Sortieralgorithmen
- Bubblesort
- Sortieren durch Auswahl (Selectionsort)
- Sortieren durch Einfügen (Insertionsort)
- Heapsort
- Mergesort
- Quicksort

Bubblesort
"Hochblubbern" der grössten Zahl, danach nur noch auf n-1 etc. rekursiv anwenden
- Best Case: O(n)
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$

Selectionsort
- Vordere Teil des Arrays sortiert, suche das kleinste Element im unsortierten Teil und füge es ans Ende des sortierten Teils, mache das so lange bis unsortierter Teil leer ist.
- Best Case: $O(n^2)$
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$

Insertionsort
- Wähle Element aus unsortiertem Teil aus und füge ihn an der richtigen Stelle in den sortierten Teil ein. Verschiebe alle Elemente, die grösser als das Element sind um eine Stelle nach hinten.
- Best Case: O(n)
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$

Heapsort
- Bilde Min-Heap (oder Max-Heap) aus dem zu sortierenden Teil, nehme dann immer das min-Element aus dem Baum (oder max-Element) und füge es in die sortierte Liste am Ende (oder am Anfang) ein.
- Best Case: O(n * log(n))
- Average Case: O(n * log(n))
- Worst Case: O(n * log(n))

Mergesort
- Teile zu sortierenden Teil in zwei kleinere Teile und rufe Mergesort rekursiv auf. Beim Mergen arbeite nach dem Reissverschlussprinzip.
- Best Case: O(n * log(n))
- Average Case: O(n * log(n))
- Worst Case: O(n * log(n))

Quicksort
- Wähle Pivotelement, setzte alle Elemente die kleiner sind als Pivot links davon, aller grösseren rechts davon. Rufe Quicksort rekursiv auf die beiden Teillisten auf.
- Best Case: O(n * log(n))
- Average Case: O(n * log(n))
- Worst Case: $O(n^2)$

Heaps

Ein Heap ist ein Binärbaum, bei dem die Heap-Bedingung gilt:

- **Minheap**
- Jeder Knoten ist kleiner als seine Kinder (in Array: A[i] <= A[2i] & A[i] <= A[2i + 1])
- Das kleinste Element steht an der Wurzel des Baumes

- Maxheap
- Jeder Knoten ist grösser als seine Kinder (in Array: A[i] >= A[2i] & A[i] >= A[2i + 1])
- Das grösste Element steht an der Wurzel des Baumes

- Extract Min (Max): Entferne Wurzel, setzte letztes Element an dessen Platz, stelle Heap-bedingung rekursiv wieder her.

Trees

Natürlicher Suchbaum = binärer Suchbaum

Preorder: Wurzel -> Links rekursiv -> Rechts rekursiv
Postorder: Links rekursiv -> Rechts rekursiv -> Wurzel
Inorder: Links rekursiv -> Wurzel -> Rechts rekursiv (falls Sortierbaum -> Sortierte Ausgabe)

AVL Baum:
Balancierter Baum, Balancefaktoren nie grösser als 1, sonst muss Baum neu balanciert werden.
Einfachrotation: Wenn linker Teilbaum linkslastig oder rechter Teilbaum rechtslastig ist.
Doppelrotation: Wenn linker Teilbaum rechtslastig oder rechter Teilbaum linkslastig ist.

Graphen

Gerichteter Graph besteht aus Menge $V = \{v_1, \ldots, v_n\}$ von Knoten und einer Menge $E \subseteq V \times V$ von Kanten.
Ungerichteter Graph besteht aus Menge $V = \{v_1, \ldots, v_n\}$ von Knoten und einer Menge $E \subseteq \{\{u, v\} \mid u, v \in V\}$ von Kanten.
Schleife Kante der Länge 1 die wieder zum gleichen Knoten geht.
Vollständiger Graph E enthält jede Kante zwischen zwei verschiedenen Knoten, d.h. jeder Knoten ist direkt mit allen anderen Knoten verbunden.
Bipartiter Graph ist ein Graph dessen Knotenmenge in zwei disjunkte Mengen U und V aufteilen lässt, wobei in den Mengen kein Knoten mit einem anderen verbunden ist.
Eingangsgrad $\deg^-(v)$ des Knotens v ist die Anzahl Kanten, die "zu ihm gehen"
Ausgangsgrad $\deg^+(v)$ des Knotens v ist die Anzahl Kanten, die "von ihm wegführen"

