# Algorithmen und Datenstrukturen

**Übungsblatt 6**

Ioan Oleksii Kelier Florian Keppeler

# Aufgabe 1

(a) Zuerst sortieren wir das ursprüngliche Array, was $\mathbb{O}(n \cdot log(n))$ Zeit kostet (z.B mit Merge Sort). Danach gehen wir durch das sortierte Array und schauen welche Paare von Elementen $(A[i], A[i+1])$ den kleinsten Abstand haben. Wir sollten uns nur das $i$-te und das $(i+1)$-te Element ansehen, denn da das Array sortiert ist, liegen die Elemente mit dem kürzesten Abstand nebeneinander.

Insgesamt brauchen wir $\mathbb{O}(n \cdot log(n) + (n-1)) = \mathbb{O}(n \cdot log(n))$ Zeit und es gilt $n \cdot log(n) \in o(n^2)$, da $n \cdot log(n)$ langsamer als $n^2$ wächst.

Dieser Algorithmus wird im Folgenden vorgestellt:

```
def min_dist(A: list[int]) -> tuple[int, int] | None:

    if not A:
        return None

    # Sort the array
    A = sorted(A)

    # Initial values
    min_ = float("inf")
    val1 = None
    val2 = None

    # Iterate through the array
    for i in range(len(A) - 1):
        diff = abs(A[i] - A[i + 1])

        # If the current difference is smaller
        # change values

        if diff < min_:
            min_ = diff
            val1 = A[i]
            val2 = A[i + 1]

    return val1, val2
```

(b) Da sich die Zahlen in einem binären Suchbaum und nicht in einem Array befinden, können wir 'in order traversal' verwenden, was $\mathbb{O}(n)$ Zeit kostet, um ein sortiertes Array mit den Zahlen zu erhalten. Wir können dann die beiden Elemente mit kleinsten Abstand auf die gleiche Weise wie in Teil $a$) finden. Insgesamt brauchen wir $\mathbb{O}(2n-1) = \mathbb{O}(n)$.

## Aufgabe 2

Die Grundidee dieses Algorithmus ist die Verwendung von DFS (Preorder). Zu Beginn initialisieren wir das maximale Gewicht gleich $-\infty$ und die Liste, in der der aktuelle Pfad gespeichert wird. Beim Durchlaufen des Baums fügen wir dem Array, das den aktuellen Pfad speichert, Elemente hinzu und aktualisieren den Summenwert. Wenn das aktuelle Element keine Kinder hat, ist dieses Element ein Blatt, was bedeutet, dass wir einen der Pfade gefunden haben. Wenn das aktuelle Gewicht größer ist als das vorherige, aktualisieren wir die Werte für die maximale Summe und den Pfad. Danach wird rekursiv nach den linken und rechten Kindern des aktuellen Elements gefragt. Am Ende müssen wir auch das letzte Element entfernen und seinen Wert von der Summe subtrahieren, um die richtigen Werte zu erhalten. Dieser Algorithmus wird im Folgenden vorgestellt:

```
MAX_SUM = float("-inf")
MAX_SUM_PATH = []

def dfs(node, current_path: list, current_weight):

    global MAX_SUM_PATH, MAX_SUM

    if not node:
        return

    current_path.append(node.value)
    current_weight += node.value

    # If left & right clildren are none
    # we have arrived at leaf node
    if not node.left and not node.right:

        # Update weight and path if
        # current values are bigger then previous
        if current_weight > MAX_SUM:
            MAX_SUM = current_weight
            # create a copy of a current path
            MAX_SUM_PATH = current_path[:]

    dfs(node.left, current_path, current_weight)
    dfs(node.right, current_path, current_weight)

    # Delete last elements from old path
    current_path.pop()
    current_weight -= node.value

def find_max_sum_path(root):
    if not root:
        return [], 0
    dfs(root, [], 0)
```

Da wir DFS (Preoder) verwenden, ist die Laufzeit des Algorithmus $\mathbb{O}(n)$. Nachfolgend sind einige

Beispiele für die Funktionsweise des Algorithmus aufgeführt (der gesuchte Weg ist rot markiert):

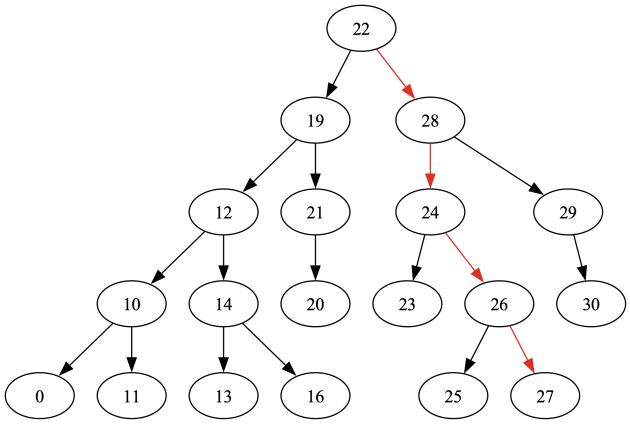


Abbildung 1: Beispiel 1

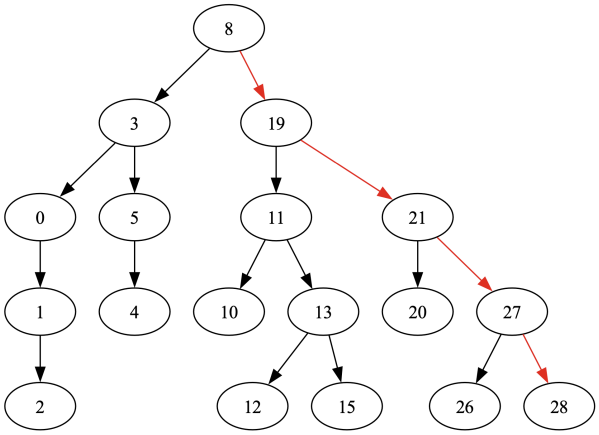


Abbildung 2: Beispiel 2

Der vollständige Code ist in der Datei bst_longest_path.py enthalten.