Problems

Sorting

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

Algorithm	Best	Average	Worst	in pl.	stabl.
Bubble	Ω(n)	Θ(n²)	O(n²)	yes	yes
Selection	Ω(n²)	Θ(n²)	O(n²)	yes	no
Insertion	Ω(n)	Θ(n²)	O(n²)	yes	yes
Merge	Ω(nlogn)	Θ(nlogn)	O(nlogn)	poss.	yes
Heap	Ω(nlogn) Ω(n)	Θ(nlogn)	O(nlogn)	yes	no
Quick	Ω(nlogn)	Θ(nlogn)	O(n²)	yes	no
Radix	Ω(nk)	Θ(nk)	O(nk)		
Bucket	Ω(n+k)	Θ(n+k)	O(n²)		

Bubblesort

1. gehe von links nach rechts durch
2. vergleiche i mit $i+1$
3. falls $i > i+1$ tausche
4. wiederhole bis beim durchgang keine vertauschung mehr nötig

Vergleiche: $\Theta(n^2)$
Vertauschungen: worst case: $\Theta(n^2)$
Worst case: "Umgekehrt" sortiert

Selection Sort

1. kleinstes Elementes durch Suche im unsortierten Teil des Arrays
2. tausche kleinstes Element mit erstem in unsort. Teil
3. Unsortierter Teil wird ein Element kleiner, wiederhole n mal

Vergleiche: worst case: $\Theta(n^2)$
Vertauschungen: wost case $n-1 \in \Theta(n)$

Insertion Sort

1. erstes Element rechts vom sort. Teil an richtige stelle im sort. Teil "schieben"
2. so lange bis sort Teil alle Elemente enthält
3. (sort Teil sollte von links nach rechts in jedem schritt wachsen)

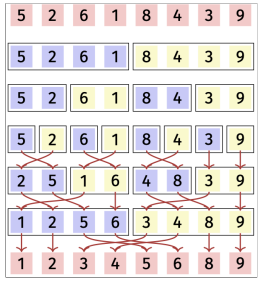
Vergleiche: worst case: $\Theta(n \log n)$
Vertauschungen: worst case $(n-1)^n \in \Theta(n^2)$ (in reverse order)

Shellsort

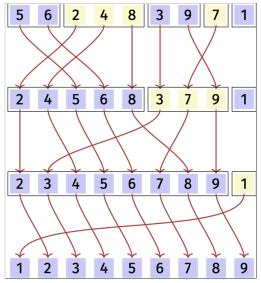
Insertion Sort auf Teilfolgen der Form $(A_{k \cdot i}, i \in \mathbb{N})$ mit absteigenden Abständen k. Letzte Länge ist zwingend $k = 1$. Pratt 1971: $\mathcal{O}(n \log^2 n)$

Mergesort

normal



natural



1. Divide unsorted list into n sublists, each containing one element (trivially sorted)
2. Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining (=sorted list)

Vergleiche: worst case: $\Theta(n \log n)$
Vertauschungen: immer (auch natural) $\Theta(n \log n)$

Natürliches 2-Wege Mergesort

same wie Mergesort, aber man nutzt bestehende sortierte Listen "Runs" aus. Asymptotisch nicht besser als regulär mergesort!

Quicksort

1. Pivotieren!
   (a) von links kommen, stehenbleiben wenn etwas grösser als pivot
   (b) von rechts kommen, stehenbleiben wenn etwas kleiner als pivot
   (c) switch diese
   (d) wenn links und rechts gekreuzt: halt und dort pivot einfügen
   (e) ein Durchgang abgeschlossen; nächste iteration
2. Quicksort auf Subproblem

worst case: Pivotelemente = extrema
Vergleiche: unwahrscheinlich worst case $\Theta(n^2)$ aber im Mittel $\mathcal{O}(n \log n)$
Vertauschungen: worst case $\mathcal{O}(n \log n)$. Speicherplatzbedarf $O(n)$, aber kann vermieden werden: Rekursion nur auf dem kleineren Teil. Dann garantiert $\mathcal{O}(\log n)$ Rekursionstiefe und Speicherplatzbedarf.

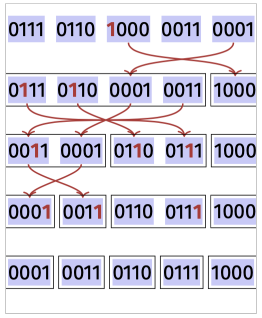
Heapsort

Bucket Sort

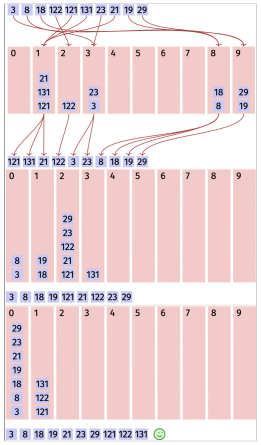
1. Verteilung der Elemente auf die Buckets nach Ziffer i
2. Jeder Bucket wird mit einem weiteren Sortierverfahren (Mergesort) sortiert
3. Der Inhalt der sortierten Buckets wird konkateniert
4. Zifferanzahl-mal wiederholen

Radix vs. Bucket

Radix



Bucket



The initial pass of both is exactly the same. The elements are put in buckets of incremental ranges depending on the number of digits in the largest number. In the next pass, BucketSort orders up these buckets and appends them into one array. RadixSort appends the buckets without further sorting and re-buckets it based on the second digit of the numbers. BucketSort is more efficient for Dense arrays, while RadixSort can handle sparse arrays well.

Selection (+Median)

The k'th element of a sequence of n elements can (worst-case) be found in $\Theta(n)$ steps. Special case $k = n/2$: median.

Ranking

$\mathcal{O}(n^2)$	Repeatedly find minimum
$\mathcal{O}(n \log n)$	Sort, then choose $A[k]$
$\mathcal{O}(n)$ exp.	Quickselect with random piv
$\mathcal{O}(n)$ worst	Median of Medians (Blum)

Search

- The binary sorted search algorithm requires $\Theta(\log n)$ fundamental operations
- Any comparison-based *search* algo on *sorted* data with length n requires in the worst case $\Omega(\log n)$ comparisons
- Any comparison-based *search* algo with unsorted data of length n requires in the worst case $\Omega(n)$ comparisons

Abstract Data Types and Implementations

List

Usage
- Insertion (begin/middle/end to be considered sperately when implementing!)
- traversal in one direction

Implementation
- Linked Lists (insert at begin/middle)
- Arrays (insert at end)

Queue

Usage
- insert (at end)
- traverse in order of insertion
- FIFO

Implementation
- (doubly) linked list

	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

Priority Queue ("Heap")

Usage
- insert elems
- extraction of some highest/lowest elem (for some specified order (="priority"))
- NOT actually SORTED

Implementation
- min/max-Heap

Stack

Usage
- insert at begin
- remove from begin
- LIFO

Implementation
- singly linked list

Dictionary

Usage
- elem insertion and removal
- search of elems

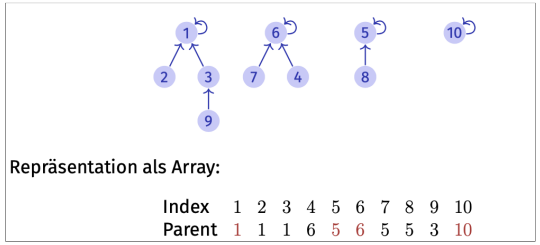
Implementation
- Hash Tables (unordered mapping)
- Balanced Trees (AVL) (ordered mapping)

ADT zur Verwaltung von Schlüsseln aus $\mathcal{K}$ mit Operationen
- `insert`(k, D): Hinzufügen von $k \in \mathcal{K}$ in Wörterbuch D. Bereits vorhanden $\Rightarrow$ Fehlermeldung.
- `delete`(k, D): Löschen von k aus dem Wörterbuch D. Nicht vorhanden $\Rightarrow$ Fehlermeldung.
- `search`(k, D): Liefert `true` wenn $k \in D$, sonst `false`.