Eingangsgrad und Ausgangsgrad sind nur für gerichtete Graphen sinnvoll, für ungerichtete Graphen bezeichnet man es nur als **Grad** von v: deg(v). Achtung: Eine Schleife erhöht den Grad in einem *ungerichteten Graphen* um 2, in gerichteten Graphen wird eine Schleife aber nur Einfach gezählt!

Es gilt:
1) Die Summe aller $deg^-(v_i)$ = Die Summe aller $deg^+(v_i)$ = |E| falls G gerichtet
2) Die Summe aller deg(v) = 2 |E| falls G ungerichtet ist.
Weg ist eine Sequenz von Knoten, wenn zwischen allen Knoten des Weges eine Kante existiert.
Pfad ist ein Weg der keinen Knoten zweimal benutzt.
Ein ungerichteter Graph, bei dem zwischen allen Knotenpaaren ein Weg existiert heisst **zusammenhängender Graph**.
Ein Weg, bei dem Start- und Endknoten gleich sind heisst **Zyklus** und ein Zyklus der keinen Knoten (ausser Start / Ende) und keine Kante mehr als einmal benutzt. Ein Graph ohne kreis ist ein **Kreisfreier Graph**.

Die **Adjazenzmatrix** eines Graphen mit n Knoten ist eine n x n-Matrix, bei der der Eintrag $a_{i, j}$ 1 ist, falls zwischen den Knoten v_i und v_j eine Kante besteht, und 0 sonst. Ist $a_{i, i}$ 1, hast der Knoten v_i eine Schleife.

Die **Adjazenzliste** eines Graphen mit n Knoten ist eine Liste mit n Einträgen, wobei jeder Eintrag eine Liste ist, der die Nachfolger vom Knoten v_i ist.

Wenn eine Adjazenzmatrix A quadriert wird (d.h. A*A) gerechnet wird, sind genau die Einträge $a_{i, j}$ 1, falls es zwischen den Knoten a_i und a_j einen Weg der Länge 2 gibt. Generell kann man so durch Berechnen von A^x berechnen, zwischen welchen Knoten es einen Weg der Länge x gibt.

Die **Reflexive Hülle** eines Graphen bzw. einer Adjazenzmatrix erhält man, indem man bei jedem Knoten eine Schleife zu sich selbst hinzufügt, d.h. alle $a_{i, i}$ in der Matrix auf 1 setzt.
Die **Transitive Hülle** eines Graphen bzw. einer Adjazenzmatrix ist die Matrix die entsteht, wenn man alle $a_{i, j}$ auf 1 setzt, falls es für die Knoten v_i und v_j einen beliebig langen Weg gibt. Um diese zu berechnen, gehen wir wie folgt vor:
1) Für alle i, j, k $\in$ {1, ..., n} durchiterieren und setzte $a_{i, j}$ auf 1 falls $a_{i, k}$ und $a_{k, j}$ 1 sind. Nach diesem Schritt beinhaltet die Matrix alle Wege der Länge 1 und 2.
2) Widerhole dies, um alle Wege der Länge max 4 zu finden, danach alle Wege der Länge max 8 usw. Dies muss man nur $log_2 n$ mal wiederholen um die Transitive Hülle zu erhalten.
Macht man beides, erhält man dann die **Reflexive und Transitive Hülle** des Graphen.

Tiefensuche Starte bei Knoten und folge einem Pfad der dort startet, bis man am Ende des Pfades ist oder einen schon besuchten Knoten findet. Erst dann geht man einen Schritt zurück. Die Tiefensuche läuft in $\Theta(|V| + |E|)$.

