Woche 5 – Übersicht & Tricks

(Disclaimer: Die hier vorzufindenden Notizen haben keinerlei Anspruch auf Korrektheit oder Vollständigkeit und sind nicht Teil des offiziellen Vorlesungsmaterials. Alle Angaben sind ohne Gewähr.)

Anmerkungen zu Serie 3: Allgemeines

- A, I sind die Arrays von unserem Freund. Daher können wir nicht wissen, ob $A_i = 0$ oder $I_i = 0$ für irgendwelche $i \in \mathbb{N}$.
- Wenn gezeigt werden soll, dass Schranke eng ist – und ihr beim Bestimmen der Schranke (in O-Notation) Abschätzungen **nach oben** verwendet, müsst ihr auch mit Abschätzungen **nach unten** beweisen, dass ihr gute Schranke bestimmt habt und nicht "zu weit" nach oben abgeschätzt habt.
- Wenn die Laufzeit eines Algorithmus (in Teilen) nicht von der Grösse der Eingabe abhängig ist, ist die Laufzeit konstant.
- Sehr komplizierten Pseudecode kurz kommentieren, damit der Leser versteht, was genau euer Algorithmus eigentlich macht. Denn da wir uns nicht auf eine konkrete Sprache einigen, können einige Textzeilen einiges bedeuten.
- Wenn ihr Algorithmen aus dem Skript benutzen wollt, verweist darauf – aber kopiert nicht Textpassagen. Zum einen ist das schlechter Stil und zum anderen nicht notwendig.

Heaps:

Heaps sind Datenstrukturen, die das sehr effiziente Extrahieren bestimmter (typischerweise Max./Min.) Schlüssel erlauben. Für einen, aus n Schlüsseln bestehenden Max-Heap kann beispielsweise der maximale Schlüssel in $O(\log(n))$ bestimmt werden.

Nun will ich zeigen:

1. Wie man einen Max-Heap definiert
2. Wie man den maximalen Schlüssel aus einem Max-Heap extrahiert
3. Wie man 1) und 2) verwendet, um aus einem beliebigen Array einen Heap zu konstruieren.

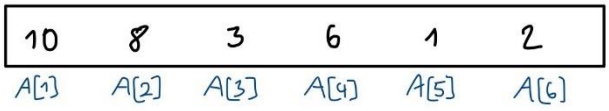
Beginnen wir mit der Definition des Max-Heap. Im Grunde genommen ist "Max-Heap" nur ein Name, den ein Array bekommt, wenn es eine folgende Eigenschaft erfüllt:

$$\text{Heap} - \text{Eigenschaft: Für Array A} = (\text{A}[1], \dots, \text{A}[\text{n}]) \text{ gilt:}$$

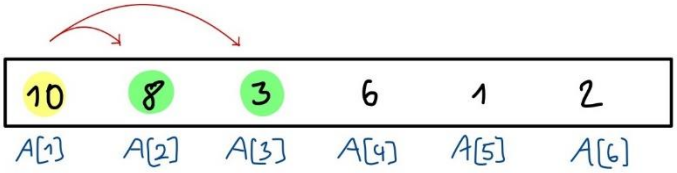
$$\forall k \in \{1, \dots, n\} : \big((2k \leq n) \Rightarrow (A[k] \geq A[2k])\big) \text{ und } ((2k+1 \leq n) \Rightarrow (A[k] \geq A[2k+1]))$$

Wenn man sich die Definition genau anguckt, ist schon klar, was von dem Array gefordert wird, um "Max-Heap" genannt zu werden, allerdings ist es leichter verständlich, wenn wir uns das Anhand eines Beispiels angucken:

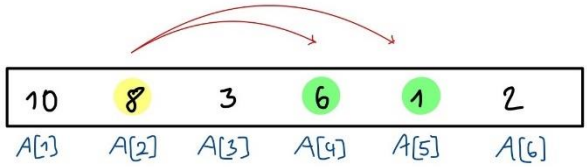
Betrachten wir folgendes Array:

