // Ausgabe: Summe: 15
```

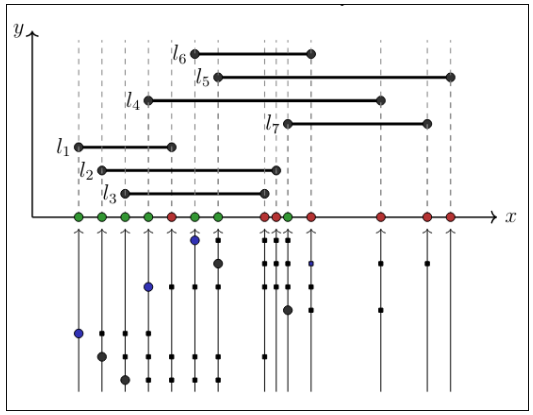
Sweepline

Konzept (Sweepline) 2-D Problem welches einfach ist für einzelne x-Koordinaten und nur an diskreten Orten Änderungen aufweist (Eventpunkte).

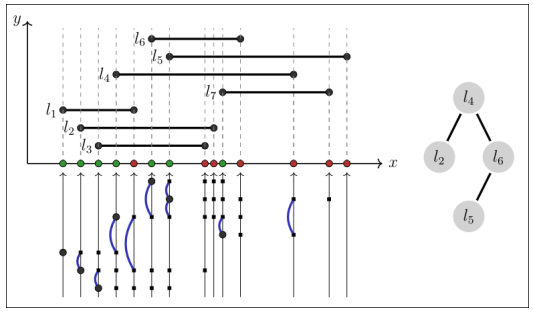
Alg. (Überlappende Intervalle) Finde maximale Anzahl überlappender Intervalle. Eventpunkte: Start- und Endpunkte der Intervalle gespeichert in Sorted Array. Sweepline Status: Anzahl der aktuell überlappenden Intervalle. Update: Bei Startpunkt +1, bei Endpunkt -1. Laufzeit: $\Theta(n \log n)$ für Sortieren der Eventpunkte, $\Theta(n)$ für die Sweepline.



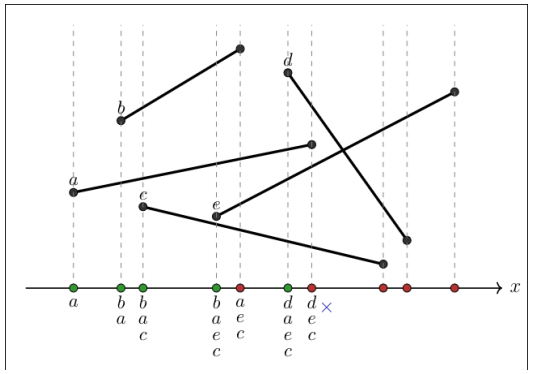
Alg. (Oberste Strecken) Finde alle Strecken, welche für ein $x \in \mathbb{R}$ zuoberst sind. Eventpunkte: Start- und Endpunkte der Strecken in Sorted Array. Sweepline Status: Aktive Strecken, sortiert nach y-Koordinate an der Sweepline, gespeichert in RB-Tree. Update: Bei Startpunkt füge Strecke hinzu, bei Endpunkt entferne Strecke. Laufzeit: $\Theta(n \log n)$ für Sortieren der Eventpunkte, $\Theta(n \log n)$ für die Sweepline (Einfügen und Entfernen von Strecken).



Alg. (Benachbarte Strecken) Finde alle Paare von Strecken, welche für ein $x \in \mathbb{R}$ benachbart sind. Eventpunkte: Start- und Endpunkte der Strecken in Sorted Array. Sweepline Status: Aktive Strecken, sortiert nach y-Koordinate an der Sweepline, gespeichert in RB-Tree. Update: Bei Startpunkt füge Strecke hinzu, bei Endpunkt entferne Strecke. Überprüfe bei Einfügen einer Strecke die benachbarten Strecken auf Benachbarung, und bei Entfernen einer Strecke die beiden benachbarten Strecken auf Benachbarung. Laufzeit: $\Theta(n \log n)$ für Sortieren der Eventpunkte, $\Theta(n \log n)$ für die Sweepline (Einfügen und Entfernen von Strecken).



Alg. (Schnittpunkte von Strecken) Finde alle Schnittpunkte von Strecken. Eventpunkte: Start- und Endpunkte der Strecken, sowie Schnittpunkte in minheap/RB-Tree. Sweepline Status: Aktive Strecken, sortiert nach y-Koordinate an der Sweepline, gespeichert in RB-Tree. Update: Bei Startpunkt füge Strecke hinzu, bei Endpunkt entferne Strecke, bei Schnittpunkt tausche die beiden Strecken. Überprüfe bei Einfügen oder Entfernen einer Strecke die benachbarten Strecken auf Schnittpunkte, und bei einem Schnittpunkt die neuen benachbarten Strecken auf Schnittpunkte. Laufzeit: $\Theta((n+k) \log n)$, wobei k die Anzahl der Schnittpunkte ist.



Alg. (Schnittpunktdetektion) Bestimme ob sich zwei strecken schneiden ist äquivalent zu der Frage ob

$$sgn((p_s - p_e) \times (p_s - q_s)) \neq sgn((p_s - p_e) \times (p_s - q_e)),$$

für die Strecken $s = (p_s, p_e)$ und $t = (q_s, q_e)$ (und umgekehrt).

Dichtestes Punktepaar

Alg. (Dichtestes Punktepaar) Finde das dichteste Paar von Punkten in einem Satz von n Punkten in der Ebene. 1. Sortiere die Punkte nach x-Koordinate. 2. Teile die Punkte in zwei Hälften und finde rekursiv das dichteste Paar in jeder Hälfte. 3. Finde das dichteste Paar, das über die Mitte hinweg liegt, indem du nur Punkte in einem Streifen der Breite 2δ um die Mittellinie betrachtest, wobei δ die minimale Distanz der beiden Hälften ist. Sortiere diese Punkte nach y-Koordinate und überprüfe nur die nächsten 7 Punkte, um das dichteste Paar zu finden. Laufzeit: $\Theta(n \log n)$, mit Sortieren und Rekursion.

Graphen

Def. (Graph) Ein Graph G ist ein Paar (V, E), wobei V eine Menge von Knoten (Vertices) und E eine Menge von Kanten (Edges) ist, die Paare von Knoten verbinden. Ein Graph kann gerichtet oder ungerichtet sein, abhängig davon, ob die Kanten eine Richtung haben oder nicht.

Satz Ein ungerichteter zusammenhängender Graph hat einen 1. Eulerschen Kreis genau dann wenn alle Knoten geraden Grad haben. 2. Eulerschen Pfad genau dann wenn genau 0 oder 2 Knoten ungeraden Grad haben.

Satz (Handshaking Lemma) In jedem Graphen ist $\sum_v \deg^-(v) = \sum_v \deg^+(v) = |E|$. Insbesondere ist für ungerichtete Graphen $\sum_v \deg(v) = 2|E|$ gerade.

Alg. (Tiefensuche (DFS)) Durchlaufe einen Graphen G beginnend bei einem Startknoten s und besuche alle erreichbaren Knoten, indem du so tief wie möglich in den Graphen vordringst, bevor du zurückkehrst und andere Pfade erkundest. Laufzeit: $\Theta(|V| + |E|)$