Set

Usage
- Representing math sets
- insert
- check if in same set

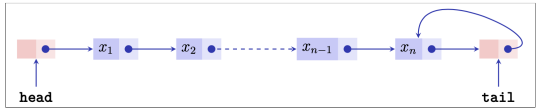
Implementation
- Union-find datastruct
  - `find()` worst case runtime $\mathcal{O}(n)$ optimized ($\mathcal{O}(\log(n))$)
  - all other virtually $\mathcal{O}(\alpha(n))$, α : inv. Ackermann-$f()$



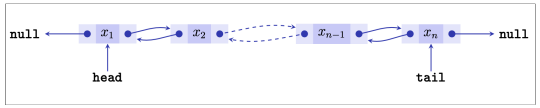
Data structures

Linked Lists

Singly Linked List



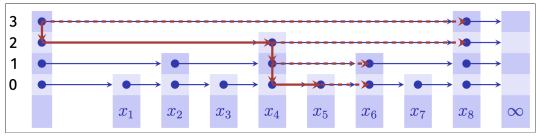
Doubly Linked List



	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

Skiplist



Hash Tables

Hashing

Vocabulary

- **Prehashing**
  $ph(k) \to \mathbb{N}$. i.e. mapping keys onto integers for further use
- **Collision**
  $h(k_i) = h(k_j)\, i \neq j$. i.e. hash function maps two different keys onto same integer
- **Chaining:** Store all $h(k_i) = h(k_j)\, i \neq j$ in one (worst case very long) linked list. Positive: can overcommit (more entries than slots) and easy to remove entries. Negative: Memory consumption of the chains. Alternative: Closed hashing with open addressing
- **Simple Uniform Hashing**
  each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to
- **Uniform Hashing**
  the probing sequence of each key is equally likely to be any of the $m!$ permutations of the possible sequences over the hash table of size m
- **Open Addressing**
  Position in hash table is not fixed and depends on previous entries
- **Closed Hashing**
  Entries stays in table

Probing

m : Table size
k : Prehashed value
j : j'th step in probing sequence starting at $j = 0$ and from the very first hash on!

- **Linear Probing**

$$s(k,j) = h(k) + j$$

  Im Fall einer Kollision einfach linear in die vorgegebene Richtung "wandern" bis man auf einen freien Slot kommt
- **Quadratic Probing**

$$s(k,j) = h(k) + \left\lceil \frac{j}{2} \right\rceil^2 \cdot (-1)^{j+1}$$

  Bei Koll.: jeweils j^2 für jeden Wert j^2 erst in pos. dann in neg. Richtung springen und so weiter bis freier Slot auftacht
- **Double Hashing**

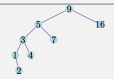
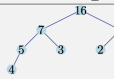
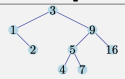
$$s(k,j) = h(k) + j \cdot h'(k)$$

  Basically linear probing, nur halt nicht in 1-Schritten sondern in $h'(k)$-Schritten. **Tipp:** Ganz am Anfang $s(k,j)$ explizit ausschreiben!

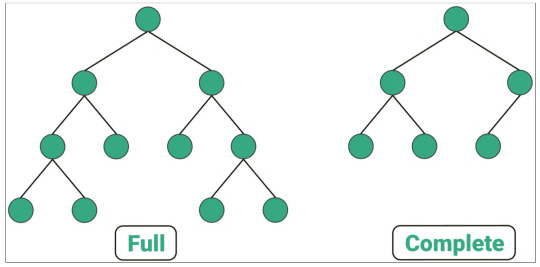
Trees

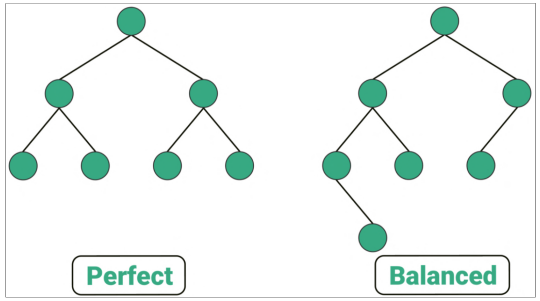
Traversing
- pre-order: n, $n.L$, $n.R$
- in-order: $n.L$, n, $n.R$
- post-oder: $n.L$, $n.R$, n

Overview

	Suchbäume	Heaps Min- / Max-Heap	Balancierte Bäume AVL, red-black tree
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 + $\Theta(\log n)$	$\Theta(\log n)$

Bemerke: $\Theta(\log n) \leq \Theta(h(T)) \leq \Theta(n)$





Binary Search Trees (BST)

Suchbaumeigenschaften

- Jeder Knoten v speichert 1 Schlüssel k
- Schlüssel im linken Teilbaum $v.left$ kleiner als $v.key$
- Schlüssel im rechten Teilbaum $v.right$ grösser als $v.key$

Suchen von k

1. falls $v.key = k$: gefunden!
2. falls $v.key < k$: $v.key \leftarrow v.l$
3. sonst $v.key \leftarrow v.r$
4. repeat

Einfügen von k

1. suchen(k)
2. falls gef.: bereits drin
3. falls nicht: dort inserten

k entfernen ($\mathcal{O}(h(T))$)

1. k **keine kinder:** k durch Blatt ersetzen
2. k **ein kind:** k durch Kind ersetzen
3. k **zwei kinder:** k duch grössten in k.l oder kleinsten in k.r ersetzen ("symmetrischer Nachfolger")

(Max-)Heaps

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

	1-indexed	0-indexed
child(i)	$= \{2i, 2i+1\}$	$= \{2i+1, 2i+$
parent(i)	$= \{\lfloor i/2 \rfloor\}$	$= \{\lfloor (i-1)/2 \rfloor\}$
$H(n)$	$= \lceil \log_2(n+1) \rceil$	
`heapify()`	$\in \mathcal{O}(\log n)$ redoes heap prop.	
`build_H()`	$\in \mathcal{O}(n)$[sic!] array into heap	
`insert(v)`	$\in \mathcal{O}(\log n)$ insert at last pos. redo heap prop. by climbing	
`pop_max()`	$\in \mathcal{O}(\log n)$ switch w/last pos. redo heap prop. by sinking /switching w/ ggf. bigr child	

(i-th level of a bin. tree has max. 2^i nodes)

Huffmann Trees

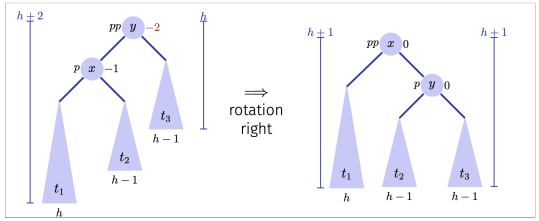
Replace iteratively the two nodes with the smallest frequencies, with one parent node with the frequency of the sum of its children's frequencies. **Do it on spare paper first!**

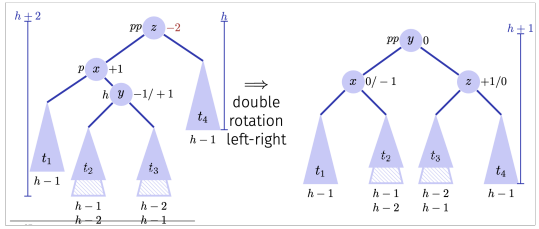
AVL Trees

AVL-Bedingung: $\forall v$ eines Baumes gilt:

$$bal(v) := h(T_r(v)) - h(T_l(v)) \in \{-1, 0, 1\}$$

Worst-case runtimes of $\mathcal{O}(\log n)$ for searching, insertion *and* deletion of keys.





