Aufgabe 1

(a) Anstatt der Kosten speichern wir den Baum in S und fügen jedem Baum noch eine Klassenvariable cost hinzu. Dann brauchen wir nur wenige Änderungen des Algorithmus aus der Vorlesung.

binTree[][] S = new binTree[n][n]
initialisiere S auf *undefiniert*
function OPTTREE(i, j)
  if S[i][j] *undefiniert* **then**
    if $i > j$ **then**
      S[i][j] = Binärbaum ohne Knoten mit cost = 0
    else if $i = j$ **then**
      S[i][j] = Binärbaum mit x_i als einzigem Knoten und cost = f_i
    else
      int c = ∞
      $f_{i,j} = \sum_{k=i}^{j} f_k$
      for k = i to j **do**
        int $c' = f_{i,j} +$ OPTTREE$(i, k-1)$.cost + OPTTREE$(k+1, j)$.cost
        if $c' < c$ **then**
          baue einen neuen Binärbaum mit Wurzel x_k,
          Kindern OPTTREE(i, $k-1$) und OPTTREE($k+1$, j),
          cost = c' und speichere in S[i][j]
          $c = c'$
        end if
      end for
    end if
  end if
  return S[i][j]
end function

Laufzeit Wie in der Vorlesung haben wir $\mathcal{O}(n^2)$ definierende Aufrufe (einen für jedes Paar (i, j)), die jeweils eine Laufzeit von $\mathcal{O}(n)$ besitzen. Die $\mathcal{O}(n^3)$ nicht definierenden Aufrufe haben eine Laufzeit von $\mathcal{O}(1)$. Insgesamt ist die Laufzeit also $\mathcal{O}(n^3)$.

(b) Es ändert sich nicht viel. Wir haben nun zusätzliche Kosten, die sich jeweils aus der Pfadlänge der nicht erfolgreichen Suche multipliziert mit der Frequenz g_i ergeben.

Wir ändern den obigen Algorithmus folgendermaßen ab:

- Die Kosten im Basisfall $i = j$ sind nun $f_i + g_{i-1} + g_i$
- Die Kosten im rekursiven Fall berechnen sich nun aus

$$c' = f_{i,j} + g_{i,j} + \text{OPTTREE}(i, k-1).\text{cost} + \text{OPTTREE}(k+1, j).\text{cost}$$

  wobei $g_{i,j} = g_{i-1} + g_i + g_{i+1} + \ldots + g_j$

Die Laufzeit ändert sich gegenüber 2(a) nicht.

Aufgabe 2

(a) Für $n = 3$ gibt es zwei mögliche Klammerungen: $(A_1 \cdot A_2) \cdot A_3$ oder $A_1 \cdot (A_2 \cdot A_3)$. Die Kosten für die erste Klammerung sind $p_0p_1p_2 + p_0p_2p_3 = p_0p_2(p_1 + p_3)$ und die Kosten für die zweite Klammerung sind $p_1p_2p_3 + p_0p_1p_3 = p_1p_3(p_0 + p_2)$. Wählt man $p_0 = p_2 = 1$ und $p_1 = p_3 = 2$ ergeben sich jeweils Kosten von 4 und 6. Für $p_0 = p_2 = 1$ und $p_1 = p_3 = N/3$ ergeben sich lineare und quadratische Kosten in Abhängigkeit von N.

(b) Für ein beliebiges gerades $n \geq 6$ wählen wir $p_i = 1$ für allen geraden i und $p_i = B = N/n$ für alle ungeraden i. Die Matrizen alternieren also zwischen Zeilen- und Spaltenvektoren und das Gesamtergebnis ist eine Zahl ($p_0 = p_n = 1$).