```
v.color = GRAY;
  for (auto w: v.succ){
    if (w.color == WHITE){
      DFS(G,w);
    }
  }
v.color = BLACK;
```

Alg. (Breitensuche (BFS)) Durchlaufe einen Graphen G beginnend bei einem Startknoten s und besuche alle erreichbaren Knoten, indem du zuerst alle Nachbarn von s besuchst, dann die Nachbarn der Nachbarn usw. Laufzeit: $\Theta(|V| + |E|)$

```
std::queue<Node*> Q;
enqueue(Q, s);
s.seen = true;
while (!Q.empty()){
  u = dequeue(Q);
  for (auto w: u.succ){
    if (!w.seen){
      enqueue(Q, w);
      w.seen = true;
    }
  }
}
```

Def. (Topologische Sortierung) Anordnung der Knoten eines gerichteten azyklischen Graphen (DAG) in einer linearen Reihenfolge, so dass für jede gerichtete Kante (u, v) von Knoten u zu Knoten v gilt, dass u vor v in der Reihenfolge kommt.

Satz Ein gerichteter Graph hat genau dann eine topologische Sortierung, wenn er azyklisch ist.

Alg. (Topologisch Sortieren) Bestimme den Eingangsgrad aller Knoten und füge die mit Eingangsgrad 0 in eine Queue ein. Solange die Queue nicht leer ist, entferne einen Knoten u aus der Queue, füge ihn zur topologischen Sortierung hinzu, und verringere den Eingangsgrad aller Nachfolger von u um 1. Wenn der Eingangsgrad eines Nachfolgers 0 wird, füge ihn in die Queue ein. Laufzeit: $\Theta(|V| + |E|)$

Kürzeste Wege

Def. (Gewichteter Graph) Ein gewichteter Graph $G = (V, E, c)$ ist ein Graph, bei dem jeder Kante $e \in E$ ein Gewicht $c(e) \in \mathbb{R}$ zugeordnet ist, das die Kosten oder die Länge der Kante repräsentiert.

Satz Der Kürzeste Pfad von einem Startknoten s zu einem Zielknoten t ist nicht zwingend eindeutig.

Alg. (Djikstra) Finde den kürzesten Pfad von einem Startknoten s zu allen anderen Knoten in einem gewichteten Graphen mit nicht-negativen Kantengewichten. Verwende eine Prioritätswarteschlange, um den nächsten Knoten mit der geringsten vorläufigen Distanz zu wählen. Laufzeit: $\Theta(|E| \log |V|)$ (wenn connected).

