Woche 4 – 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 2: Allgemeines

- Korrektur einer vorhergehenden Zusammenfassung: In Induktionshypothese nicht den Existenzquantor ∃ verwenden, sondern die Aussage "für ein allgemeines/beliebiges..." annehmen.
- Argumentationen nicht zu kurz halten, Dinge die nicht trivial sind müssen vollständig ausgeschrieben und gezeigt werden. Zu Algorithmen gehört Angabe von Laufzeit (+ kurzer Begründung) und Korrektheitsbeweis. Aufpassen mit Indizes und Array-Elementen. Wenn "k" ein Index ist, ist k **nicht** automatisch auch das k-te Element!

Laufzeiten "erkennen":

Zur Erinnerung die Tricks für Summen von Woche 2:

1. $\sum_{i=1}^{n} i = \frac{n*(n+1)}{2}$
2. $\sum_{j=1}^{n}\sum_{k=j}^{n} 1 = \sum_{j=1}^{n}(n-j+1) = \frac{n(n+1)}{2}$ (denn $\sum_{j=1}^{n}(n-j+1)$ ist letzten Endes nur die Summe aus 1., bloss "andersherum gezählt")
3. $\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n} 1 = n^3$
4. $\sum_{i=1}^{n} i^3 = \frac{n^2*(n+1)^2}{4}$
5. $\sum_{i=1}^{n}\sum_{k=1}^{i} 1 = \sum_{i=1}^{n} i = \frac{n*(n+1)}{2}$
6. $\sum_{i=1}^{n} i*\log(i) \geq \sum_{i=\frac{n}{2}}^{n} i*\log(i) \geq \sum_{i=\frac{n}{2}}^{n} \frac{n}{2}*\log\left(\frac{n}{2}\right) \geq \frac{n^2}{4}*\log\left(\frac{n}{2}\right)$
7. $\sum_{i=1}^{n} i*\log(i) \leq \sum_{i=1}^{n} n*\log(n) = n^2*\log(n)$

In Serie 3 (Aufgabe 3.2) habt ihr bereits erste Erfahrungen mit dem bestimmen "enger" oberer Schranken für gegebene Algorithmen gemacht. Diese Fähigkeit ist beim Arbeiten mit Algorithmen oder Datenstrukturen auch grundsätzlich enorm wichtig, da ihr häufig anhand eines gegebenen (Pseudo-) Codes schnell erkennen können müsst, wie schnell eure Lösung tatsächlich ist. Daher gibt es solche Aufgaben auch immer wieder in Prüfungen – dort müsst ihr dann allerdings (**meistens**) nur das Endergebnis angeben. Allerdings sind diese Aufgaben meistens ein wenig schwieriger und es bedarf gewisser Tricks, auf die ich hier eingehen werde. Hier ein Beispiel:

/ 1 P

g) Geben Sie für das folgende Codefragment die asymptotische Laufzeit in Abhängigkeit von $n \in \mathbb{N}$ in **möglichst knapper Θ-Notation** an. Die Funktion f läuft in $\Theta(1)$ Zeit. Sie müssen Ihre Antwort nicht begründen.

```
1 for(int i = 1; i ≤ n; i = 2*i ) {
2       for(int j = 1; j*j*j ≤ n; j = j+1 ) {
3             for(int k = 1; k ≤ i*i; k = k+i ) {
4                   f();
5             }
6       }
7 }
```

Asymptotische Laufzeit:

So eine Aufgabe löst man normalerweise am leichtesten, indem man einfach die Loops in mathematische Summen umwandelt und diese dann auflöst. Hier zeige ich den naheliegendsten Ansatz, der immer für jeden Loop betrachtet, wieviele Iterationen dieser macht und das dann in Form einer Summe ausdrückt. Dieser Ansatz funktioniert **fast** immer, allerdings hier leider nicht. Es hat sich ein **typischer Fehler** eingeschlichen:

- Wir sehen, dass der äusserste Loop von $i = 1$ bis $i = n$ läuft, und sich in jeder Iteration das i aber verdoppelt. Insgesamt gibt es also nur $\lfloor \log_2(n) \rfloor + 1$ Iterationen des äusseren Loops. Gut, das können wir leicht mit einer Summe darstellen:
$$\sum_{i=1}^{\lfloor \log_2(n) \rfloor + 1} (INNERE\ LOOPS)$$
- Jetzt gucken wir uns den nächsten inneren Loop an: Dieser läuft von $j = 1$ bis $j * j * j = n$ und in jeder Iteration wird das j um eins inkrementiert. Also gibt es insgesamt $\lfloor \sqrt[3]{n} \rfloor$ Iterationen, da $j * j * j = j^3 > n$, sobald $j > \lfloor \sqrt[3]{n} \rfloor$ ist. Insgesamt haben wir also:
$$\sum_{i=1}^{\lfloor \log_2(n) \rfloor + 1} \sum_{j=1}^{\lfloor \sqrt[3]{n} \rfloor} (INNERE\ LOOPS)$$