Breitensuche Starte bei einem Knoten und schaue zuerst alle Nachfolger von diesem Knoten an. Dann gehe eine Stufe weiter und schaue die Nachfolger der Nachfolger an, usw. Die Breitensuche läuft ebenfalls in n $\Theta(|V| + |E|)$.

Zusammenhangskomponente ist ein Teilgraph G' = (E',V') eines Graphen G, der nicht mit G verbunden ist, d.h. E' = {{v, w} $\in$ E | v, w $\in$ V'} und in E (also im gegebenen Graphen) keine Kante existiert, die einen Knoten in V' und einen in V \ V' hat.

Die **Topologische Sortierung** eines Graphen G ist so, dass die Vorgänger eines Knotens v vor ihm, und seine Nachfolger nach ihm in der Sortierung stehen, z.B. als Dependenciesliste während der Installation eines Programms. Ein gerichteter Graph hat nur dann eine Topologische Sortierung, wenn er kreisfrei, d.h. ein **Gerichteter Azyklischer Graph (DAG)** ist.

Topologische Sortierung Algorithmus:

```
1  S ← EmptyStack                          ▷ Stapel S initialisieren
2  for each v ∈ V do A[v] ← 0
3  for each (v, w) ∈ E do A[w] ← A[w] + 1  ▷ Berechne Eingangsgrade
4  for each v ∈ V do                       ▷ Lege Knoten mit Eingangs-
5      if A[v] = 0 then Push(S, v)         ▷ grad 0 auf den Stapel S
6  i ← 1
7  while S not empty do
8      v ← Pop(S)                          ▷ Knoten mit Eingangsgrad 0
9      ord[v] ← i; i ← i + 1               ▷ Weise korrekte Position zu
10     for each (v, w) ∈ E do              ▷ Verringere Eingangsgrad
11         A[w] ← A[w] − 1                 ▷ der Nachfolger
12         if A[w] = 0 then Push(S, w)
13 if i = |V| + 1 then return ord else "Graph enthält einen Kreis"
```

Algorithmus von Dijkstra

Der Algorithmus von Dijkstra berechnet anhand des Greedy-Prinzipes den kürzesten Pfad zwischen zwei Knoten eines gerichteten Graphen für einen gegebenen Startknoten.

Funktionsweise:
1) Weise jedem Knoten die beiden Eigenschaften "Vorgänger" und "Distanz" zu. Initialisiere die Distanz im Startknoten mit 0 und in allen anderen mit ∞.
2) Solange es noch unbesuchte Knoten gibt, wähle darunter denjenigen aus mit minimaler Distanz zum aktuellen Knoten und
    1) speichere, dass dieser Knoten schon besucht wurde
    2) Berechne für alle noch unbesuchten Nachbarknoten die Summe des jeweiligen Kantengewichts und der Distanz, die im aktuellen Knoten gespeichert ist.
    3) Ist dieser neue Wert für einen Knoten kleiner als der dort schon gespeicherte Wert, aktualisiere den Wert und setze den aktuellen Knoten als Vorgänger (wird als Update bzw. Relaxieren bezeichnet.)
4) Trifft man "unterwegs" auf den Endknoten ist der Algorithmus beendet. Und die Kosten die man dort einträgt sind die minimalen Kosten.

Achtung: Der Algorithmus von Dijkstra funktioniert nur für Graphen, wo jede Kante positives Gewicht (d.h. >= 0) hat.

Algorithmus von Bellman-Ford

Der Algorithmus von Bellman-Ford kann auch kürzeste Wege in Graphen berechnen, in denen negative Gewichte vorkommen, jedoch dürfen darin keine Zyklen vorkommen die insgesamt negatives Gewicht haben, da sonst diese unendlich durchlaufen würden und es so keinen kürzesten Weg geben würde.