```
for (auto v: V){
  v.dist = INFINITY; v.prev = nullptr;
}
s.dist = 0;
std::priority_queue<Node*> Q;
Q.push(s);
while (!Q.empty()){
  u = Q.top(); Q.pop();
  for (auto w: u.succ){
    if (u.dist + c(u,w) < w.dist){
      w.dist = u.dist + c(u,w);
      w.prev = u;
      Q.push(w);
} } }
```

Konzept (Lazy-Deletion) Anstatt Prioritätswarteschlange zu aktualisieren, wenn die Distanz eines Knotens verbessert wird, fügen wir den Knoten einfach erneut mit der neuen Distanz hinzu und ignorieren die veralteten Einträge, wenn sie aus der Warteschlange entfernt werden. Nachteil ist Speicherbedarf von Heap auf $\Theta(|V| + |E|)$, anstatt $\Theta(|V|)$ wachsen kann.

Alg. (A*) Finde den kürzesten Pfad von einem Startknoten s zu einem Zielknoten t in einem gewichteten Graphen mit nicht-negativen Kantengewichten. Verwende eine Prioritätswarteschlange, um den nächsten Knoten mit der geringsten vorläufigen Distanz plus einer Heuristik $\hat{h}(v)$ zu wählen, die die geschätzten Kosten von v zum Ziel t darstellt. Laufzeit: $\Theta(|E| \log |V|)$ (wenn connected).

Konzept (Heuristiken) Die Heuristik muss die Distanz unterschätzen, d.h. $\hat{h}(v) \leq h(v)$ für alle Knoten v, wobei $h(v)$ die tatsächlichen Kosten von v zum Ziel t ist. Eine häufig verwendete Heuristik ist die Manhattan-Distanz oder die euklidische Distanz zwischen v und t. Idealerweise ist die Heuristik auch Monoton um mehrfaches entnehmen und einfügen von Knoten zu vermeiden, d.h. $\hat{h}(u) \leq c(u, v) + \hat{h}(v)$ für jede Kante (u, v). (Ansonsten kann sich die Laufzeit verschlechtern.)

Konzept (Relaxieren) Der Prozess der Aktualisierung der vorläufigen Distanz eines Knotens v durch Überprüfen, ob der Pfad von s über einen Knoten u zu v kürzer ist als die bisher bekannte Distanz zu v. Wenn ja, aktualisiere die Distanz von v und setze u als Vorgänger von v.

Satz Ein kürzester Weg enthält keine Zyklen. Folglich hat ein kürzester Weg maximal $|V| - 1$ Kanten.

Alg. (Bellman-Ford) Finde den kürzesten Pfad von einem Startknoten s zu allen anderen Knoten in einem gewichteten Graphen, auch wenn negative Kantengewichte vorhanden sind. Wiederhole den Relaxationsprozess für alle Kanten $V - 1$ Mal. Wenn man beim nächsten mal noch relaxieren kann existiert ein negativer Zyklus. Laufzeit: $\Theta(|V| \cdot |E|)$

```
for (auto v: V){
  v.dist = INFINITY; v.prev = nullptr;
}
s.dist = 0;
for (int i=0; i<|V|; i++){
  bool f = false;
  for (auto e: E) f = relax(e) || f;
  if (!f) return true;
}
return false; // negative cycle exists
```

Konzept (Bellman-Ford auf DAGs) Wenn der Graph ein DAG ist, können wir die Knoten in topologischer Reihenfolge durchlaufen und jede Kante nur einmal relaxieren, um die kürzesten Pfade zu berechnen. Laufzeit: $\Theta(|V| + |E|)$

$n := |V|, m := |E|$

Problem	Methode	Laufzeit	dicht $m \in \mathcal{O}(n^2)$	dünn $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
DAG[1] $(c \in \mathbb{R})$	Top-Sort	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
$c \geq 0$	Dijkstra	$\mathcal{O}((m+n)\log n)$	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(n \log n)$
allgemein $(c \in \mathbb{R})$	Bellman-Ford	$\mathcal{O}(m \cdot n)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$

Alg. (Floyd-Warshall) Finde die kürzesten Pfade zwischen allen Paaren von Knoten in einem gewichteten Graphen. Verwende eine dynamische Programmierung, um die kürzesten Pfade zu berechnen, indem du schrittweise die Anzahl der erlaubten Zwischenknoten erhöhst. Laufzeit: $\Theta(|V|^3)$