- Es gibt aber noch einen Loop: Dieser läuft von $k = 1$ bis $k = i * i$, wobei jedoch k in jeder Iteration um i erhöht wird. In jeder Iteration dieses innersten Loops wird f() einmal aufgerufen. Also insgesamt gibt es hier i iterations in denen es je einen Funktions-Aufruf gibt. Insgesamt gibt uns das:
$$\sum_{i=1}^{\lfloor \log_2(n) \rfloor + 1} \sum_{j=1}^{\lfloor \sqrt[3]{n} \rfloor} \sum_{k=1}^{i} 1$$

- Spätestens jetzt kann man das **Problem (und damit auch den Fehler)** gut erkennen: Die Anzahl der Iterationen des inneren Loops ist von der Loop-Variable des äussersten Loops abhängig. Dieser hat (was korrekt ist) bloss $\lfloor \log_2(n) \rfloor + 1$ Iterationen. Allerdings wird i nicht bloss maximal $\lfloor \log_2(n) \rfloor + 1$ gross, sondern tatsächlich nimmt i die Werte $i = 1,2,4,8,16, \ldots \frac{n}{2}, n$ an. Das heisst, wenn wir die Summe so darstellen, wie wir es gemacht haben, also, dass i bloss von 1 bis $\lfloor \log_2(n) \rfloor + 1$ läuft, dann haben wir folgendes Problem: Der innerste Loop läuft nie bis n obwohl i tatsächlich zu irgendeinem Zeitpunkt n «gross» wird. Wie können wir das also elegant lösen und trotzdem die angenehme Summen-Schreibweise verwenden? Mit folgendem **Trick:**

- Wir sehen folgendes: Da i folgende Werte annimmt: $i = 2^x$ für $x = 0 \ldots \lfloor \log_2(n) \rfloor$ können wir auch einfach i zu x umbennen (Denn x nimmt auch bloss $\lfloor \log_2(n) \rfloor + 1$ viele verschiedene Werte an, was exakt der Anzahl der Iterationen des äussersten Loops entspricht). Und gleichzeitig lassen wir den innersten Loop einfach bis $k = 2^x$ laufen, da i immer eine 2er Potenz ist. Also so:
$$\sum_{x=0}^{\lfloor \log_2(n) \rfloor} \sum_{j=1}^{\lfloor \sqrt[3]{n} \rfloor} \sum_{k=1}^{2^x} 1$$
- Wir sehen: Der äusserste Loop führt immer noch exakt $\lfloor \log_2(n) \rfloor + 1$ Iterationen durch – genau wie er es soll. Der j-Loop immer noch $\lfloor \sqrt[3]{n} \rfloor$ Iterationen – genau wie er es soll. Und der innere Loop läuft jetzt immer bis zu den Werten, die i eigentlich annehmen soll. Also

nicht nur bis $i = \lfloor \log_2(n) \rfloor$ sondern bis $i = n$. Jetzt müssen wir diese Summe nurnoch auflösen und sehen:

$$\sum_{x=0}^{\lfloor \log_2(n) \rfloor} \sum_{j=1}^{\lfloor \sqrt[3]{n} \rfloor} \sum_{k=1}^{2^x} 1 = \sum_{x=0}^{\lfloor \log_2(n) \rfloor} \sum_{j=1}^{\lfloor \sqrt[3]{n} \rfloor} 2^x = \sum_{x=0}^{\lfloor \log_2(n) \rfloor} \lfloor \sqrt[3]{n} \rfloor * 2^x = \lfloor \sqrt[3]{n} \rfloor * \left(2^{\lfloor \log_2(n) \rfloor + 1} - 1\right)$$
$$= \theta\left(n^{\frac{4}{3}}\right)$$

Such-Algorithmen 1: Lineare Suche:

Lineare Suche ist der Name des gleichzeitig einfachsten und langsamsten Algorithmus, der das Suchproblem löst. Gegeben sei ein Array, sortiert oder unsortiert spielt keine Rolle, iteriert dieser einfach ab dem ersten Element so lange nach rechts bis er das zu findende Element entdeckt – oder am Ende des Arrays angelangt. Im Worst Case muss dieser Algorithmus also einmal durch das ganze Array iterieren und jedes mal das vorliegende Element untersuchen, was insgesamt n Operationen sind, also $O(n)$. Während lineare Laufzeiten für viele andere Probleme **extrem** gut sind, sind diese beim Suchen meistens eher ungünstig. Das liegt einfach daran, dass die Input-Sizes (hier also die grösse der Arrays, die durchsucht werden müssen) mitunter gigantisch sein können.