Funktionsweise:
1) Weise jedem Knoten die beiden Eigenschaften "Vorgänger" und "Distanz" zu. Initialisiere die Distanz im Startknoten mit 0 und in allen anderen mit ∞.
2) Mache die folgende Anweisung n-1 Mal (da die günstigsten Wege ohne negative Kreise so alle korrekt berechnet wurden):
    1) Für jede Kante, untersuche ob die Distanz im Zielknoten der Kante grösser ist als die Distanz vom Anfängerknoten + das Gewicht der Kante. Falls die Distanz im

Zielknoten grösser ist, aktualisiere die Distanz zu (Distanz Anfangsknoten + Gewicht Endknoten) sowie den Vorgänger des Endknotens zum Startknoten dieser Kante. So kann man für alle Knoten herausfinden, was die minimalen Kosten sind, um diese von einem gegebenen Startknoten zu erreichen.

3) Nachdem alle Distanzen so berechnet wurden, schaue für jeden Knoten ob die Distanz des Endknotens grösser ist als die Distanz des Startknotens + das Gewicht der Kante. Falls ja, hat der Graph einen Kreis mit insgesamt negativer Distanz und der Algorithmus bricht ab mit einem Fehler (da für einen Graphen mit negativem Kreis keine kürzeste Distanz berechnet werden kann.)

Algorithmus von Floyd-Warshall

Der Algorithmus von Floyd-Warshall berechnet für alle Paare von Knoten in einem Graphen den kürzesten Weg (im Gegensatz zu Dijkstra und Bellman-Ford, die dies nur für gegebene Startpunkte erreichen).

Der Algorithmus basiert auf der Dynamischen Programmierung und dem Fakt, dass wenn der kürzeste Weg von u nach v durch w geht, dann sind die Teilpfade (u, w) sowie (w, v) auch minimal.

Funktionsweise:
1) Sei G ein Graph wo alle Knoten von 1 bis n durchnummeriert sind. Initialisiere für alle i, j (d.h. für alle Paare an Knoten, die direkt verbunden sind durch die Kante (v_i, v_j)) die Distanz d[i, j] als w[i, j] (d.h. für alle Knoten die direkt verbunden sind ist die Distanz in diesem Schritt das Gewicht des direkten Pfades).
2) Alle Knoten v_i und v_j die nicht direkt miteinander verbunden sind haben d[i, j] = ∞.
3) Wiederhole für k=1 bis k=n:
    1) Berechne für alle Paare i, j den Wert d[i, j] = min(d[i, j], (d[i, k] + d[k, j])) d.h. prüfe ob der Wert der schon gespeichert ist, durch einen "Umweg" bzw. anderen Weg unterboten werden kann, wenn ja, aktualisiere den Wert d[i, j].
4) Am Ende des Algorithmus ist überall wo d[i, j] ungleich ∞ ist die Distanz des kürzesten Weges gespeichert.

Spannbaum

Ein Spannbaum ist ein Teilgraph eines ungerichteten Graphen, der ein Baum ist und alle Knoten dieses Graphen enthält. Spannbäume existieren nur in zusammenhängenden Graphen. Ein **minimaler Spannbaum** ist dabei ein Spannbaum, bei dem die Summe der Kantengewichte minimal ist.

Algorithmus von Kruskal

Um einen minimalen Spannbaum zu erhalten, kann man mit dem Algorithmus vom Kruskal, der nach dem Greedy Prinzip arbeitet, diesen einfach berechnen.

Funktionsweise:
1) Führe so oft wie möglich aus: Wähle aus den noch nicht ausgewählten Kanten des Graphen G die kürzeste Kante (Kante mit kleinstem Gewicht), die mit den schon ausgewählten Kanten keinen Kreis bildet.
2) Wenn dies nicht mehr weiter möglich ist, d.h. jede Kante die dazu genommen würde,

würde einen Kreis bilden, ist der Algorithmus beendet und man hat einen minimalen Spannbaum.

Achtung: Normalerweise gibt es mehrere minimale Spannbäume und nicht *den* minimalen Spannbaum, da man oft durch wählen anderer Kanten mit gleichem Gewicht am Ende einen anderen Baum erhält.