```
for (v,w:V){
  if (v==w) dist[v][w] = 0;
  else if ((v,w) in E) dist[v][w] = c(v,w);
  else dist[v][w] = INFINITY;
}
for (k:V){
  for (v,w:V){
    dist_k[v][w] = min(dist_{k-1}[v][w], dist_{k-1}[v][k] +
    ↪  dist_{k-1}[k][w]);
  }
}
```

Alg. (Anzahl Wege) Gegeben Adjanzenzmatrix A eines Gerichteten Graphen, so ist

$$(A^k)_{ij} = \#\text{ Wege } i \to j \text{ mit genau } k \text{ Kanten.}$$

Berechnung in $\Theta(l \cdot |V|^3)$, wobei l die maximale Pfadlänge ist.

Def. (Reflexive Transitive Hülle) gegeben G gerichtet, $G^* = (V, E^*)$ mit

$$E^* = \{(u, v) | \text{ es gibt einen Pfad } u \to v \text{ in } G\}.$$

Alg. Bestimme ob es einen Weg von u nach v gibt, indem die Adjanzenzmatrix A von G mit sich selbst multipliziert wird, bis $(A^k)_{uv} > 0$ oder $k = |V|$. Wir können zudem ein alternatives Matrixprodukt nehmen welches beim Wert 1 gedeckelt ist. Da die Matrizenmultiplikation assoziativ ist, können wir die Potenzen von A effizient mit Exponentiation by Squaring berechnen. Laufzeit: $\Theta(|V|^3 \log |V|)$.

Minimum Spanning Tree

Def. (Spannbaum) Gegeben ein ungerichteter, zusammenhängender Graph $G = (V, E)$, ist ein Spannbaum von G ein Teilgraph $T = (V, E')$ von G, sodass $E' \subset E$ und T ein Baum ist, d.h. T ist zusammenhängend und enthält keine Zyklen. Ein Spannbaum enthält genau $|V| - 1$ Kanten.

Def. (Minimaler Spannbaum) Ein Spannbaum T von einem gewichteten Graphen G ist minimal, wenn die Summe der Gewichte der Kanten in T minimal ist, d.h. es gibt keinen anderen Spannbaum T' von G mit geringeren Gesamtkosten.

Satz Ein MST ist eindeutig, wenn alle Kantengewichte verschieden sind. Ansonsten kann es mehrere MSTs geben.

Alg. (Jarnik, Prim, Djikstra) Finde einen minimalen Spannbaum in einem gewichteten, ungerichteten Graphen. Beginne mit einem Startknoten und füge iterativ die günstigste Kante hinzu, die einen neuen Knoten mit dem bereits gebildeten Teilbaum verbindet, bis alle Knoten verbunden sind. Laufzeit: $\Theta(|E| \log |V|)$ (wenn connected).

Implementation analog Djikstra, aber mit Kantengewicht anstatt vorläufiger Distanz als Priorität.

- to get A*: extend by these parts
- to get Prim: remove these parts

```
for u ∈ V do
    d[u] ← ∞; π[u] ←null; f̂[u] ← ∞
end
U = {s}; d[s] ← 0                                   /* Put only s in U and keep track */
while U ≠ ∅ do
    u ← extractMin(U) using s[u] ( use f̂[u] instead )   /* Get nearest u */
    for v ∈ N(u) with d[u]+ c(u, v) < d[v] do
        d[v] ← d[u]+ c(u, v)                        /* Update distance to v */
        π[v] ← u                                    /* Store predecessor */
        f̂[v] ← d[v] + ĥ[v]                          /* Update estimates */
        U ← U ∪ {v}                                 /* Queue v */
    end
end
```

Algorithm 22.2: Dijkstra as the base for three algorithms

Alg. (Kruskal) Sortiere die Kanten nach Gewicht und füge sie nacheinander zum Teilbaum hinzu, solange sie keinen Zyklus bilden, bis alle Knoten verbunden sind. Laufzeit: $\Theta(|E| \log |E|)$ (für Sortieren) + $\Theta(|E|\alpha(|V|))$ (für Union-Find), insgesamt $\Theta(|E| \log |V|)$.

Def. (Union-Find) Datenstruktur zur Verwaltung von disjunkten Mengen, die effiziente Operationen zum Finden der Menge eines Elements (Find) und zum Vereinen zweier Mengen (Union) bietet.

Interne Darstellung: Jeder Knoten zeigt auf seinen Elternknoten, und die Wurzel eines Baums repräsentiert die Menge. Find-Operation: Folge die Elternzeiger, bis die Wurzel erreicht ist wobei alle Knoten direkt an die Wurzel angehängt werden können. Union-Operation: Verbinde die Wurzeln von zwei Mengen, wobei die kleinere Menge an die größere angehängt wird (Union by Size/Rank), um die Höhe der Bäume zu minimieren. Laufzeit: $\Theta(\alpha(n))$ pro Operation, wobei α die inverse Ackermann-Funktion ist, die für alle praktischen Werte von n kleiner als 5 ist.

Max Flow - Min Cut

Def. (Flussnetzwerk) Ein Flussnetzwerk ist ein gerichteter Graph $G = (V, E)$ mit einer Kapazitätsfunktion $c : E \to \mathbb{R}^+$, einem Quellknoten s und einem Senkenknoten t. Jede Kante (u, v) hat eine Kapazität $c(u, v)$, die die maximale Menge an Fluss angibt, die durch diese Kante fließen kann.

Def. (Fluss) Funktion $f : E \to \mathbb{R}^+$, sodass $f(e) \leq c(e)$ und

$$\sum_{e \in E^-(v)} f(e) = \sum_{e \in E^+(v)} f(e) \quad \forall v \in V \setminus \{s, t\}.$$

Def. (Restkapazität) Für eine Kante (u, v) in einem Flussnetzwerk mit Fluss f ist die Restkapazität definiert als $c_f(u, v) = c(u, v) - f(u, v)$. Sie gibt an, wie viel zusätzlicher Fluss durch die Kante (u, v) fließen kann, bevor die Kapazität erreicht ist. Die Restkapazität eines Weges ist $r(p) = \min_{e \in p} c_f(e)$.

Alg. (Ford-Fulkerson) Führe die Möglichkeit ein Flüsse Rückwärts zu schicken ein, indem wir die Restkapazität auch für die Rückwärtskante definieren als $c_f(v, u) = f(u, v)$. Solange es einen Pfad von s nach t mit positiver Restkapazität gibt, schicke den maximal möglichen Fluss entlang dieses Pfades und aktualisiere die Restkapazitäten entsprechend. Laufzeit: $\Theta(|E| \cdot \text{maximaler Fluss})$.

Alg. (Edmonds-Karp) Implementierung von Ford-Fulkerson, bei der der Pfad mit der geringsten Anzahl von Kanten (kürzester Pfad) in jedem Schritt gewählt wird. Laufzeit: $\Theta(|V| \cdot |E|^2)$.