Allerdings hat dieser Algorithmus einen Vorteil: Es ist vollkommen egal, ob das Array, das durchsucht werden soll, sortiert oder unsortiert ist. Damit läuft er stabil **unter allen Bedingungen** immer in $O(n)$ – ist also nicht komplett nutzlos.

Such-Algorithmen 2: Binary Search:

Binary-Search ("Binäre Suche") ist ein extrem effizienter Suchalgorithmus, der auf dem "Divide-And-Conquer" Prinzip basiert. Wichtige Voraussetzung: Das Array, in dem wir ein Element suchen, **muss gemäss einer totalen Ordnung sortiert sein**. Wie genau funktioniert der Algorithmus?

Wir suchen ein Element **b** in einem Array **A[]** mit **n** Einträgen. Dazu initialisieren wir zwei weitere Variablen: **l = 0** und **r = n** («links» und «rechts»)

Jetzt gehen wir immer folgende Schritte ab:

1. Ist $l > r$? → JA? Dann gibt es das Element nicht in unserem Array, return -1;
2. Gucke das Element an Stelle $m = \left\lfloor \frac{l+r}{2} \right\rfloor$ an.
3. Ist $A[m] == b$? → JA? Dann fertig, return Index m
4. Ist $A[m] > b$ → JA? Dann setze $r := m$, da das Element offensichtlich links liegen muss und wir also unser window (l bis r) auf die linke Hälfte beschränken können
5. Sonst: → Setze $l := m$, da das Element offensichtlich rechts liegen muss und wir also unser window (l bis r) auf die rechte Hälfte beschränken können

```
int binarySearch(int a[], int b) {
        int l = 0, r = a.length - 1;
        while (l ⩽ r) {
            int m = l + (r - l) / 2;
            if (a[m] == b) return m;
            if (a[m] < b) l = m + 1;
            else r = m - 1;
        }
        return -1;
}
```

Warum genau ist dieser Algorithmus so effizient? Wir betrachten immer nur ein Teil des Arrays: Am Anfang suchen wir noch auf dem gesamten Array. Dann betrachten wir das in der Mitte liegende Element und verwenden, dass das Array sortiert ist. Ist dieses, in der Mitte liegende, Element nun bspw. grösser als das Element, das wir suchen, wissen wir sofort, dass auch alle Elemente, die rechts von diesem Element liegen, grösser als das Element, das wir suchen, sind. Also müssen wir nur die linke Hälfte betrachten. Das beschriebene Prozedere wiederholen wir dann auf der linken (bzw. rechten) Hälfte und dann wieder auf der dort bestimmten Hälfte etc. etc. etc. bis wir das Element entweder finden – oder unsere Hälften eine Grösse kleiner/gleich 0 bekommen, es also keine Möglichkeit gibt, das Element in dem Array noch zu finden.

Da wir das Array – und damit die Input-Size jedes Mal halbieren und dabei immer nur eine kurze Berechnung (zum Bestimmen des mittleren Elements) und den beschriebenen Vergleich ausführen, lässt sich die totale Laufzeit folgendermassen rekursiv beschreiben:

$$T(n) = \begin{cases} c, falls\ n = 0 \\ T\left(\frac{n}{2}\right) + d, falls\ n \geq 1 \end{cases}$$

Mit der Technik des Teleskopierens können wir diese Rekursion auflösen und eine geschlossene Form finden:

$$T(n) = T\left(\frac{n}{2}\right) + d = T\left(\frac{n}{4}\right) + d + d = T\left(\frac{n}{8}\right) + d + d + d = \cdots = T(0) + \log_2(n) * d$$
$$= c + \log_2(n) * d \leq O(\log(n))$$

Diese geschlossene Form könnte man nun noch per Induktion beweisen, was an dieser Stelle aber dem gelangweilten Leser als Übung überlassen wird.

Beweise per Invariante (Einführung):

Beweise per Invariante sind ein mächtiges Werkzeug um die Korrektheit von Algorithmen, in denen Loops vorkommen, zu beweisen. Im Grunde genommen beruhen diese Beweise auf einem einfachen Prinzip, das dem der mathematischen Induktion sehr ähnlich ist: Stelle eine Invariante auf, zeige, dass diese zu Beginn der Schleife gilt (**"Base Case"**), zeige, dass, wenn sie zu Beginn einer Iteration hält, sie auch am Ende einer Iteration hält ("**Induction Step**") und zeige, dass die Schleife terminiert und sie auch nach der Schleife gilt ("**Termination**").