Algorithmus von Prim

Der Algorithmus von Prim berechnet ebenfalls für einen gegebenen Graphen G = (V, E) einen minimalen Spannbaum. Dieser Algorithmus funktioniert ebenfalls nach dem Greedy Prinzip.

Funktionsweise:
1) Wähle einen beliebigen Knoten des Graphen als Startgraphen T des minimalen Spannbaumes.
2) Solange der Baum (bzw. Graph) T noch nicht alle Knoten von G enthält:
    1) Wähle eine Kante e mit minimalem Gewicht aus, die einen noch nicht in T enthaltenen Knoten v mit T verbindet.
    2) Füge e und v dem Graphen T hinzu.
3) Wenn so alle Knoten dem Graphen T hinzugefügt wurden ist der Graph T ein minimaler Spannbaum des Graphen G.

Abstrakte Datentypen

Union-Find-Struktur

Die Union-Find-Struktur ist ein abstrakter Datentyp, der die Partition einer Menge verwaltet. In der Graphentheorie wird er vor allem zur Verwaltung von Zusammenhangskomponenten von Graphen benutzt.
Der Datentyp unterstützt die Operationen *MakeSet*, *Find* und *Union***.** Diese Operationen funktionieren wie folgt:

MakeSet: Die MakeSet Operation erstellt aus einer Menge S aus n Elementen n disjunkte Teilmengen, die je ihr einziges Element als Repräsentanten haben, bzw. erstellt aus einem einzelnen Element eine Menge mit sich selbst als Repräsentant.

Find(x): Find(x) findet die Teilmenge, in der sich das Element x befindet, und gibt den Repräsentanten k davon zurück.

Union(r, s): Die Operation Union(r, s) vereinigt die beiden Teilmengen, die zu den Repräsentanten r und s gehören, zu einer Menge und bestimmt r zum Repräsentanten der neuen Menge.

Man kann die Zugehörigkeiten trivial in einem Array speichern, wo A[i] den Repräsentanten speichert, zu dem das i-te Element gehört, in der Praxis jedoch werden meistens Bäume verwendet, da diese normalerweise kürzere Laufzeiten garantieren.

Vorrangwarteschlange (Priorityqueue)

Die Priorityqueue ist eine erweiterte Version einer normalen Queue. Anders als bei der normalen Queue, bei der die Elemente in der Reihenfolge ausgegeben werden, wie sie eingefügt wurden, wird hier ein Schlüssel mit dem Element mitgegeben, der die Reihenfolge der Ausgabe der Elemente bestimmt (so wird ein Element mit einer höheren Priorität zuerst ausgegeben, auch wenn es später als andere Elemente eingefügt wird.)

Die Priorityqueue hat die beiden Operationen *insert* und *extractMin.* Die Operation insert fügt ein Element mit seiner Priorität in die Queue ein, und extractMin entfernt das Element mit dem kleinsten Schlüssel (= höchste Priorität) aus der Queue und gibt dieses zurück.

Die Priorityqueue kann mit AVL-Bäumen effizient implementiert werden, da sowohl insert als auch extractMin in O(log n) Zeit ausgeführt werden können.

Max Subarray Problem

Sei A ein Array mit n Elementen, in dem es sowohl positive als auch negative Werte drin hat. Gesucht ist nun ein Teilarray (d.h. ein zusammenhängender Teil des Arrays), das maximale Summe der Elemente hat. Erlaubt ist auch das leere Array (wenn alle Werte negativ wären, so dass 0 grösser ist).
Obwohl es bei einigen Eingaben mehr als eine Lösung gibt, genügt es, eine beliebige maximale Lösung auszugeben.