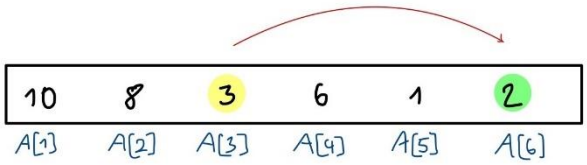


Der erste Teil (links von dem “und”) der Heap-Eigenschaft sagt folgendes: Wenn $2k \leq n$, also wenn die Position $2k$ existiert (da unser Array nur n Positionen hat), muss der Schlüssel an Position k mindestens so gross sein, wie der Schlüssel an Position $2k$, also $A[k] \geq A[2k]$. Der andere Teil sagt folgendes: Gibt es auch noch ein Element an Position $(2k + 1)$ in dem Array, muss $A[k]$ auch grösser als dieses Element sein, also $A[k] \geq A[2K + 1]$. Überprüfen wir das für $k = 1$ bei unserem Array: Wir sehen das $2k = 2 * 1 = 2 \leq 6 = n$ gilt, also muss gelten, dass $A[1] \geq A[2 * 1]$. Ausserdem gilt auch $2k + 1 = 2 * 1 + 1 = 3 \leq 6 = n$, also muss auch gelten, dass $A[1] \geq A[2 * 1 + 1]$:



Ich habe das Element $A[1]$ in Gelb und die Elemente $A[2 * 1], A[2 * 1 + 1]$ in Grün markiert. Wir sehen, dass die geforderten Eigenschaften für $k = 1$ gelten. Simultan prüfen wir für die restlichen Elemente und sehen, dass das Array tatsächlich ein Max-Heap ist:

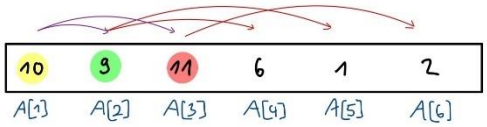




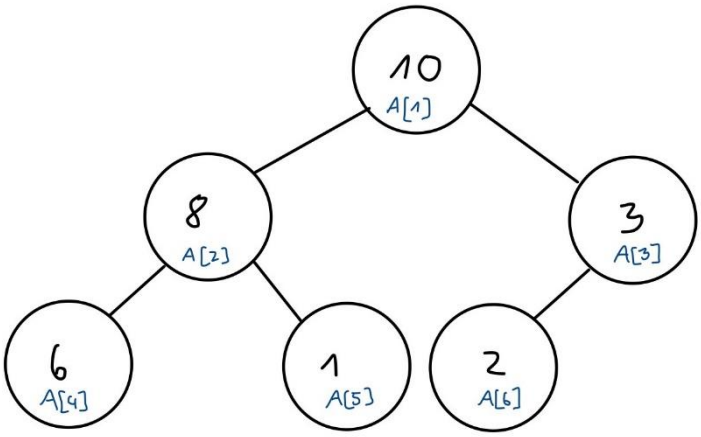
Für die Elemente $A[4], A[5], A[6]$ müssen wir nicht vergleichen, da $2 * 4 \geq 6$, etc. und damit die linke Seite der Implikation als falsch evaluiert, wodurch die Implikationen trivialerweise halten.

Es fällt auf, dass das Array, wenn es ein “Max-Heap” ist, **nicht sortiert** ist, sondern lediglich die Heap-Eigenschaft erfüllt. Man kann sich jedoch leicht davon überzeugen, dass ein absteigend sortiertes Array automatisch ein Max-Heap ist.

Der Vollständigkeit halber ist hier nun ein Array, dass man nicht als “Max-Heap” bezeichnen kann:



Nun fällt uns folgendes auf: Max-Heaps haben offensichtlich eine gewisse Struktur: Für jedes Element gibt es $\{0,1,2\}$ andere Elemente, für die gelten muss, dass sie höchstens so gross wie das Element sind. Das legt nahe, den Sachverhalt als Baum zu modellieren: Die einzelnen Schlüssel des Arrays werden dabei zu “Knoten” des Baums. Für den Knoten $A[k]$ gilt: Die maximal 2 Elemente $A[2k], A[2k + 1]$ sind “Nachfolger”-Knoten und werden mit einem Strich, einer sogenannten “Kante” verbunden. Das sieht dann so aus:



Damit klar wird, welche Array-Elemente sich wo in unserer Baum-Darstellung befinden, habe ich diese in die Knoten hereingeschrieben. Das ist allerdings sonst nicht notwendig – man schreibt gewöhnlicherweise nur den Schlüssel selbst in einen Knoten.