(Minimum) Spanning Trees

- kann mit greedy Algos gefunden werden (Kruskal, oder Jarnik/Prim/Dijkstra)
- (not unique) tree

Beweise

Induktionsbeweise

1. Hypothesis: Definiere $f(n)$
2. Base Case: Beweise $f(1) = T(1)$
3. Step: $Hyp \implies T(2^{k+1}) = f(2^{k+1})$ Forme $T(2^{k+1})$ mit Hyp um, damit $= f(2^{k+1})$ rauskommt

Beweise mit Invarianten

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    if (b % 2 != 0){
      res += a;
      --b;
    }
    a *= 2;
    b /= 2;
  }
  return res;
}
```

let $x := a \cdot b$.

here: $x = \boxed{a \cdot b + res}$

if here $x = a \cdot b + res$ ...

... then also here $x = a \cdot b + res$ b even

here: $x = a \cdot b + res$

here: $x = a \cdot b + res$ and $b = 0$

therefore $res = x$.

Asymptotics

$\mathcal{O}$-Reihenfolge bestimmen

$\mathcal{O}$-Mächtigkeiten

$c, \log\log n, \log^c n, \sqrt{n}, n, n\log n, n^c, c^n, n!, n^n$
$\binom{n}{k} = \frac{n!}{k!\cdot(n-k)!} \in \Theta(n^k)$ $\log(n!) \in \Theta(n\log n)$ $n! \in \mathcal{O}(n^n)$

Summen

$\sum_{i=0}^{n} c^i$	$= \frac{cb^{n+1}-1}{c-1}$	$\in \mathcal{O}(1)$
$\sum_{i=0}^{n} i$	$= \frac{n\cdot(n+1)}{2}$	$\in \mathcal{O}(n^2)$
$\sum_{i=0}^{n} i$	$= \sum_{i=0}^{n}(n-i)$	$\in \mathcal{O}(n^2)$
$\sum_{i=0}^{n} i^2$	$= \frac{n(n+1)(2n+2)}{6}$	$\in \mathcal{O}(n^3)$
$\sum_{i=0}^{n} i^k$		$\in \mathcal{O}(n^{k+1})$
$\binom{n}{k}$	$= \frac{n!}{k!\cdot(n-k)!}$	$\in \mathcal{O}(n^k)$
$\sum_{i=0}^{n^2} i$		$\in \mathcal{O}(n^4)$

Logarithmen

$x = \sqrt[n]{a}$	$\Leftrightarrow x^n = a$
$\log_b 1$	$= 0$
$\log_b b$	$= 1$
$\log_b(a\cdot c)$	$= \log_b a + \log_b c$
$\log_b\left(\frac{a}{c}\right)$	$= \log_b a - \log_b c$
$\log_b(a^c)$	$= c\cdot\log_b a$
$\log_b a$	$= \frac{\log_c a}{\log_c b}$

Landaunotation

Standarddefinition

$$\mathcal{O}(g(n)) = \{\exists c>0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq f(n) \leq c\cdot g(n)\}$$
$$\Omega(g(n)) = \{\exists c>0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq c\cdot g(n) \leq f(n)\}$$
$$\Theta(g(n)) = \{\exists c>0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq \frac{1}{c}\cdot g(n) \leq f(n) \leq c\cdot g(n)\}$$
$$\mathcal{O}(g(n)) \cap \Omega(g(n))$$

Grenzwerte

$$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \begin{cases} 0 & \Longrightarrow f \in \mathcal{O}(g) \\ c & \Longrightarrow f \in \Theta(g) \\ \infty & \Longrightarrow f \in \Omega(g) \end{cases}$$

Rekursionsgleichungen

Mastertheorem

$$T(n) = a\cdot T\left(\frac{n}{b}\right) + f(n), \quad (a \geq 1, b > 1)$$

a	: Anzahl Unterprobleme
$1/b$	: Aufteilungsquotient
$f(n)$	: Div.- und Summ.kosten

1. Forme Rek.gleichung ins Schema um
2. Berechne $K := \log_b a$
3. Fallunterscheidung machen ($\varepsilon > 0$):

$$f \in \begin{cases} \mathcal{O}\left(n^{K-\varepsilon}\right) & \Longrightarrow T(n) \in \Theta\left(n^K\right) \\ \Theta\left(n^K\right) & \Longrightarrow T(n) \in \Theta\left(n^K\log(n)\right) \\ \Omega\left(n^{K+\varepsilon}\right) & \wedge af\left(\frac{n}{b}\right) \leq cf(n),\ 0<c<1 \\ & \Longrightarrow T(n) \in \Theta(f(n)) \end{cases}$$

Plug and Chuck

- Expand few times
- Notice patterns (careful with multiplications on of $T(n)$)
- Write down explicitly
- Formulate explicit formula $f(n)$
- Prove via induction (starting at $f(1)$)

Aufrufe von `f()`

1. Falls loops: einfach mal rechnen
2. Falls schwer: usually $\Theta(2^n)$
3. Auf Notizpapier einfach stur $g(0), g(1), g(2), g(3), \ldots$ rechnen, dabei Rekursion nutzen und Trend erkennen
4. **Notfalls einfach Rekglg aufstellen und lösen** (siehe Code wie)

Code, damit `f()` $T(n)$-mal aufgerufen

$$T(n) = \begin{cases} 2T(n/4) + \log_4(n), & n > 1 \\ 0, & n \leq 1 \end{cases}$$