Funktionsweisen:

Vorberechnung der Präfixsummen
1) Berechne für jede Position i von 1 bis n die Summe S_i = Summe aller Elemente von 1 bis i. Dies benötigt O(n) Zeit.
2) Für die Berechnung der Zahl von i bis j reicht es nun aus, S_{i-1} von S_j abzuziehen, d.h. man kann die Summe des Intervalls (i, i+1, …, j-1, j) durch eine Subtraktion sehr schnell und einfach berechnen, was O(1) Zeit benötigt.
3) Für alle i als Startwert
    1) Gehe durch alle j als Endwert durch und berechne die Summe der Elemente von i bis j, merke immer das aktuelle Maximum.
4) Gebe das Maximum aus.

Dieser Algorithmus benötigt $\Theta(n^2)$ Zeit.

Divide-and-Conquer
Beim Divide-and-Conquer Ansatz nutzt man aus, dass genau 3 Fälle eintreten können:
1. Fall: Die Lösung liegt vollständig in der rechten Hälfte des Arrays.
2. Fall: Die Lösung liegt vollständig in der linken Hälfte des Arrays.
3. Fall: Die Lösung läuft über die Mitte des Arrays.

Für die beiden ersten Fälle kann man das Problem einfach durch einen rekursiven Aufruf lösen, solange bis nur noch ein Element übrig ist.
Für den dritten Fall machen wir folgende Beobachtung: Diese Lösung setzt sich aus der grössten Summe an Präfixsummen aus der ersten und der grössten Summe an Suffixsummen der zweiten Hälfte zusammen.

So kann man einfach für jeden Aufruf (auch die rekursiven) einfach alle 3 Fälle berechnen, und dann je das Maximum davon Ausgeben. Dieser Algorithmus funktioniert in Θ(n log n) Zeit.

Induktiv von links nach rechts
Wir nehmen an, dass wir nach dem (i – 1)-ten Schritt der Induktion den Wert max einer optimalen Lösung für die ersten i – 1 Elemente kennen. Wir kennen ebenfalls den Wert randmax, also eine beste Lösung die am rechten Rand endet. Wenn wir nun das i-te Element dazu nehmen, können wir überprüfen ob dadurch randmax grösser als max wird. Falls ja, ist max = randmax. Wenn randmax < 0 wird, wird randmax auf 0 gesetzt und das randmax wird "gelöscht". Dies macht man für i von 1 bis n, und am Ende gibt man den Wert max zurück, der der Wert des maximalen Subarrays ist.

Dieser Algorithmus läuft in Θ(n), und es ist nicht möglich eine schnelleren Algorithmus zu erhalten, da mindestens jedes der n Elemente 1 mal betrachtet werden muss, da sonst die Lösung nicht korrekt ist.

Rucksackproblem (Knapsackproblem)

Beim Rucksackproblem wollen wir eine Menge an Gegenständen mit dem Gewicht w_i und dem Wert v_i so aufteilen, dass wir ein Gewicht von w nicht überschreiten und den Gesamtwert v maximieren (als Beispiel: Bankräuber der möglichst viele wertvolle Dinge in seinen Rucksack packen will, damit er möglichst viel Geld klauen kann, jedoch kann er nur 20 kg an Last tragen).

Die Greedy-Algorithmen, die entweder die Dinge nach Gewicht oder nach Wertedichte (Wert / Gewicht) sortieren und dann die "wertvollsten" daraus auswählen sind zwar schnell, in Θ(n log n), jedoch ist die Lösung dabei meistens nicht korrekt.

Der korrekte Ansatz für diese Aufgabe ist, Dynamische Programmierung zu verwenden. Man erstellt dabei eine Tabelle die in der ersten Dimension (von oben nach unten) von 0 bis n (n = Anzahl Gegenstände) und in der zweiten Dimension (links nach rechts) von 0 bis W indiziert ist. Der Eintrag maxWert[i, w] ist dann der maximal erreichbare Wert, wenn man nur aus den Gegenständen {1, …., i} auswählt und das Gewicht maximal w betragen darf.
Zur Initialisierung der Tabelle setzen wir alle maxWert[0, w] auf 0 (da wir mit 0 Gegenständen maximal den Wert 0 erreichen können).
Für alle anderen maxWert[i, w] berechnen wir dann:
$maxWert[i, w] = max\{ maxWert[i – 1, w], maxWert[i – 1, w – w_i] + v_i \}$
D.h. für alle Einträge berechnen wir, ob es "besser" ist, den i-ten Gegenstand nicht zu nehmen (d.h. gleicher Wert wie Menge ohne diesen Gegenstand) oder den i-ten Gegenstand zu nehmen (d.h. Wert der Menge, in der der i-te Gegenstand reinpasst + Wert des i-ten Gegenstands).