Wir sehen: die Nachfolgerknoten eines jeden Knoten sind genau die (0 bis max. 2) Elemente, für die gelten muss, dass sie höchstens so gross sein dürfen wie der Knoten selbst. Nun lässt sich mithilfe der Visualisierung auch die Heap-Eigenschaft viel schneller verifizieren: Anstatt mühsam im Array zur richtigen Position zu zählen, müssen wir nur an jedem Knoten prüfen, ob seine Nachfolgerknoten nur höchstens genau so gross sind wie er selbst.

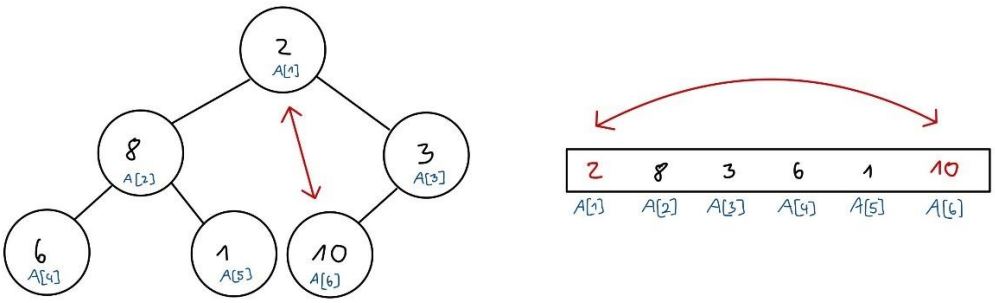
Denk jetzt aber bitte nicht, dass wir den Max-Heap so als Baum im Computer abspeichern! Wir verwenden nach wie vor das Array. Der Baum dient nur zu Visualisierung.

Nun will ich, das bisherige Wissen über die Definition und Visualisierung von Max-Heaps nutzend, zeigen, wie genau man das maximale Element aus einem Max-Heap in $O(\log(n))$ extrahieren kann:

Dazu gibt es ein klares “Rezept”, dass wir immer exakt so verwenden können:

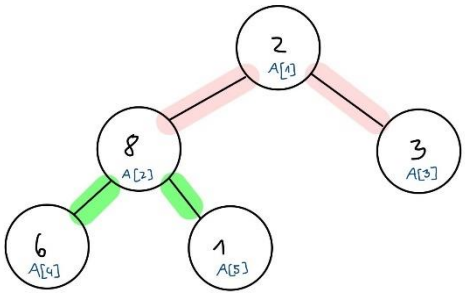
(Ich habe im Folgenden immer einerseits notiert, was “tatsächlich” im Array passiert und andererseits anhand der Visualisierung gezeigt, wie sich der zugehörige Baum verändert.)

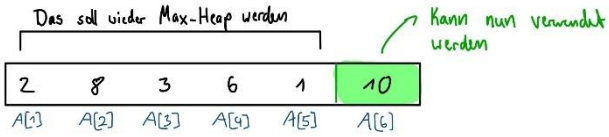
1. Als erstes Vertauschen wir das erste und das letzte Element aus dem Max-Heap.



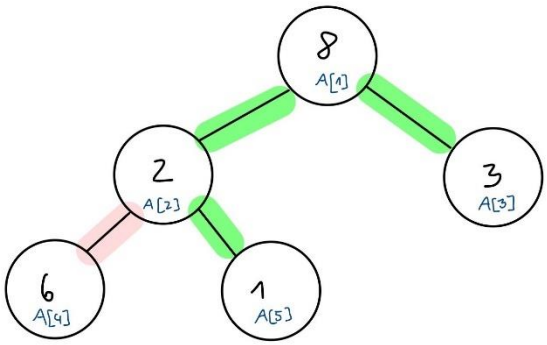
2. Nun kann das maximale Element, das jetzt an letzter Stelle steht, “extrahiert” werden. Wir werden diesen Teil des Arrays fortan ignorieren. Sind wir jetzt nicht fertig? Nein: Das

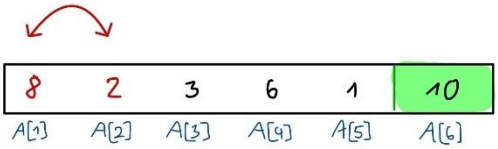
restliche Array ist leider kein Max-Heap mehr. Ich habe alle Kanten, an denen die Heap-Eigenschaft verletzt wird, rot markiert. Alle anderen sind grün markiert:



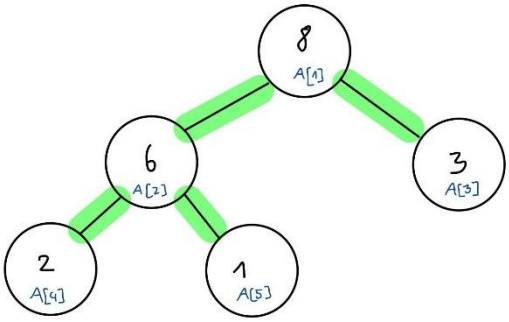


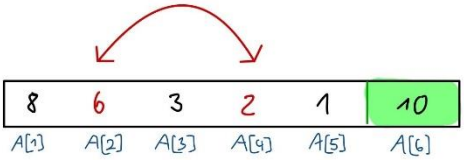
3. Die Heap-Eigenschaft ist also an der Wurzel verletzt, da nun ein sehr kleines Element oben liegt. Also müssen wir dieses nun im Heap "versickern" lassen: Wir tauschen es gegen seinen grösseren Nachfolger aus





4. An der Wurzel des Baums gilt die Heap-Eigenschaft nun offensichtlich wieder. Allerdings ist sie nun dort verletzt, wo wir die 2 hingetauscht haben. Also sind wir mit dem "Versickern lassen" offensichtlich noch nicht fertig. Und wiederholen den Schritt aus 3), bloss dieses mal mit der 6 und der 2:



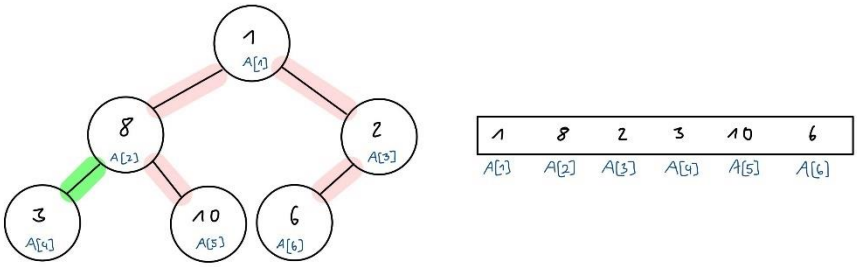


→ Wir sind fertig: Die Heap-Eigenschaft ist wieder überall hergestellt. Insgesamt müssen wir, um das maximale Element aus einem Max-Heap zu extrahieren, also: 1) Das erste und das letzte Element vertauschen und 2) das nach oben getauschte, letzte Element, "versickern" lassen. Da wir bei dem Versickern im Baum jedes Mal (bei jeder notwendigen Vertauschung) eine Stufe hinabsteigen, und der Baum nur Höhe $h = \log(n)$ haben kann, ist dies insgesamt also in $O(\log(n))$ möglich.

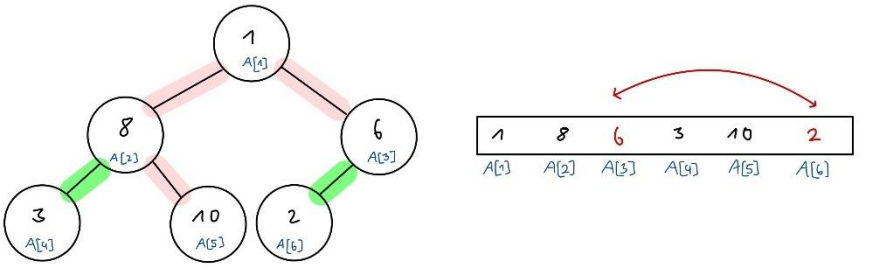
Nun wollen wir betrachten, wie wir diese Techniken nutzen können, um ein beliebiges Array in einen Max-Heap zu transformieren:

Dazu verfolgen wir wieder folgendes Rezept: Von hinten nach vorne lassen wir Elemente versickern, sodass nach dem Versickern von dem Element an Position k danach die Heap Eigenschaft für das Element an Position k und alle seine Nachfolger und Nach-Nach-….-Nachfolger gilt. Allerdings fällt uns folgendes auf: Nur Elemente an den Positionen $k = 1, \dots, \left\lfloor \frac{n}{2} \right\rfloor$ haben überhaupt Nachfolger. Für alle anderen Elemente gilt ja, dass $2 * k > n$. Das heisst, wir können an Position $k = \left\lceil \frac{n}{2} \right\rceil$ beginnen.