Eine mögliche Klammerung kombiniert in einem ersten Schritt jede Matrix mit geradem Index mit ihrem Vorgänger und geht danach einfach von links nach rechts vor:

$$(A_1 \cdot A_2) \cdot (A_3 \cdot A_4) \cdot (A_5 \cdot A_6) \ldots \cdot (A_{n-1} \cdot A_n)$$

Der erste Schritt hat Kosten $n/2 \times 1B1 = n/2 \times B$. Danach sind alle Zwischenergebnisse Zahlen und die verbleibenden $n/2 - 1$ vielen Multiplikationen haben Kosten 1.

Die gesamten Kosten betragen also $n/2 \times N/n + n/2 - 1 = N/2 + n/2 - 1$

Eine zweite mögliche Klammerung kombiniert in einem ersten Schritt jede Matrix mit geradem Index außer n mit ihrem Nachfolger und kombiniert danach die Ergebnisse ($B \times B$-Matrizen) von links nach rechts bevor in einem dritten Schritt mit A_1 und A_n multipliziert wird:

$$A_1 \cdot ((A_2 \cdot A_3) \cdot (A_4 \cdot A_5) \ldots \cdot (A_{n-2} \cdot A_{n-1})) \cdot A_n$$

Der erste Schritt hat Kosten $(n/2 - 1) \times B1B = (n/2 - 1)B^2$.
Der zweite Schritt hat Kosten $(n/2 - 2) \times BBB = (n/2 - 2)B^3$.
Der dritte Schritt hat Kosten $1BB + 1B1 = B^2 + B$

Die Differenz der Kosten ist also in $\Theta(N^3)$, was offensichtlich nicht übertroffen werden kann, da alle p_i kleiner N sind und somit alle $n - 1$ Multiplikationen weniger als N^3 Kosten.

Hinweis: Wählt man n um den Faktor vor N^3 zu optimieren ergibt sich $\frac{1}{216}$ für $n = 6$.

(c) Es gibt $n-1$ mögliche äußerste Multiplikationen, nämlich lässt sich die Folge zwischen jedem Paar von Matrizen in zwei Hälften Teilen, deren Produkte zuvor berechnet werden. Die optimalen Kosten lassen sich ermitteln, indem alle $n-1$ Optionen verglichen werden nachdem die optimalen Kosten für alle Hälften rekursiv berechnet wurden.

S = new Integer[n][n]
initialisiere S auf *undefiniert*
function COST(i, j)
  if S[i][j] *undefiniert* **then**
    if $i \geq j$ **then**

```
                S[i][j] = 0
            else
                int min = ∞
                for k = i to j − 1 do
                    int c = p[i − 1] ∗ p[k] ∗ p[j] + COST(i, k) + COST(k + 1, j)
                    if c < min then
                        min = c
                    end if
                end for
                S[i][j] = min
            end if
        end if
        return S[i][j]
    end function
```

Es gibt maximal n^2 definierende Aufrufe. Definierende Aufrufe haben Kosten in $\mathcal{O}(n)$.

Nicht definierende Aufrufe entstehen nur durch definierende Aufrufe. Da ein definierender Aufruf maximal n Aufrufe verursacht ist die Anzahl der nicht definierenden Aufrufe maximal n^3. Nicht definierende Aufrufe haben konstante Kosten.

Die Laufzeit des Algorithmus ist also in $\mathcal{O}(n^3)$.

(d) Wir nutzen die selbe Vorgehensweise wie in Aufgabe 1.

```
S = new Tree[n][n]
initialisiere S auf undefiniert
function TREE(i, j)
    if S[i][j] undefiniert then
        if i = j then
            S[i][j] = Tree(null, null, cost = 0)
        else
            int min = ∞
            for k = i to j − 1 do
                left = TREE(i, k)
                right = TREE(k+1, j)
                int c = p[i − 1] ∗ p[k] ∗ p[j] + left.cost + right.cost
                if c < min then
                    min = c
                    S[i][j] = Tree(left, right, cost = c)
                end if
            end for
        end if
    end if
    return S[i][j]
end function
```

Die asymptotische Laufzeit und rekursive Funktionsweise des Algorithmus ändert sich nicht.