```
// BFS
template <typename C>
bool get_augmenting_path(DiGraph<C>& g, std::vector<int>& path) {
  int n = std::ssize(g); int source = 0; int sink = n - 1;
  path.clear();

  std::vector<int> predecessor(n, n); predecessor[source] = source;

  std::deque<int> queue; queue.push_back(source);

  while (!queue.empty()) {
    int current = queue.front(); queue.pop_front();
    for (int i = 0; i < n; ++i) {
      if (i != current && predecessor[i] == n) {
        C residual_capacity = g.get_edge_capacity(current, i) -
        ↪  g.get_flow_value(current, i);
        if (residual_capacity > 0) {
          predecessor[i] = current;
          if (i == sink) {
            for (int j = i; j != source; j = predecessor[j])
            ↪  path.push_back(j);
            path.push_back(source);
            return true;
          }
          queue.push_back(i);
  } } } }
  return false;
}

template <typename C>
void max_flow(DiGraph<C>& g) {
  std::vector<int> path;

  while (get_augmenting_path(g, path)) {
    C path_cost = g.get_edge_capacity(path[1], path[0]) -
    ↪  g.get_flow_value(path[1], path[0]);
    for (int i = 0; i < std::ssize(path) - 1; ++i) {
      C current = g.get_edge_capacity(path[i + 1], path[i]) -
      ↪  g.get_flow_value(path[i + 1], path[i]);
      if (current < path_cost)
        path_cost = current;
    }
    for (int i = 0; i < std::ssize(path) - 1; ++i) {
      int a = path[i + 1], b = path[i];  // path is in reverse order
      g.set_flow_value(a, b, g.get_flow_value(a, b) + path_cost);
      g.set_flow_value(b, a, g.get_flow_value(b, a) - path_cost);
    }
  }
}
```

Def. (Cut) Ein Schnitt ist eine Partition (S, T) von V mit $s \in S, t \in T$. Ausserdem ist

$$c(S, T) = \sum_{u \in S, v \in T} c(u, v)$$

$$f(S, T) = \sum_{e \in E(S,T)} f(e) - \sum_{e \in E(T,S)} f(e).$$

Satz $|f_{\max}| \leq c(S, T)$ für alle Schnitte (S, T).

Satz (Max-Flow-Min-Cut) Folgende Aussagen sind äquivalent: f ist ein maximaler Fluss von s nach t. Das Restnetzwerk enthält keinen Pfad von s nach t. Es gibt einen Schnitt (S,T) mit $|f| = c(S,T)$

Alg. (Bipartite Matching) Gegeben ein Graph, gesucht zwei Bipartite Mengen so dass keine Kanten innerhalb einer Menge sind. Ein Matching ist eine Teilmenge von Kanten, so dass kein Knoten in mehr als einer Kante enthalten ist. Ein maximales Matching ist ein Matching mit der größtmöglichen Anzahl von Kanten.

Finden mit Max-Flow Algorithmus: Erstelle ein Flussnetzwerk, indem du eine Quelle s hinzufügst, die mit allen Knoten der einen Menge verbunden ist, und eine Senke t, die mit allen Knoten der anderen Menge verbunden ist. Setze die Kapazität aller Kanten auf 1. Finde den maximalen Fluss von s nach t. Die Kanten, die im maximalen Fluss verwendet werden, bilden das maximale Matching.

Dynamische Programmierung

Konzept (Memoisierung) Bei einer Rekursion, anstatt stets alles neu zu berechnen können Werte in einer Tabelle gespeichert werden.

Konzept (Dynamische Programmierung) Das Problem von unten herauf konstruieren, also Rekursion vermeiden.

Konzept (Verfahren für DP) 1. Top-Down Rekursion 2. Memoisierung 3. Bottom-Up Iteration 4. Platz sparen (optional) 5. Rekonstruktion der Lösung (evtl separate Tabelle) 6. Laufzeit: Anzahl der Teilprobleme × Zeit pro Teilproblem.

Alg. (Rod-Cutting) Gegeben einen Stab der Länge n und eine Preisliste p_i für Stäbe der Länge i, finde die maximale Einnahme, die durch das Schneiden des Stabs erzielt werden kann. Definiere r_j als die maximale Einnahme für einen Stab der Länge j. Dann gilt:

$$r_j = \max_{1 \le i \le j} (p_i + r_{j-i}).$$

Speichere ausserdem in einer Tabelle die Länge des ersten Schnitts, um die Lösung rekonstruieren zu können. Laufzeit: $\Theta(n^2)$. Rekonstruktion: $\Theta(n)$.

Alg. (Longest Increasing Subsequence) Gegeben eine Sequenz von Zahlen, finde die Länge der längsten streng monoton zunehmenden Teilsequenz. Definiere $M_{i,j}$ als eine wachsende Teilsequenz der Länge j in den ersten i Elementen der Sequenz, welches mit dem kleinsten möglichen letzten Element endet. Um M_{k+1} zu berechnen, suche das größte j mit $M_{k,j} < a_{k+1}$ und setze $M_{k+1,j+1} = a_{k+1}$, ansonsten $M_{k+1,1} = a_{k+1}$. Laufzeit: $\Theta(n^2)$, mit binärer Suche $\Theta(n \log n)$.