```
if (n > 1){
   g(n/4); g(n/4);    // 2T(n\4)
   while(n > 1){      //  0, n <= 1
      f();
      n /= 4;         //  log_4(n)
   }
}
```

$$T(n) = \begin{cases} 1 + T(n/4), & n > 1 \\ 2, & n = 1 \end{cases}$$

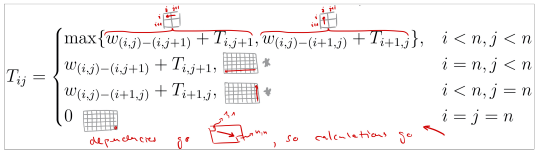
```
if (n == 1){
   f(); f();       // 2 falls n == 1
}else){
   f();
   g(n/4);         // T(n/4)
}
```

Dynamic Programming

1. **Table with info on subproblems**
   Dimension of the table?
   What are the entries "mathematically"?
2. **Computation of base cases**
   Which entries do not depend on others?
3. **Determine computation order**
   In which order can the entries be computed?
4. **Read-out the solution**
   How can the solution be read out?

$$\text{Runtime} = \text{Entries}\cdot\frac{\text{Operations}}{\text{Entry}}$$

$$T_{ij} = \begin{cases} \max\{w_{(i,j)-(i,j+1)} + T_{i,j+1}, w_{(i,j)-(i+1,j)} + T_{i+1,j}\}, & i<n, j<n \\ w_{(i,j)-(i,j+1)} + T_{i,j+1}, & i=n, j<n \\ w_{(i,j)-(i+1,j)} + T_{i+1,j}, & i<n, j=n \\ 0 & i=j=n \end{cases}$$



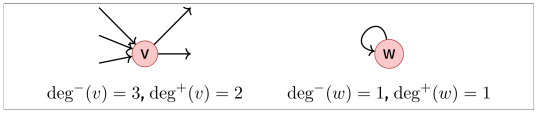
Knapsack