Hier ein kleines Beispiel für einen Algorithmus, der die Elemente eines Arrays aufsummiert:

```
int sum(int a[]){
  int result = a[0];
  for(int i = 1; i<n; i++)
    result += a[i];
  return result;
}
```

Invariante: Zu Beginn der i-ten Iteration gilt: $result = \sum_{k=0}^{i-1} A[k]$

Base Case: Nach Invariante muss zu Beginn der ersten Iteration gelten: $result = \sum_{k=0}^{0} A[k] = A[0]$. Da result mit $A[0]$ initialisiert wurde, hält die Invariante hier.

Induction Hypothesis: : Sei $i \in \{0 \dots n-1\}$ beliebig. Wir nehmen an, dass die Invariante zu Beginn der i-ten Iteration hält, also $result = \sum_{k=0}^{i-1} A[k]$.

Induction Step: Da in der i-ten Iteration $A[i]$ zu result addiert wird, gilt dann automatisch zu beginn der $(i+1)$-ten Iteration: $result = \sum_{k=0}^{i-1} A[k] + A[i] = \sum_{k=0}^{i} A[k]$.

Termination: Die Schleife ist durch die Kondition begrenzt, dass die Schleifenvariable, die anfang zu 0 initialisiert wird, höchstens n sein darf - und in jeder iteration wird die Schleifenvariable inkrementiert. also terminiert der Algorithmus. Nach der letzten Iteration gilt: $i = n$, daraus folgt anhand der Invariante: $result = \sum_{k=0}^{n-1} A[k]$ und das ist die Summe des Arrays, womit die Korrektheit des Algorithmus bewiesen ist.

Sortieralgorithmen 1: Bubble-Sort:

Idee des Algorithmus: In der i-ten Iteration der äusseren Schleife verschieben wir den grössten Key des unfertigen Bereichs des Arrays an den rechten Rand dieses unfertigen Bereichs und verkleinern diesen unfertigen Bereich anschliessend auf der rechten Seite um 1. Anfangs betrachten wir natürlich das gesamte Array als "unfertigen Bereich". Die Korrektheit lässt sich leicht per Invariante beweisen, jedoch ist der Algorithmus sehr schlecht – da er sehr langsam läuft: $O(n^2)$

```
int bubbleSort(int a[]){
    for(int i=0; i<a.length-1; i++){
        for(int j=0; j<a.length-i-1; j++){
            if(a[j] > a[j+1]){
                int temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
}
```

Sortieralgorithmen 2: Selection-Sort:

Idee: In der i-ten Iteration verschieben wir das i-t grösste Element an die richtige Position. Dazu suchen wir im "unfertigen Bereich" nach dem i-t grösstem Element und vertauschen es an die Position $n - i$. Dieser Algorithmus ist allerdings nicht wirklich viel effizienter als Bubble-Sort und läuft auch in $O(n^2)$

Sortieralgorithmen 3: Insertion-Sort:

Idee: Zu Beginn der i-ten Iteration ist das Subarray $A[0: i-1]$ sortiert. In der i-ten Iteration gucken wir uns dann das Element $A[i]$ an und setzen es an die richtige Position im Subarray $A[0: i]$. Dieser Algorithmus läuft jedoch leider auch in $O(n^2)$.

```
int insertionSort(int a[]){
    for(int i=0; i<a.length; i++){
        int next_element = a[i];
        int j = i-1;
        while(j >= 0 && a[j] > next_element){ //Suche nach Position
          a[j+1] = a[j];    //Elemente nach Rechts verschieben
          j -= 1;
        }
        a[j+1] = next_element;
    }
}
```

Aufgabe zum Sortieren (HS20)

> i) Consider the sequence $6, 5, 4, 1, 2, 3$. How many swaps does `Bubble Sort` perform to sort this sequence? *Give the exact number of swaps required.*

Lösung: 12

Wir brauchen 5 Vertauschungen, um die 6 nach ganz hinten zu bringen. Danach nochmal 4, um die 5 an die korrekte Position zu bringen. Zum Schluss muss noch die 4 mit 3 Vertauschungen an die richtige Stelle gebracht werden, um das Array fertig zu sortieren.

ii) Consider the sequence $6, 5, 4, 1, 2, 3$. How many swaps does `Selection Sort` perform to sort this sequence? *Give the exact number of swaps required.*

Lösung: 4
Wir tauschen die 1 mit der 6, die 2 mit der 5, die 3 mit der 4 und zum Schluss die 4 mit der 6.

iii) Let $n \in \mathbb{N}$ be an even number and consider the sequence with the following structure:

$$2, 1, 4, 3, 6, 5, \ldots, n, n-1.$$

How many swaps does `Insertion Sort` perform to sort this sequence? *Give the exact number, not just the asymptotics.*

Lösung: $\frac{n}{2}$
Wir tauschen die Zahlen an Index i mit $i + 1$ für alle $i \in \{0, 2, 4, \ldots, n-2, n\}$.