```cpp
std::vector<int> longest_increasing_subsequence(const
    std::vector<int>& sequence) {
  int n = sequence.size();
  // Step 1: Initialise DP tables
  std::vector<int> last_elements(n, pos_inf);
  last_elements.at(0) = neg_inf;

  std::vector<int> predecessor_values(n, pos_inf);

  // Steps 2,3: Compute DP table entries according to their
  ↪ dependencies

  for (int i = 1; i < n; ++i) { // Ignore sequence[0] (see comment in
  ↪ main())
    int a_k = sequence.at(i);

    // Use std::upper_bound performs to perform a binary search for
    ↪ a_k,
    // and to obtain an iterator to the first element largest than
    ↪ a_k.
    // The iterator thus corresponds to T[l+1] from the lecture
    ↪ slides.
    auto upper_it = std::upper_bound(last_elements.begin(),
    ↪ last_elements.end(), a_k); // TODO: replace
    ↪ last_elements.end() with i (+- 1)

    *upper_it = a_k;
    predecessor_values.at(i) = *(upper_it - 1);
  }
  // Step 4: Reconstruct solution

  // Use std::lower_bound to perform a binary search for pos_inf,
  // and to obtain an iterator to the first element not smaller than
  ↪ pos_inf.
  // The iterator thus corresponds to T[l] from the lecture slides.
  auto lower_it = std::lower_bound(last_elements.begin(),
  ↪ last_elements.end(), pos_inf);
  int l = lower_it - last_elements.begin() - 1;
  int e = last_elements.at(l); // or e = *(lower_it - 1)

  // At this point we know that the LIS has length l, and we
  ↪ instantiate
  // a result vector with corresponding length, to store the LIS in.
  // During solution reconstruction, we fill the result vector back to
  ↪ front.
  std::vector<int> result(l);
  auto result_it = result.end() - 1;

  l = n;
  while (e != neg_inf) {
    // Next line corresponds to "find i, < l such that sequence[i] ==
    ↪ e" from
    // the lecture slides, but here we update l directly instead of
    ↪ introducing
    // a dedicated variable i.
    for (--l; sequence.at(l) != e; --l);

    // Store next LIS element in result vector
    *result_it = e; --result_it;