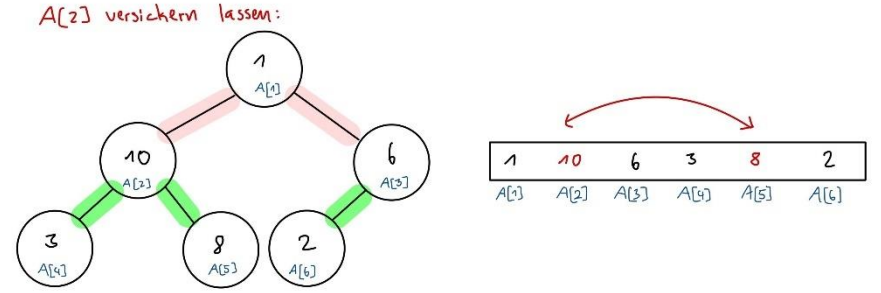
Im Folgenden nun ein unkommentiertes Beispiel. Ich habe wieder einerseits den Baum aufgezeichnet, als auch gezeigt, was im Array passiert:



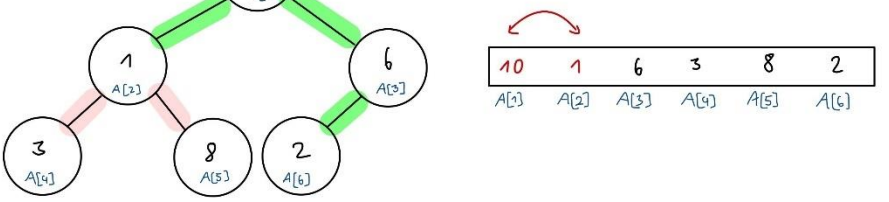
A[3] versickern lassen:

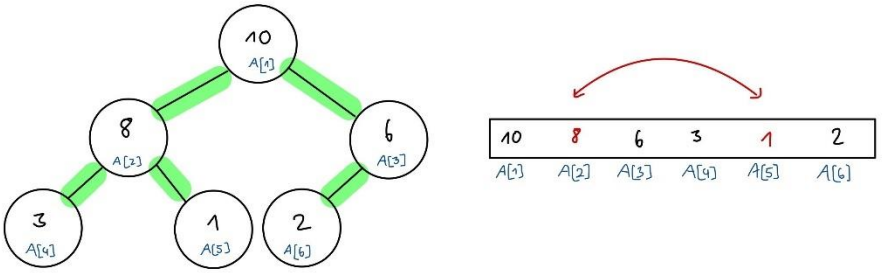


A[2] versickern lassen:



A[1] versickern lassen:





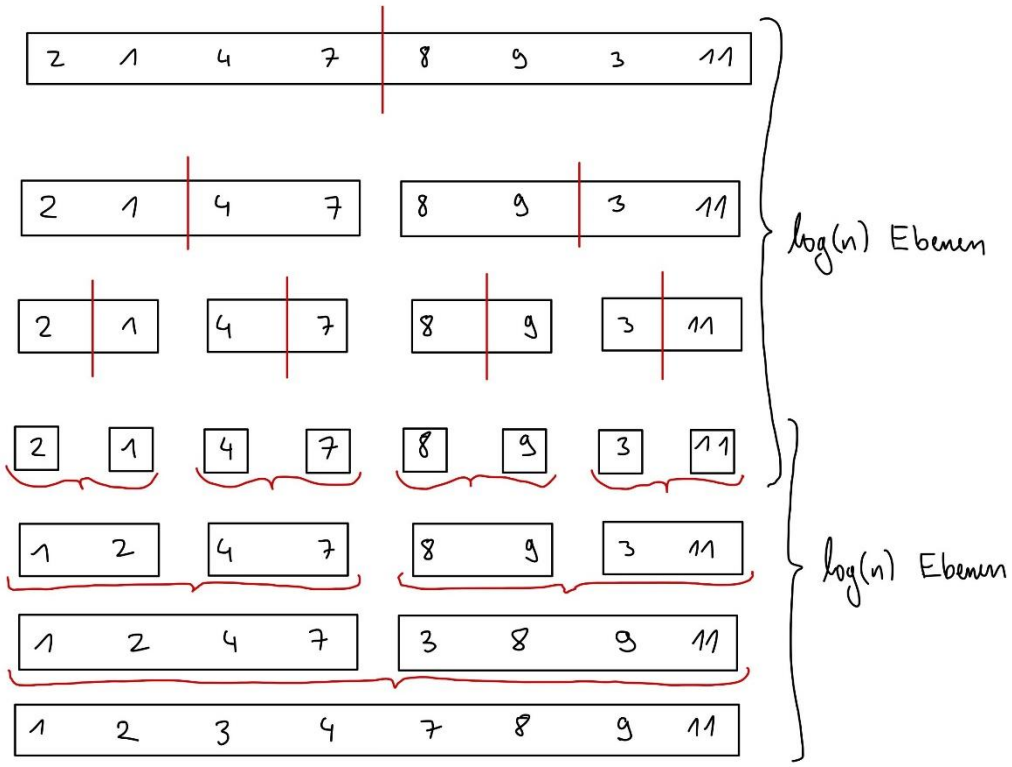
Insgesamt müssen wir also $\frac{n}{2}$ mal ein Element versickern lassen. Wir wissen, dass das Versickern in $O(\log(n))$ möglich ist. Also ist das Konstruieren insgesamt in $O(n * \log(n))$.