```
int max(int a, int b){
   return (a > b) ? a : b;
}
// Returns the max value that can
// be put in knapsack of capacity W
int knapSack(int W, int wt[], int
    val[], int n){
   int i, w;
   vector<vector<int>> K(n + 1,
       vector<int>(W + 1));
   // Build table K[][] in bottom up
       manner
   for(i = 0; i <= n; i++){
      for(w = 0; w <= W; w++){
         if (i == 0 || w == 0){
            K[i][w] = 0;
         }else if (wt[i - 1] <= w){
            K[i][w] = max(val[i - 1] +
                      K[i - 1][w -
                          wt[i - 1]],
                      K[i - 1][w]);
         } else {
            K[i][w] = K[i - 1][w];
         }
      }
   }
   return K[n][W];
}
int main(){
   int val[] = { 60, 100, 120 };
   int wt[] = { 10, 20, 30 };
   int W = 50;
   int n = sizeof(val) /
       sizeof(val[0]);
   cout << knapSack(W, wt, val, n);
   return 0;
}
```

Graphs

Notation/Representation

Degrees



For each graph $G(V, E)$ it holds that

- for directed:

$$\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|$$

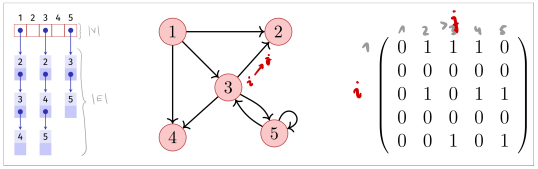
- for *un*directed Graphs:

$$\sum_{v \in V} \deg(v) = 2|E|$$

Simple Observations

- Generally: $0 \leq |E| \in \mathcal{O}(|V|^2)$
- Connected G: $|E| = \Omega(|V|)$
- Complete G: $|E| = \frac{|V| \cdot (|V|-1)}{2}$
- Maximally:
  - $|E| = |V|^2$ (directed)
  - $|E| = \frac{|V| \cdot (|V|+1)}{2}$ (undirected)
- Also: undirected graphs cannot contain cycles with length 2 (loops have length 1)

Matrix vs. List



Operation	Matrix	List
Find neighbours/successors of $v \in V$	$\Theta(n)$	$\Theta(\deg^+ v)$
find $v \in V$ without neighbour/successor	$\Theta(n^2)$	$\Theta(n)$
$(v, u) \in E$?	$\Theta(1)$	$\Theta(\deg^+ v)$
Insert edge	$\Theta(1)$	$\Theta(1)$
Delete edge (v, u)	$\Theta(1)$	$\Theta(\deg^+ v)$

Directed Acyclic Graphs (DAG)

a directed graph with *no* directed cycles. A directed graph is a DAG $\iff$ it can be topologically ordered, by arranging the vertices as a linear ordering that is consistent with all edge directions. ($\rightarrow$ *TopologicalSort*)

Algorithms on Graphs

Depth First Search (DFS)

- (nützlich um Zyklen zu finden)
- **Laufzeit:** $\Theta(|V| + |E|)$
- verfolgt einen Pfad in die Tiefe, bis kein neuer Knoten mehr besucht werden kann
- dann geht zum nächsten Nachbar des Startknotens
- sind alle Pfade, die bei einem Nachbarn beginnen, durchlaufen worden, wird ein neuer Startknoten aus der Menge der unbesuchten Knoten gewählt

Breadth First Search (DFS)

- **Laufzeit:** $\Theta(|V| + |E|)$
- Lösung für kürzester-Weg-Problem wenn Kantengewichte alle uniform

Manual Execution

1. Draw ring around starting vertex s
2. Set dist$(s) = 0$
3. while($\exists$ Neighbours) do
   (a) Draw ring around all neighbouring vertices from current set
   (b) Draw tree of visited nodes (useful for Qs)
   (c) Move on to next iteration with newly drawn ring/set

Topological Sort

Gegeben sei ein gerichteter, kreisfreier Graph. Mit top.Sort kann man in $\Theta(|V| + |E|)$ Zyklen detektieren.

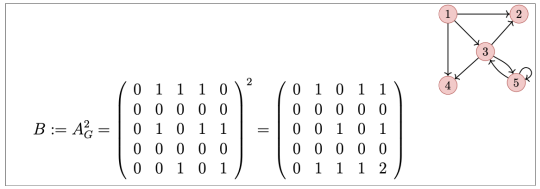
Manual Execution

1. pick unvisited node
2. begin with the unvisited node and do a DFS exploring only unvisited nodes until you hit a dead end
3. now just add the nodes you go through on the way back in the order you go through them

Warshall's Algo for Transitive Closure

Transitive Closure: Smallest extension of a R such that $R*$ is *reflexive* and *transitive*. For Graphs that is basically a *reachability*-relation ($\exists$path from v to w?)

1. setup adj. matrix B
2. set all $b_{i,i} = 1$
3. for $k : 1 \rightarrow n$
   - for $r : 1 \rightarrow n$
     - on r'th row:
       * if($r = k$): skip row
       * if($b_{r,k} = 0$): skip row
       * add all 1s from k'th row



Finding MSTs

Kruskal's MST Algo

- $\Theta(|E| \log |V|)$
- nutzt ADT "Union-Find"

Manual Execution

1. Sort edges by weight (min-Heap!)
2. walk through the sorted edges and look at the two nodes that the edge is between
   - if nodes are already in same set (union-find), don't include that edge
   - otherwise we do and now unify those nodes
3. algo terminates when all nodes are unified (i.e. in same set)

Jarnik/Prim/Dijkstra's MST Algo

- **Use this one for manual execution!**
- $\Theta(|E| \cdot |V|)$ (Fibonacci-heap: $\Theta(|E| + |V| \log |V|)$)
- better than Kruskal on dense graphs (= many edges)
- ähnlich zu Dijkstra's shortest path algo

Manual Execution

1. start with some node as your inner set
2. keep add cheapest edge that is going out from current set until entire graph is in inner set

Shortest Path Problem

Overview/Rankings

Shortest Path (SP) Algorithms

	BFS	Dijkstra's	Bellman Ford	Floyd Warshall
Complexity	O(V+E)	O((V+E)logV)	O(VE)	O(V^3)
Recommended graph size	Large	Large/ Medium	Medium/ Small	Small
Good for APSP?	Only works on unweighted graphs	Ok	Bad	Yes
Can detect negative cycles?	No	No	Yes	Yes
SP on graph with weighted edges	Incorrect SP answer	Best algorithm	Works	Bad in general
SP on graph with unweighted edges	Best algorithm	Ok	Bad	Bad in general

Johnson (Solves APSP)
$\mathcal{O}(|V|^2 \log(|V| + |V||E|))$ (with FibHeaps in Dijkstra)
A*
$\mathcal{O}(|E|) = \mathcal{O}(b^d)$

Observations

1. negative cycle $\implies$ no shortest path
2. there can be exp. many paths (trying all is futile)
3. triangle inequality holds
4. subpaths of shortest paths are shortest paths too (optimal substructure)
5. shortest paths don't contain cycles

Shortest Path Algorithms

General Algorithm

1. Initialise d_s and π_s: $d_s[v] = \infty$, $\pi_s[v] = \text{null}$ for each $v \in V$
2. Set $d_s[s] \leftarrow 0$
3. Choose an edge $(u, v) \in E$
   Relaxiere (u, v):
   if $d_s[v] > d_s[u] + c(u, v)$ then
   $d_s[v] \leftarrow d_s[u] + c(u, v)$
   $\pi_s[v] \leftarrow u$
4. Repeat 3 until nothing can be relaxed any more.
   (until $d_s[v] \leq d_s[u] + c(u, v) \quad \forall (u, v) \in E$)

Dijkstra

- Greedy
- Worst-case $\mathcal{O}(|E| + |V| \log |V|)$ (mit Fibheaps)
- Worst-case $\mathcal{O}((|E|+|V|) \cdot \log |V|)$ (ohne Fibheaps)
- funzt nur mit pos. Kantengewichten
- Grundidee: 3 Mengen (1: kürz. Weg bekannt, 2: direkt anliegend an 1., und 3: noch im Fog of War)

Manual Execution

1. Give all vertieces $\text{dist}_{v_i} = \infty$
2. Give starting vertex $\text{dist}_s = 0$
3. while(t (= endvertex) is unmarked) do
   (a) chose vertex with smallest dist and mark it
   (b) relax all reachable vertices

Bellman-Ford

- $\mathcal{O}(|V| \cdot |E|)$ with fibheaps
- based on Dynamic Programming (not greedy!)
- used for **neg-cycle detection** (Arbitrage!)
- funktioniert auch bei negatievn Kantengewichten **aber nicht** bei negativen Zyklen!
- nach $|V|$ Iterationen können noch Relaxierungen gemacht werden $\implies$ negative Zyklen im Graph (da dass einen shortest path von Länge $|V|$ impliziert, was nicht sein kann)

Manual Execution

foreach $u \in V$ **do**
  $d_s[u] \leftarrow \infty$; $\pi_s[u] \leftarrow$ **null** // Dyn Proj!
$d_s[s] \leftarrow 0$;
for $i \leftarrow 1$ **to** $|V|$ **do**
  $f \leftarrow$ false
  foreach $(u, v) \in E$ **do**
    $f \leftarrow f \vee \text{Relax}(u, v)$
  if $f =$ false **then return** true // in case of already max. relaxed
return false; { if relaxation was possible in |V|'th Iteration, this would imply a negativ cycle

Floyd-Warschall

- löst all-pairs-shortest path, gibt aber nur path lengths an, nicht paths (paths können aber konstruirt werden)
- **funktioniert bei negativen Kanten** aber keine negativen Zyklen!
- best implemented with Adjazenzmatrizen
- can be executed in single matrix (in place)
- also based on Dynamic Programming
- pain in the ass to do by hand since it's $\mathcal{O}(|V|^3)$

Input: Graph $G = (V, E, c)$ without negative weight cycles.
Output: Minimal weights of all paths 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**
      $d^k(v_i, v_j) = \min\{d^{k-1}(v_i, v_j), d^{k-1}(v_i, v_k) + d^{k-1}(v_k, v_j)\}$

Johnson

Runs Bellman-Ford when *reweighting* once and then $|V|$-times Dijsktra on the *reweighted* graph $\tilde{G}$ for shortest path between all pairs of vertices

Reweighting Process, that basically reassigns all path weights c_i a new $\tilde{c}_i$ which is *non-negative*. The shortest path of any graph on which this is done **remains the same**. This is done by:

- add a new vertex s which has paths of cost $= 0$ to all other vertices
- calculate the shortest paths from s to each of the $v \in V$; call it $d(s, v)$ (with the Bellman-Ford Algo)
- reassign all $c(u, v)$ like this:

$$h(v) := d(s, v)$$
$$\tilde{c}(u, v) := c(u, v) + h(u) - h(v) \geq 0$$

A*

Basically Dijstra but with a Heuristik (usually Manhatten-distance) (that *underestimates* the actually distance it's goind to take)

Input: Positively weighted Graph $G = (V, E, c)$, starting point $s \in V$, end point $t \in V$, estimate $\hat{h}(v) \leq \delta(v, t)$
Output: Existence and value of a shortest path from s to t
foreach $u \in V$ **do**
  $d[u] \leftarrow \infty$; $\hat{f}[u] \leftarrow \infty$; $\pi[u] \leftarrow$ null
$d[s] \leftarrow 0$; $\hat{f}[s] \leftarrow \hat{h}(s)$; $R \leftarrow \{s\}$; $M \leftarrow \{\}$
while $R \neq \emptyset$ **do**
  $u \leftarrow \text{ExtractMin}_{\hat{f}}(R)$; $M \leftarrow M \cup \{u\}$
  if $u = t$ **then return** success
  foreach $v \in N^+(u)$ with $d[v] > d[u] + c(u, v)$ **do**
    $d[v] \leftarrow d[u] + c(u, v)$; $\hat{f}[v] \leftarrow d[v] + \hat{h}(v)$; $\pi[v] \leftarrow u$
    $R \leftarrow R \cup \{v\}$; $M \leftarrow M - \{v\}$
return failure

Flow Networks G

Flow and Cuts

Flow

A flow $f : V \times V \to \mathbb{R}$ fulfills:

Kapazitätsbeschr:	$f(u,v) \leq c(v,u)$
Symmetrie:	$f(u,v) = -f(v,u)$
Flusserhaltung:	$\sum_{v \in V} f(u,v) = 0$ $\uparrow \forall u \in V \setminus s,t$
Value:	$\lvert f \rvert = \sum_{v \in V} f(s,v)$

- Flow network $G = (V, E, c)$: directed graph with capacities
- Antiparallel edges forbidden: $(u,v) \in E \Rightarrow (v,u) \notin E$.
- Model a missing edge (u,v) by $c(u,v) = 0$.
- Source s and sink t: special nodes. Every node v is on a path between s and t: $s \rightsquigarrow v \rightsquigarrow t$

Cuts

- Cuts are partitions of V into S and T with $s \in S$ and $t \in T$
- Capacity of a cut:

$$c(S,T) = \sum_{v \in S, v' \in T} c(v,v')$$

  i.e. Summe der c von den e die gecuttet werden
- minimal cut: cut with minimal capacity
- Flow over a cut:

$$f(S,T) = \sum_{v \in S, v' \in T} f(v,v')$$

  i.e. Summe der f von den e die gecuttet werden

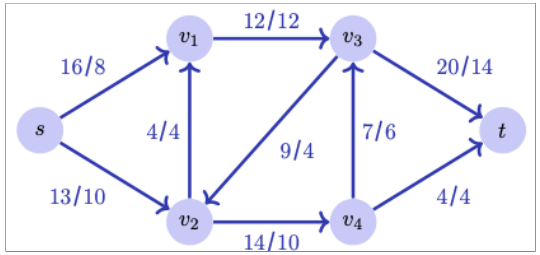
Max-Flow Min-Cut Theorem

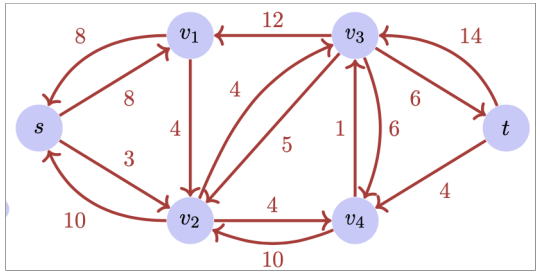
Let f be a flow in a flow network $G = (V, E, c)$ with source s and sink t. The following statementsa are equivalent:

- f is a maximal flow in G
- The rest network G_f does not provide any expansion paths
- It holds that $\lvert f \rvert = c(S,T)$ for a cut (S,T) of G

Rest Network G_f

Definition (c/f)





Rest network G_f provided by the **edges with positive *rest* capacity**. Weg in Richtung s einfach c von G aber Weg in Richtung t jeweils Restkapazität $c - f$
Rest networks provide the same kind of properties as flow networks with the exception of **permitting antiparallel capacity-edges**

Theorem 31:

Let $G = (V, E, c)$ be a flow network with source s and sink t and f a flow in G. Let G_f be the corresponding **rest networks** and let f' be a flow in G_f. Then $f \oplus f'$ with

$$(f \oplus f')(u,v) = f(u,v) + f'(u,v)$$

defines a flow in G with value $\lvert f \rvert + \lvert f' \rvert$.

Augmenting Paths

expansion path p: simple path from s to t in the rest network G_f
Rest capacity:

$$c_f(p) = \min\{c_f(u,v),\ (u,v) = e \in p\}$$

(smallest capacity along path s in G_f)

Flow in G_f

Theorem 32: the map $f_p : V \times V \to \mathbb{R}$,

$$f_p(u,v) = \begin{cases} c_f(p), & (u,v) \in p \\ -c_f(p), & (v,u) \in p \\ 0, & \text{otherwise} \end{cases}$$

provides a flow in G_f with value $\lvert f_p \rvert = c_f(p) > 0$

Strategy for an Algoritm

With an expansion path p in G_f the flow $f \oplus f_p$ defines a new flow with value

$$\lvert f \oplus f_p \rvert = \lvert f \rvert + \lvert f_p \rvert > \lvert f \rvert$$

Maximal Flow Algorithms

Ford-Fulkerson

- gibt den maximalen Fluss eines Netzes zurück
- anfangs haben alle Kanten den Fluss $f(u,v) = 0$, dann bestimmt man das Restnetzwerk G_f und dadurch einen Erweiterungspfad p der vorgibt, um wieviel man den Fluss erhöhen kann
- might not terminate for $c \in \mathbb{R}$ but always will in $\leq \lvert f_max \rvert$ (since always increases by at least 1) steps for $c \in \mathbb{Z}$
- Laufzeit beträgt $\mathcal{O}(f_{max} \cdot \lvert E \rvert)$

Manual Execution

- start with $f(u,v) = 0,\ \forall v \in V$
- determine rest network G_f and find expension path p in it
- increase flow along p by the smallest capacity c that lies along p
  (careful with the direction of flow, sometimes one must decrease it)
- repeat until no more expansion paths p can be found in G_f