    e = predecessor_values.at(l);
  }
  return result;
}
int main() {
  int n;
  // To simplify the use of indices in the LIS algorithm, we ignore
  // sequence[0], and start at sequence index 1 (as done on the
  ↪ slides).
  ++n;
  std::vector<int> sequence(n);
  sequence[0] = 0; // Not necessary, will be ignored anyway
  // Fill sequence with a permutation of the first n numbers
  std::iota(++sequence.begin(), sequence.end(), 1); // Numbers 1..n
  auto result = longest_increasing_subsequence(sequence);
}
```

Alg. (Editierdistanz) Gegeben zwei Strings s und t, finde die minimale Anzahl von Einfüge-, Lösch- und Ersetzungsoperationen, um s in t zu transformieren. Definiere $E(i,j)$ als die Editierdistanz zwischen den Präfixen $s[1..i]$ und $t[1..j]$. Dann gilt:

$$E(i,j) = \min \begin{cases} E(i-1,j) + del(s[i]) \\ E(i,j-1) + ins(t[j]) \\ E(i-1,j-1) + (s[i] = t[j] ? 0 : 1) \end{cases}$$

Laufzeit: $\Theta(|s| \cdot |t|)$

Alg. (Matrixmultiplikation) Gegeben eine Kette von Matrizen $A_1, A_2, \ldots, A_n$ mit Dimensionen $p_0 \times p_1$, $p_1 \times p_2$, $\ldots$, $p_{n-1} \times p_n$, finde di e minimale Anzahl von Skalar-Multiplikationen, um das Produkt $A_1 A_2 \cdots A_n$ zu berechnen. Definiere $M(i,j)$ als die minimale Anzahl von Multiplikationen, um das Produkt der Matrizen von A_i bis A_j zu berechnen. Dann gilt für $i < j$:

$$M(i,j) = \min_{i \le k < j} (M(i,k) + M(k+1,j) + r_i \cdot c_k \cdot c_j).$$

Für $i = j$ ist $M(i,i) = 0$. Laufzeit: $\Theta(n^3)$. Rekonstruktion der optimalen Klammerung durch Speicherung des optimalen k für jedes Teilproblem. $\Theta(n^2)$ zusätzlicher Speicher, Rekonstruktion linear.

Alg. (Subset Sum) Gegeben eine Menge von n positiven ganzen Zahlen und eine Zielsumme S, finde heraus, ob es eine Teilmenge der Zahlen gibt, deren Summe genau S ist. Definiere $DP(i,s)$ als wahr, wenn es eine Teilmenge der ersten i Zahlen gibt, die die Summe s ergibt, und falsch sonst. Dann gilt:

$$DP(i,s) = DP(i-1,s) \vee DP(i-1, s-a_i).$$

Für $s < 0$ ist $DP(i,s) = false$, und für $s = 0$ ist $DP(i,s) = true$. Laufzeit: $\Theta(n \cdot S)$

Alg. (Optimal Search Tree) Gegeben n Schlüssel $k_1 < k_2 < \ldots < k_n$ und ihre Suchwahrscheinlichkeiten $p_1, p_2, \ldots, p_n$, finde den binären Suchbaum mit minimaler erwarteter Suchkosten. Definiere $E(i,j)$ als die minimalen erwarteten Kosten für die Schlüssel k_i bis k_j. Dann gilt:

$$E(i,j) = \min_{i \le r \le j} (E(i,r-1) + E(r+1,j) + \sum_{m=i}^{j} p_m).$$

Für $i > j$ ist $E(i,j) = 0$, und für $i = j$ ist $E(i,j) = p_i$. Laufzeit: $\Theta(n^3)$ analog zur Matrixmultiplikation. Rekonstruktion des Baums durch Speicherung des optimalen r für jedes Teilproblem. $\Theta(n^2)$ zusätzlicher Speicher, Rekonstruktion linear.

Konzept (DP-Problem) Für eine komplette Lösung soll vorhanden sein: 1. Definition von Teilproblemen / DP-Tabelle 2. Rekursionsformel / DP-Formel 3. Reihenfolge der Berechnung der DP-Tabelle 4. Lösungsrekonstruktion und Laufzeit.

Greedy Algorithmen

Def. (Greedy-Choice-Property) Ein Problem hat die Greedy-Choice-Property, wenn eine lokale Optimierung (die Auswahl der besten Option in einem Schritt) zu einer globalen optimalen Lösung führt.

Def. (Optimal Substructure) Ein Problem hat die optimale Substruktur, wenn eine optimale Lösung des Problems aus optimalen Lösungen seiner Teilprobleme zusammengesetzt werden kann.

Alg. (Huffman Coding) Gegeben eine Menge von Symbolen mit ihren Häufigkeiten, finde eine binäre Präfixcodierung, die die durchschnittliche Anzahl von Bits pro Symbol minimiert. Verwende einen Greedy-Algorithmus, der wiederholt die zwei Symbole mit den geringsten Häufigkeiten kombiniert, bis nur noch ein Symbol übrig ist. Laufzeit: $\Theta(n \log n)$ (für das Einfügen und Entfernen aus der Prioritätswarteschlange (min heap)).

Brute Force Enumeration	Backtracking	Divide and Conquer	Dynamic Programming	Greedy
Recursive Enumerability	Constraint Satisfaction, Partial Validation	Optimal Substructure	Optimal Substructure, Overlapping Subproblems	Optimal Substructure, Greedy Choice Property
DFS, BFS, all Permutations, Tree Traversal	n-Queen, Sudoku, m-Coloring, SAT-Solving, naive TSP	Binary Search, Mergesort, Quicksort, Hanoi Towers, FFT	Bellman Ford, Warshall, Rod-Cutting, LAS, Editing Distance, Knapsack Problem DP	Dijkstra, Kruskal, Huffmann Coding

Parallel Programming

Konzept (Parallelisierbarkeit) Ein Problem ist parallelisierbar, wenn es in Teilprobleme zerlegt werden kann die unabhängig voneinander gelöst werden können, und deren Lösungen dann kombiniert werden können, um die Lösung des ursprünglichen Problems zu erhalten.