Heap-Sort:

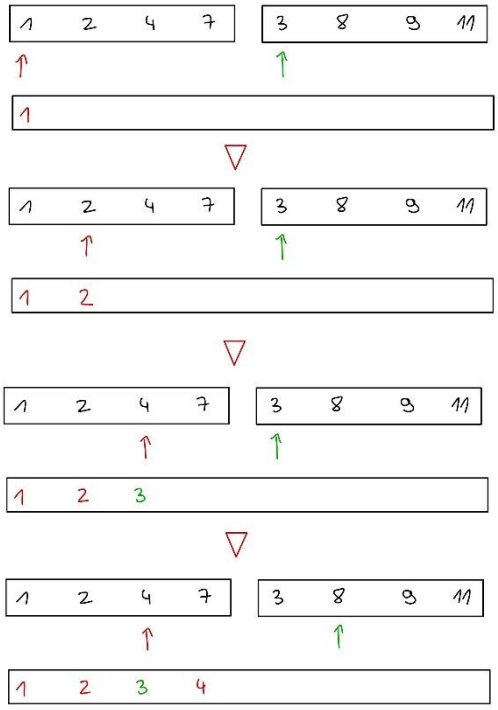
Der Sortieralgorithmus Heap-Sort macht sich Heaps zu Nutze, um in $O(n * \log(n))$ zu sortieren: Er funktioniert nach einem Prinzip, das extrem simpel ist – sobad man Heaps verstanden hat: Erst konstruiert er aus dem gegebenen Array einen Max-Heap. Das ist in $O(n * \log(n))$ möglich, wie wir gezeigt haben. Dann extrahiert er erst das grösste Element, verlegt es nach ganz hinten (durch Vertauschen mit dem letzten Element des Heaps) und verringert die Grösse des Max-Heaps nach Hinten um 1. Anschliessend repariert er den Heap durch Versickern lassen des nach vorne getauschten letzten Elements. Nun wiederholt er die gleiche Prozedur für das zweitgrösste Element, dann für das drittgrösste, etc. Insgesamt muss also n mal ein Element vertauscht und anschliessend der Max-Heap wieder hergestellt werden. Also läuft dies in $O(n * \log(n))$ und wir kommen insgesamt auf eine Laufzeit von: $O(n * \log(n) + n * \log(n))) = O(n * \log(n))$.

Merge-Sort:

Dieser Sortieralgorithmus basiert auf folgendem, grundsätzlich auch simplen Prinzip: Zunächst halbieren wir das Array. Dann halbieren wir die beiden Hälften. Dann halbieren wir die halbierten Hälften, etc. ....., bis die Hälften nur noch aus einem oder null Elementen bestehen. Dann "mergen" wir die Hälften, indem wir sie so zusammenfügen, dass sie sortiert sind. Das wiederholen wir dann für immer grösser werdende Teilstücke des Arrays – bis wir fertig sind. Aber da eine textliche Erklärung eines Algorithmus immer verwirrend ist, hier ein Beispiel:



Bevor wir zu Laufzeit kommen, wollen wir uns noch genauer angucken, wie der Merge-Algorithmus funktioniert:



Das Prinzip ist einfach: Wir verwenden zwei, durch die Teilstücke iterierende Zeiger, um aus sortierten Teilstücken grössere, sortierte Teilstücke zu konstruieren.