```
for (u,v) ∈ E do
    f(u,v) ← 0
while Exists path p : s ⇝ t in rest network G_f do
    c_f(p) ← min{c_f(u,v) : (u,v) ∈ p}   // smallest c_f along the path p in G_f
    foreach (u,v) ∈ p do
        if (u,v) ∈ E then
            f(u,v) ← f(u,v) + c_f(p)      f(u,v) += c_f(p)
        else
            f(v,u) ← f(v,u) − c_f(p)
```

Edmonds-Karp

- quasi **Ford-Fulkerson**, nur wird bei der Wahl des Erweiterungspfades s *der kürzeste* ausgewählt (z.B. durch BFS)
- $\mathcal{O}(\lvert V \rvert \cdot \lvert E \rvert^2)$ (if $c \in \mathbb{Z}$)
- Number of flow increases $\in \mathcal{O}(\lvert V \rvert \cdot \lvert E \rvert^2)$ (Theorem 34)

Push-Relabel

- has a relaxed flow condition at the beginning and then basically works towards restoring it but in the mean time works with *preflow*
- the residual network G_f therefore works with *preflow* and not with flow itself
- Terminates after $\mathcal{O}(n^2)$ relabels and $\mathcal{O}(n^3)$ pushes
- Runtime $\mathcal{O}(V^2\sqrt{E})$

while $\exists u \in V \setminus \{s,t\} : \alpha_f(u) > 0$ **do**
  choose u with $\alpha_f(u) > 0$ and maximal $h(u)$
  if $\exists v \in V : c_f(u,v) > 0 \ \wedge \ h(v) = h(u) - 1$ **then**
    push(u,v)
  else
    $h(u) \leftarrow h(u) + 1$

if $\alpha_f(u) > 0$ **then**
  if $c_f(u,v) > 0$ in G_f **then**
    $\Delta \leftarrow \min\{c_f(u,v), \alpha_f(u)\}$
    $f(u,v) \leftarrow f(u,v) + \Delta$.

Ford-Fulkerson (conservative)
- Invariant: flow conservation
- Steps: augmenting paths
- Goal: separate s from t in the residual network.

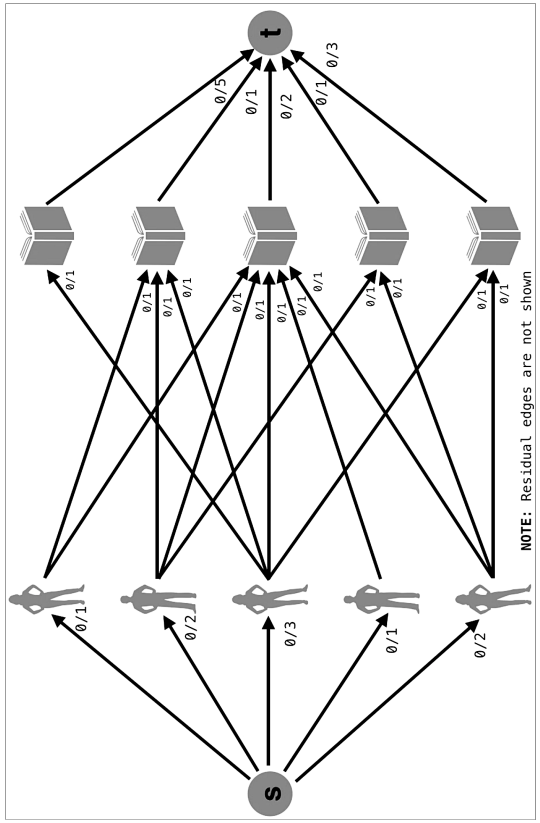
Push-Relabel
- Invariant: height invariant (no augmenting path!)
- Steps: push flow
- Goal: achieve flow conservation

Max Bipartite Matching

- Greedy approach doesn't work. Use maxflow algos (Theorem 35: Ford-Fulkerson) to solve it
- Problems of this kind usually revolve around finding matches between e.g. people and jobs, readers and books, etc.

Example: Readers to books

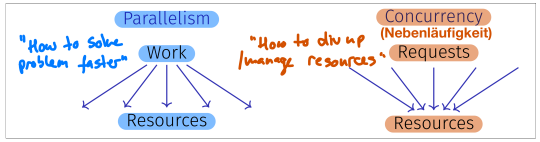
- **Capacity c_s from s to people**
  c books that person can take
- **Capacity c_t from books to sink t**
  how many times each book can be taken (by all readers combined)
- **Edge from a reader to a book**
  which books each reader wants (and how many times, even though this doens't make sense in this example)



Node capacity implementation

Let v be a vertex of G with capacity $c(v)$. Replace G with two vertices v' and v'' and connect them with an edge from v' to v'' with capacity $c(v)$. Replace any incoming edge (u,v) by the edge (u,v') with the same capacity. Further, replace any outgoing edge (v,w) by the edge (v'',w). By doing this for all vertices of G, we obtain the graph G and its capacity function c'. A flow in G' can be mapped to a valid flow in G, as the sum over the incoming flow for v' is the same as the sum of the outgoing flow for v''. Further, this sum never exceeds $c(v)$.

Parallel Programming



Parallelism: use extra resources to solve problem faster
Concurrency: How to div up/manage resources
Thread Safe: concurrent program yields expected results (esp. regarding order)
Nondeterministic Execution: Sometimes one thread finished before the other and prints out its result in a seemingly wrong order

Amdahl vs. Gustafson

Amdahl:
fixed **relative** sequential portion
Gustafson:
fixed **absolute** sequential part

Amdahl & Gustafson

Computing Work	W
Sequential Execution Time	T_1
Number of CPUs	p
Performance loss	$T_p > \frac{T_1}{p}$ (reality)
Perfection	$T_p = \frac{T_1}{p}$
Magic	$T_p < \frac{T_1}{p}$
Paralellisable part	W_p
Sequential part	W_s
Speedup	S_p
Efficiency	$E_p = \frac{S_p}{p}$
non-paralellisable fraction	λ

$$W_s = \lambda \cdot W$$
$$W_p = (1-\lambda) \cdot W$$
$$T_1 = W_s + W_p$$
$$T_p \geq W_s + \frac{W_p}{p}$$

Amdahl's Law

$$S_p = \frac{T_1}{T_p} \leq \frac{W_s + W_p}{W_s + \frac{W_p}{p}} = \frac{1}{\lambda + \frac{1-\lambda}{p}}$$

$$\Longrightarrow S_\infty \leq \tfrac{1}{\lambda}$$

Gustafson's Law

Work that can be executed by 1 processor in time T: $W_s + W_p = T$
Work that can be executed by p processors in time T:

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

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

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

Performance Model

T_∞: Critical path, execution time on ∞ processors. Longest path from root to sink. **exam relevant:** sum the node values for actual "length" of the path. Longest path gives you T_∞.
$\frac{T_1}{T_\infty}$: Parallelism. Wider = better.
Work Law: $T_p \geq \frac{T_1}{p}$
Span Law: $T_p \geq T_\infty$

Greedy Scheduler

"Teilt zu jeder Zeit so viele Tasks zu Prozessoren wie möglich"
Theorem 45: On an ideal parallel computer with p processors, a greey scheduler executes a multi-threaded computatioon with work T_1 and span T_∞ in time:

$$T_p \leq \frac{T_1}{p} + T_\infty$$

Race Conditions

Race Conditions occur, when the result of a computation depends on scheduling.

Data race:

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:

Fehlerhaftes Programmverhalten verursacht durch eine unglückliche Ausführungsreihenfolge eines Algorithmus mit mehreren Threads, selbst dann wenn die gemeinsam genutzten Resourcen anderweitig gut synchronisiert sind.

Rule of thumb:

Compiler and hardware are allowed to make changes that do not effect the semantics of a **sequentially** executed program. They insbesondere don't guarantee that global ordering of memory access is provided as in the sourcecode (= fucks up the order of commands).
C++ provides guarantees when synchronisation with a `mutex` is used.

Atomic

Basically guarantees that reading and writing happens in one step, leaving no time for race conditions.