Die Laufzeit des Problems ist Θ(nW), also Pseudopolynomiell.

Amortisierte Analyse

Bei der Amortisierten Analyse wollen wir die Kosten für teure Operationen durch billigere Operationen kompensieren, nämlich falls wir annehmen, dass die teure Operation und die billige Operation immer beide ausgeführt werden und wir so für die Operationen insgesamt eine bessere Laufzeit (= Kosten) erreichen können. Ein Beispiel für diese Analyse ist die

Implementierung eines Queues als zwei Stacks. Enqueue benötigt dabei nur O(1), da dies eine einfache push Operation auf den In Stack ist. Die Dequeue Operation hingegen benötigt im schlimmsten Fall O(n) Operationen (n mal pop auf dem In Stack, n mal push auf den Out Stack und noch ein pop auf dem Out Stack, wobei n die Anzahl Elemente in der Queue sind.)
So sieht die Dequeue Operation zwar sehr teuer aus, vor allem wenn sehr viele Elemente in der Queue drin sind. Wir können jedoch diese teure Operation kompensieren, da für i Elemente, die dequeued werden ja zuerst i Elemente enqueued werden müssen. Wir können jetzt wenn wir ein Element mit Enqueue zum Queue hinzufügen die Kosten auf 3 festlegen, d.h. wir "zahlen im Voraus", und zwar 1 für die push auf den In Stack, 1 für das pop vom In Stack und 1 für das push auf den Out Stack.
Somit hat die Enqueue amortisierte Kosten von 3, was immer noch in O(1) liegt, und Dequeue hat zwar tatsächliche Kosten von 2n + 1, da wir aber für jedes der n Elemente bereits "im Voraus" bezahlt haben, hat Dequeue amortisierte Kosten von 1, was in O(1) liegt und daher viel besser (günstiger) ist als O(n).

Dieser Ansatz können wir durch eine sogenannte **Potentialfunktion** Φ_i modellieren, die den Kontostand nach der Durchführung der ersten i Operationen darstellt. Da das Konto am Anfang leer ist, setzen wir $\Phi_0 = 0$. Es muss immer $\Phi_i \geq 0$ gelten.
Wir definieren nun die amortisierten Kosten der i-ten Operation als $a_i = t_i + \Phi_i - \Phi_{i-1}$
Es folgt dass die durchschnittlichen realen Kosten aller Operationen höchstens so gross sind wie die durchschnittlichen amortisierten Kosten aller Operationen, womit man mit den amortisierten Kosten eine obere Schranke für die tatsächlichen Kosten erstellen kann.

Im Beispiel mit dem Stack sei $\Phi_i = 2 *$ Anzahl der momentan gespeicherten Objekte auf dem In Stack. Eine Enqueue Operation hat tatsächliche Kosten 1, und amortisierte Kosten $a_i = t_i + \Phi_i - \Phi_{i-1} = 1 + 2 = 3$
Eine Dequeue Operation, falls der Out Stack nicht leer ist, kostet $t_i = a_i = 1$. Falls der Out Stack aber leer ist, müssen alle k Elemente auf dem In Stack umgeschichtet werden, d.h. die tatsächlichen Kosten sind dann $t_i = 2k + 1$. Die amortisierten Kosten dieser Operation sind $a_i = t_i + \Phi_i - \Phi_{i-1} = 2k + 1 + 0 - 2k = 1$.

Damit sind sowohl die Enqueue als auch die Dequeue Operation in amortisierter Zeit $\Theta(1)$ ausführbar.