```cpp
#include <thread>
void Hello(id){
  std::cout << "Hello from thread " << id << std::endl;
}
int main() {
  std::vector<std::thread> threads;
  for (int i = 0; i < 5; ++i) {
    threads.emplace_back(
      [=]{
        Hello(i);
      });
  }
  for (auto& t: threads) t.join();
}
```

Konzept (Detach vs Fork Join) Threads können geforkt und gejoint werden (oben) oder detached werden, d.h. sie laufen im Hintergrund weiter, auch wenn der Hauptthread bereits beendet ist. `t.detach();`

Satz (Amdahl) Die maximale Beschleunigung S eines Programms durch Parallelisierung ist

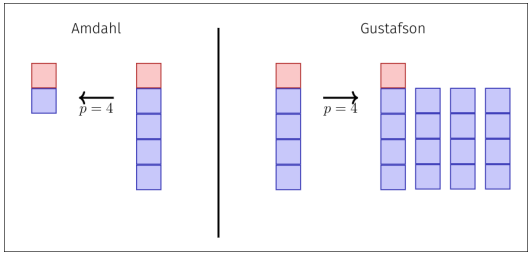
$$S_N = \frac{1}{(1-P) + \frac{P}{N}}.$$

Wobei P der Anteil des Programms ist, der parallelisiert werden kann, und N die Anzahl der Prozessoren ist. Bei unendlich vielen Prozessoren ist $S_\infty = \frac{1}{1-P}$ begrenzt.

Satz (Gustavson) Die maximale Beschleunigung S eines Programms durch Parallelisierung ist

$$S_N = N \cdot (1-P) + P.$$

Wobei P der Anteil des Programms ist, der parallelisiert werden kann, und N die Anzahl der Prozessoren ist. Bei unendlich vielen Prozessoren ist $S_\infty = N$.



Konzept (Asynchronous Threads) Wenn wir Resultate der Threads verwenden wollen, können wir futures brauchen.

```cpp
#include<future>
int Fib(int n){
  if (n <= 1) return n;
  auto f1 = std::async(Fib, n-1);
  auto f2 = std::async(Fib, n-2);
  return f1.get() + f2.get();
}
```

Satz

$$T_N \leq \frac{T_1}{N} + T_\infty.$$

Ausserdem gilt $T_N \geq \frac{T_1}{N}$ und $T_N \geq T_\infty$.

Alg. (Mutex) Ein Mutex (Mutual Exclusion) ist ein Synchronisationsmechanismus, der verwendet wird, um den Zugriff auf eine gemeinsame Ressource in einem parallelen Programm zu steuern. Er stellt sicher, dass nur ein Thread gleichzeitig auf die Ressource zugreifen kann, um Datenkorruption und Inkonsistenzen zu vermeiden. In C++ kann ein Mutex mit der Klasse 'std::mutex' implementiert werden.

```cpp
#include<mutex>
class BankAccount {
  int balance = 0;
  std::recursive_mutex mtx;
public:
  void withdraw(int amount) {
    std::lock_guard<std::mutex> lock(mtx);
    int b = getBalance();
    setBalance(b - amount);
  }
};
```

Konzept (Globale Sortierung) Wenn man zwei Ressourcen gleichzeitig haben will, kann es zu Deadlocks kommen. Um das zu vermeiden, können wir eine globale Sortierung der Ressourcen einführen und verlangen, dass alle Threads die Ressourcen in der gleichen Reihenfolge anfordern.

```cpp
std::mutex mtx1, mtx2;
std::lock(mtx1, mtx2); // Locks both mutexes without risk of deadlock
guard g(m, std::adopt_lock); // RAII wrapper to release locks when
↪  going out of scope
```

Alg. (Circular Buffer) Ein Circular Buffer (Ringpuffer) ist eine Datenstruktur, die als Puffer mit fester Größe fungiert und sich wie ein Ring verhält. Er ermöglicht das effiziente Speichern und Abrufen von Daten in einer FIFO (First-In-First-Out) Reihenfolge. In einem parallelen Kontext kann ein Circular Buffer verwendet werden, um Daten zwischen Produzenten- und Konsumenten-Threads zu übertragen, wobei Synchronisationsmechanismen wie Mutexes oder Semaphoren verwendet werden, um den Zugriff auf den Puffer zu steuern.

```cpp
#include<condition_variable>
class Buffer{
  CircularBuffer buffer;

  std::mutex mtx;

  using guard = std::lock_guard<std::mutex>;
  std::condition_variable cond;
public:
  void put(int x){
    guard g(mtx);
    cond.wait(g, [&]{ return !buffer.full(); });
    cond.notify_all();
    buffer.push(x);
  }
};
```