```
template <atomic>
std::atomic_int x{0};
std::atomic_int y{0};
```

Advanced C++

Memory Allocation

Rule of Five

If one of them is tinkered with, all of them should be:

- destructor
- copy-constructor
- copy-assignment operator
- move constructor
- move assignment operator

Copy-&-Swap Idiom

all members of `*this` are exchanged with members of `cpy`. When leaving `operator=`, `cpy` is cleaned up (deconstructed), while the copy of the data of `v` stay in `*this`.

```
class Vector{
   // move constructor
   Vector (Vector&& v) : Vector() {
      swap(v);}
   // normal assignment
   Vector& operator= (const Vector&
       v){
      Vector cpy;
      swap(cpy);
      return *this;}
   // move assignment
   Vector (Vector&& v){
      swap(v);
      return *this;}
private:
   void swap(Vector& v){
      std::swap(sz, v.sz);
      std::swap(elem, v.elem);}
};
```

Templates

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

template <typename T> \\ templated
    printer
void output(const T& t){
    for (auto x: t)   \\ ranged for
        loop
       std::cout << x << " ";
    std::cout << "\n";
}
```

Iterators

```
// for support of ranged for-loops
class Vector{

   // Iterator (const in front for
       const-it)
   double* begin(){
      return elem;
   }

   double* end(){
      return elem+sz;
   }
}
```

Dynamic Memory Guidelines

For each `new` ; a matching `delete` !
Avoid:

- **Memory Leaks:** Old object not freed/hog memory
- **Dangling Pointers:** Pointer to released object
- **Double Free:** Releasing memory more than once using `delete`
- i.e. use Smart Pointers!

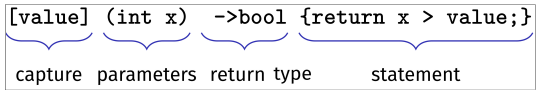
Smart Pointers

```
std::unique_ptr<Node> nodeU(new
    Node());
std::shared_ptr<Node> nodeS(new
    Node());
```

- Never call delete on a pointer contained in a smart pointer
- Where possible, use `std::unique_ptr`
- If using `std::shared_ptr` make sure there are no cycles in the pointer graph.
- Avoid new, instead:

```
std::unique_ptr<Node> nodeU =
    std::make_unique<Node>()
std::shared_ptr<Node> nodeS =
    std::make_shared<Node>()
```

Lambdas & Functors



Functors are just classes with an overloaded `operator()`, so that when you use that operator it looks like you're calling a function (which you do, more specifically a member function). Functors can store member variables in between calls ("states").

```
// this is a functor
class LessThan{
   T value;
public:
   LessThan(T x) : value(x) {} //
       constructor

   bool operator() (T otherval) const{
      return (otherval < value);
   }
};

// this is a lambda expression
auto f = [value](int x){return x <
    value;};
// [capture](parameters)->
    optional_returntype{the actual
    code;};

// things that go inside a capture:
// [x]: pass x by value
// [&x]:pass x by reference
// [&]: pass everything by ref.
// [=]: pass everything by value
// [&x,y]:pass x by ref. and y by
    value
```