Nun zur Laufzeit: Die Anzahl der Vergleiche/Bewegungen lässt sich durch folgende Rekurrenzgleichung ausdrücken: $T(n) = 2 * T\left(\frac{n}{2}\right) + c * n$, für welche gilt: $T(n) \leq O(n * \log(n))$
Interessant ist, dass dieser Algorithmus jedoch nicht "in-place" ist, wir also für die Teilstücke zusätzlichen Speicherplatz (insgesamt $O(n)$) benötigen.

Quick-Sort:

Merge-Sort ist zwar schnell, aber dafür leider nicht in-place: Wir benötigen Extra-Speicherplatz. Also hat man sich eine Lösung dafür ausgedacht: Wir teilen das Array einfach in zwei Teile auf, indem wir ein beliebiges Element auswählen, und alle Elemente, die kleiner als dieses Element sind, in den linken Teil verschieben und alle Elemente, die grösser als dieses Element sind, in den rechten Teil verschieben. Genau das wiederholen wir dann in den beiden Teilen. Wie schnell der Algorithmus ist, hängt also offensichtlich davon ab, wie das Grössenverhältnis der beiden Teile ist. Sind sie ungefähr gleich gross, halbieren wir ja jedes Mal das Array und müssen das oben beschriebene Prozedere, das in $O(n)$ läuft, insgesamt offensichtlich nur $\log(n)$ mal ausführen, kämen also auf eine Laufzeit von $O(n * \log(n))$. Allerdings kann es passieren, das wir durch eine schlechte Wahl immer das jeweils kleinste/grösste Element auswählen und damit das Prozedere insgesamt n mal ausführen müssen. Dies hat allerdings (wie man sich mit einfacher Wahrscheinlichkeitsrechnung beweisen kann) eine extrem geringe Wahrscheinlichkeit, sodass Quicksort in der Realität ein sehr guter Algorithmus ist.

Dynamic Programming 1: Fibonacci – Aber in schnell?

In diesem Kapitel soll ein erstes Beispiel für die äusserst mächtige und wichtige Algorithmen-Entwurf-Technik "Dynamic Programming" gegeben werden. Wie genau man mit Dynamic Programming Probleme löst, werden wir in den kommenden Wochen noch sehr detailliert

besprechen, dieses Kapitel ist nun eher als eine Motivation – und um eine gewisse Intuition zu vermitteln – gedacht.

Die, eventuell aus dem Gymnasium bekannte, Fibonacci Sequenz hat folgende Berechnungsvorschrift:

$$f(1) = 1$$

$$f(2) = 1$$

$$f(n) = f(n-1) + f(n-2)$$

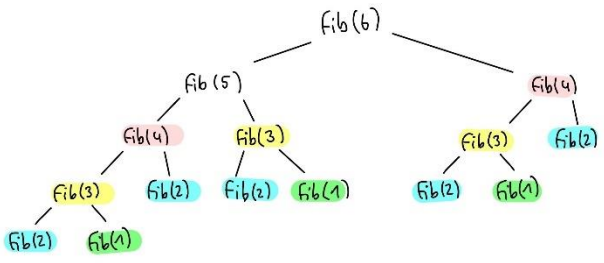
Hier sind mal einige Beispielwerte für die Fibonacci-Sequenz:

n	1	2	3	4	5	6	7	8	9
f(n)	1	1	2	3	5	8	13	21	34

Mit ein wenig Programmier/Algorithmik-Erfahrung lässt sich schnell ein einfacher Algorithmus entwerfen, der das n-te Element der Fibonacci-Sequenz berechnet:

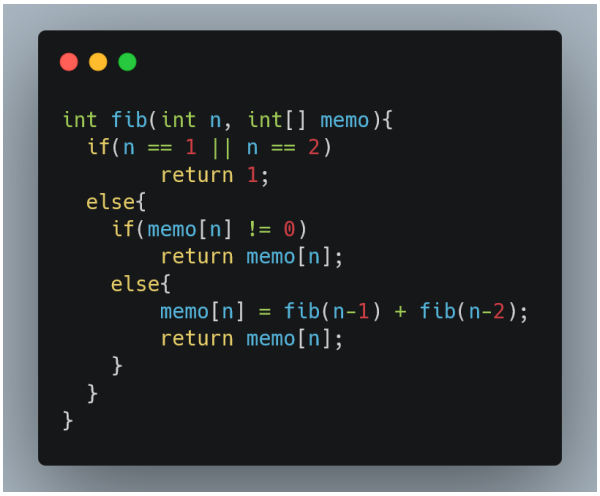
```
int fib(int n){
  if(n == 1 || n == 2)
        return 1;
  else
        return fib(n-1) + fib(n-2);
}
```