Snippets

Bridge

```
class Bridge {
private:

unsigned int capacity = 0; // assert(capacity <= 3)

std::mutex m;
using guard = std::unique_lock<std::mutex>;
std::condition_variable cond;

public:

// make sure that only three cars or one truck can
    enter!
void enter_car(){

  std::cout << "Car trying to enter the bridge" <<
      std::endl;

  guard g(m);         // lock m
  cond.wait(g, [&]{return (capacity < 3);});//
      wait until cap < 3

  std::cout << "Car ENTERING the bridge" <<
      std::endl;

  ++capacity;         // car is now on bridge
}

// make sure that waiting cars or even trucks can
    enter
void leave_car(){
  guard g(m);         // lock m
  --capacity;         // car is now on bridge
  cond.notify_all(); // tell waiting vehicles
                      // they can (maybe) go
  std::cout << "Car LEAVING the bridge. capacity =
      " << capacity << std::endl;
}

// make sure that only three cars or one truck can
    enter!
void enter_truck(){

  std::cout << "Truck trying to enter the bridge"
      << std::endl;

  guard g(m);         // lock m
  cond.wait(g, [&]{return (capacity == 0);}); //
      wait until cap == 0

  std::cout << "Truck ENTERING the bridge" <<
      std::endl;

  capacity += 3;      // truck is now on bridge
}

// make sure that waiting cars or even trucks can
    enter
void leave_truck(){
  guard g(m);         // lock m
  capacity -= 3;      // car is now on bridge
  cond.notify_all();  // tell waiting vehicles
                      // they can (maybe) go
  std::cout << "Truck LEAVING the bridge. capacity
      = " << capacity << std::endl;
} };
```

Online Store

```
class Item {
private:
    int rating_sum = 0;
    int rating_count = 0;
    std::recursive_mutex mtx;
public:
    Item() {};
    /* Returns average rating. 0 if no rating
        occured */
    double get_rating() {
        std::lock_guard<std::recursive_mutex>
            lock(mtx);
        if(rating_count == 0) return 0.0;
        return (double)rating_sum / rating_count;
    }
    void add_rating(int stars){
        assert(1 <= stars && stars <= 5);
        std::lock_guard<std::recursive_mutex>
            lock(mtx);
        rating_sum += stars;
        rating_count++;
    }
    void out_rating(){
        std::lock_guard<std::recursive_mutex>
            lock(mtx);
        std::cout << "ratings:" << rating_count <<
            ", ";
        std::cout << "score:" << get_rating() <<
            "\n";
    }
};
```

Levenshtein Distance DP

```
#include <string>
#include <vector>

void mprint(int** M, unsigned int xlen, unsigned
    int ylen){
  for(unsigned int i = 0; i <= xlen; i++){
    std::cout << std::endl;
    for(unsigned int j = 0; j <= ylen; j++){
      std::cout << M[i][j] << " ";
    }
  }
}

void print(std::string s){
  std::cout << s << std::endl;
}

unsigned int min(unsigned int v, unsigned int w){
  return (v <= w ? v: w);
}

// action   cost
// Ins(a)   = 1;
// Del(a)   = 1;
// Rpl(a, e) = 1;
// Rpl(a, a) = 0;
```

```
// returns Levenshtein distance of two given
    strings
unsigned Levenshtein(const std::string& x, const
    std::string& y){
  const unsigned int xlen = x.length();
  const unsigned int ylen = y.length();

  // std::vector<std::vector<unsigned int>> C;
  unsigned int** C = new unsigned int*[xlen+1];
  for (unsigned int i = 0; i <= xlen; ++i){
    C[i] = new unsigned int[ylen+1];
  }

  // fill the edge cases in 0th row
  for(unsigned int j = 0; j <= ylen; j++){
    C[0][j] = j;
  }
       //     - P I Z Z A
       //   C 0_1_2_3_4_5
       // - 0|0 1 2 3 4 5
       // F 1|x x x x x x
       // R 2|x x x x x x
       // Y 3|x x x x x x
  for(unsigned int i = 1; i <= xlen; i++){
    C[i][0] = i; // fill egde cases in 0th col
       //     - P I Z Z A
       //   C 0_1_2_3_4_5
       // - 0|0 1 2 3 4 5
       // F 1|1 x x x x x
       // R 2|2 x x x x x
       // Y 3|3 x x x x x
    for(unsigned int j = 1; j <= ylen; j++){
      if(x[i-1] == y[j-1]){
        C[i][j] = C[i-1][j-1]; // case: Replace(a,
            a)
      }else{
        unsigned int m;
        m = min(C[i-1][j] + 1,     // case: Insert(a)
                C[i][j-1] + 1);    // case: Delete(a)
        m = min(C[i-1][j-1] + 1, m); // case:
            Replace(a, e)
        C[i][j] = m;
      }
    }
  }
  return C[xlen][ylen];
}
```

DFS Adjacency Matrix

```
using Node = std::size_t;

// color for graph traversal
enum class Color {
  white , grey, black
};

using Colors = std::vector<Color>;

// DFS on graph <g> starting from node <v> with
    coloring nodes (grey coloring not used here)
void DFS(const DiGraph& g, Node v, Colors& colors){
  colors[v] = Color::grey;
  for (size_t i = 0; i < g.number_nodes(); ++i){
    if (g.getEdge(v, i) > 0){
      if (colors[i] == Color::white){
        DFS(g,i,colors);
      }
    }
  }
  colors[v] = Color::black;
}

// DFS on graph g starting from node v, uses
    grey-coloring to detect cycles
// returns if cycle can be found starting from
    node v
bool findCycle(const DiGraph& g, Node v, Colors&
    colors){
  colors[v] = Color::grey;
  for (size_t i = 0; i < g.number_nodes(); ++i){
    if (g.getEdge(v, i) > 0){
      if (colors[i] == Color::white){
        if (findCycle(g,i,colors)) return true;
      } else if (colors[i] == Color::grey){
        return true;
      }
    }
  }
  colors[v] = Color::black;
  return false;
}

// post: return if graph is symmetric
//     i.e. edge is present iff its antiparallel
    edge is present
bool DiGraph::isSymmetric(){
  for (Node i = 0; i < n; ++i){
    for (Node j = i+1; j < n; ++j){
      if (getEdge(i,j) != getEdge(j,i)){
        return false;
      }
    }
  }
  return true;
}

// post: returns true if there is a path from node
    <from> to node <to>
//     via the directed edges of the graph
bool DiGraph::hasPath(Node from, Node to){
  Colors colors(n,Color::white);
  DFS(*this,from,colors);
  return colors[to] != Color::white;
}

// post: returns true iff from node <from> a cycle
    can be reached
//      by traversing forward along the directed
    edges of the graph
bool DiGraph::hasCycle(Node from){
  Colors colors(n,Color::white);
  return findCycle(*this, from, colors);
}

// post: returns true iff the graph contains no
    cycle
bool DiGraph::isAcyclic(){
  for (Node i = 0; i<n; ++i){
    if (hasCycle(i))
      return false;
  }
  return true;
}
```