Allerdings gibt es ein Problem: Dieser Algorithmus ist **extrem** schlecht. Er ist sogar so schlecht, dass beispielsweise bereits über 2 Minuten dauern würde, fib(40) zu berechnen. Das liegt daran, dass dieser eine exponentielle Laufzeit hat und **extrem viele redundante Aufrufe macht:** Zeigen wir das doch mal anhand eines Beispiels, z.B. für $n = 6$:



Ich habe alle redundanten Aufrufe farbig markiert. Eigentlich müssen wir nur je einmal fib(1), fib(2), fib(3), fib(4), fib(5) berechnen, um fib(6) bestimmen zu können. Allerdings berechnen wir die einzelnen Werte immer wieder und verschwenden dabei extrem viel Zeit. Also gibt es folgende Idee:

Sobald wir ein Teilproblem gelöst (also hier bspw. fib(4) berechnet haben), speichern wir das Ergebnis ab, und wenn wir das nächste mal das Ergebnis dieses Teilproblems benötigen, gucken wir erst nach, ob wir es bereits berechnet haben – und verwenden dann einfach unser bereits berechnetes Ergebnis. Diese Dynamic-Programming Technik nennt man **Memoization** (Nein, kein Schreibfehler, wird wirklich so geschrieben). In Java könnte das so aussehen:



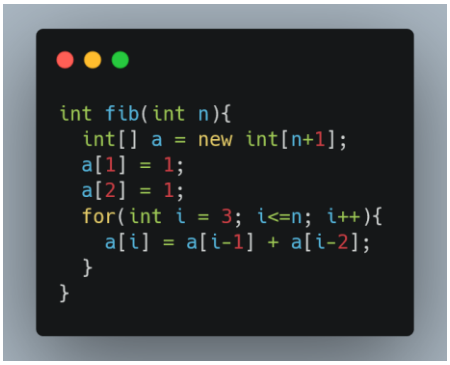
```java
int fib(int n, int[] memo){
  if(n == 1 || n == 2)
        return 1;
  else{
    if(memo[n] != 0)
        return memo[n];
    else{
        memo[n] = fib(n-1) + fib(n-2);
        return memo[n];
    }
  }
}
```

Dieses Programm läuft nun nicht mehr in exponentieller Laufzeit, sondern sogar linear. Allerdings benötigen wir nun den Extra-Speicher. Aber auch dieser hat lineare Grösse.

Es geht auch anders: Wir könnten auch so vorgehen, dass wir ein unser Ergebnis langsam "aufbauen": Wenn wir bspw. fib(3) berechnen, wissen wir, dass wir dafür erst fib(1) und fib(2) berechnen müssen. Oder wenn wir bspw. fib(5) berechnen wissen wir, dass wir dafür erst fib(4) und fib(3) (und dadurch also auch fib(2) und fib(1)) berechnen müssen. Also könnten wir ja auch einfach Stück für Stück alle fib(i) bis zu unserem n berechnen. Dieser Dynamic-Programming Ansatz heisst **bottom-up**.

Realisieren könnten wir das folgendermassen: Wir bauen einen Speicher (ein Array) in dem das Element an Position i den Wert von fib(i) speichert iterativ von links nach rechts auf: Also erst tragen wir $A[1] = 1,\ \ A[2] = 1$ ein. Jetzt berechnen wir fib(3): fib(3) = fib(2) + fib(1). Also: $A[3] = A[2] + A[1]$. Dann berechnen wir fib(4): fib(4) = fib(3) + fib(2), also $A[4] = A[3] + A[2]$ und verwenden dabei einfach die Werte für fib(3) und fib(2), die wir im Array schon an Position 3 und 2 gespeichert haben. Das wiederholen wir jetzt für alle i bis zu n.

In Java könnte das folgendermassen implementiert werden:



```java
int fib(int n){
  int[] a = new int[n+1];
  a[1] = 1;
  a[2] = 1;
  for(int i = 3; i<=n; i++){
    a[i] = a[i-1] + a[i-2];
  }
}
```

Die Laufzeit ist wieder offensichtlich in $O(n)$.