# Aufgabe 1: Flohmarkt in Langdorf

Teilnahme-ID: 59521

Bearbeiter/-in dieser Aufgabe:
Christoph Waffler

11. April 2021

## Inhaltsverzeichnis



## Lösungsidee

Ziel dieser Aufgabe ist es, mehrere Stände so anzuordnen, dass zum einen sich keine Stände untereinander oder mit der Straßenbegrenzung überschneiden, zum anderen, dass die Einnahmen des Organisators möglichst hoch sind.

Das Problem wird definiert durch die 2 Dimensionen: Zeit und Platz (Platz an sich nur eindimensional)

Dies Problem weist Ähnlichkeiten zu zwei anderen Problemen auf, die sich jeweils mit einer dieser Dimensionen beschäftigen.

Das *Weighted Job Scheduling Problem* befasst sich mit der Aufgabe gewichtete Jobs so anzuordnen, dass diese einen möglichst hohen Wert erzielen. Dabei dürfen sich die Jobs nicht überschneiden, d.h. es dürfen auch keine Jobs „parallel“, sprich gleichzeitig ausgeführt werden.[1]

Das *Rucksackproblem* befasst sich hingegen mit der anderen Dimension, nämlich dem effizienten Packen eines Rucksacks/beschränkten Platz vergleichbar zu hier in dieser Aufgabe die Straße von 1000 m.
Hierbei profitiert man von der Tatsache, dass optimale Teilergebnisse vorliegen. So kann man mithilfe von Dynamischer Programmierung diese Aufgabe meist in pseudo-polynomieller Zeit lösen, obwohl dieses Problem ein NP-Problem ist.

[1] https://www.geeksforgeeks.org/weighted-job-scheduling/

Doch was machen wir nun bei unserer Aufgabe, bei der beide Probleme in Kombination auftreten?

Nun, wir können nach dem „Greedy"-Prinzip handeln und versuchen, den Profit zu maximieren:

Wir legen unsere Stände nach dem Prinzip des *Job Scheduling Problems* an, und versuchen dies solange wiederholt auszuführen, bis die Straße (2. Dimension) voll ist.
Somit schichten sich die Stände aufeinander.

Im Folgenden wird nun die Lösung zum *Weightend Job Scheduling Problem* genauer dargestellt und erläutert wie diese Lösungsidee auch bei unserer Aufgabe konkret Anwendung findet.

- Gegeben sei eine Menge von Ständen mit Start- *s* und Endzeit *f* und einer dazugehörigen Gewichtung. Für diese Aufgabe wurde als Gewichtung der Profit *p* gewählt, der durch Auswählen dieses Standes *i* erzielt werden kann: $p = (f_i - s_i) * Breite_i$
- Ziel ist es, eine Teilmenge mit der maximalen Gewichtung zu bestimmen, bei der jeder Stand disjunkt ist, d.h. kein Stand tritt gleichzeitig auf.
- Idee (unter Verwendung mit Dynamischer Programmierung): Wir nehmen an, dass wir nur auf den letzten Stand achten, vorausgesetzt diese sind nach der Endzeit sortiert. Dann entscheiden wir, ob wir diesen auswählen, oder nicht.
  Fallunterscheidung:
  - Auswahl des aktuellen Stands: Wir müssen nun alle Stände entfernen, die sich überlappen. Dann müssen wir die Lösung für das Teilproblem der restlichen Stände noch lösen.
  - Nicht-Auswahl des aktuellen Stands: Dann müssen wir die optimale Lösung für das übrige Teilproblem mit *n – 1* Ständen berechnen.

  → Dies kann rekursiv für $i = 1, 2, \dots, n$ ausgeführt werden, wobei wir die Funktion $OPT(i)$ als maximalen Wert (oder auch Gewichtung) definieren, der durch Auswahl der Stände im Intervall $[0; i]$ erreicht werden kann.
- Entfernen der sich überlappenden Intervalle:
  Die Funktion $p(i)$ gibt den größten Index *j* zurück, wobei $f_j \leq s_i$ noch eingehalten werden soll. Das heißt, dass $p(i)$ den Index des Standes zurückgibt, der gerade noch vor dem Stand *i* auftreten kann.
  Zum Beispiel kann Stand *j* von 10 bis 15 Uhr seinen Stand benutzen und Stand *i* löst ihn dann um 15 Uhr ab.
- Folgende Rekursionsformel drückt beide oben genannten Fälle zusammengefasst aus, wobei immer der günstigste Fall ausgewählt wird. Also derjenige mit der größten Gewichtung/Wert.
  $$OPT(i) = \max\{w_i + OPT(p(i)), OPT(i-1)\} \forall\, i \in \{1, \dots, n\}$$
  mit $n :=$ "$Anzahl\ der\ Stände$"
  → nicht zu vergessen ist der *base case* $OPT(0) = 0$

  → maximaler Wert ist der Ausführung in $OPT(n)$ gespeichert

- Ausgabe der Lösung:
  Mithilfe von *Backtracking* ist die optimale Lösung für diese Aufgabe wieder zu finden.
  Falls der Fall $w_i + OPT(p(i)) \geq OPT(i-1)$ eintritt, so wurde der Stand *i* zur optimalen Lösung hinzugefügt und wir können weiter mit $OPT\big(p(i)\big)$ fortfahren.
  Tritt dieser Fall nicht ein, so ignorieren wir *i* einfach und betrachten das Teilproblem $OPT(i-1)$.

- Zeitkomplexität
  - Sortieren nach Endzeiten: $O(n \log n)$
  - Herausfiltern derjenigen Stände die nichtmehr in die Straße passen: $O(n \log n)$ unter Verwendung eines Segmentbaumes, der die maximale Höhe in einem Intervall speichert
  - Finden des vorherigen Stands (Vorgängers) $p(i)$ für $i = 1, 2, \dots, n$ hat die Zeitkomplexität von $O(n \log n)$ unter der Verwendung von Binärer Suche
  - Finden des maximalen Gewichts, nachdem *p* für alle bestimmt wurde: $O(n)$
  - *Backtracking,* um die optimalen Stände zu finden: $O(n)$

  → gesamt liegt somit die obere Schranke für das *Weighted Job Scheduling Problem* in $O(n \log n)$

Die Lösung für das *Weighted Job Scheduling Problem* wird als Lösung für das Problem *Flohmarkt in Langdorf* wiederholt ausgeführt, wobei jedes Mal eine Schicht von nicht-parallelen Ständen aufeinander gelegen wird, solange noch Platz für diese in der Straße ist.

Die Laufzeit für das gesamte Programm mit Länge *L* liegt bei $O(L * n \log n)$, da die adaptierte Lösung für *Weighted Job Scheduling Problem* maximal *L* mal aufgerufen wird, und für keine Länge zweimal aufgerufen werden kann, da entweder keine Stände gesetzt wurden, oder keine Stände mehr gesetzt werden können.

# Umsetzung

Die Lösungsidee wurde in C++ unter MacOS 11.2.3 umgesetzt.

*Falls das Programm nicht ausführbar ist, einfach mit „ g++ -std=c++11“ kompilieren.*

Zu Beginn wird ein Vector aus tuple<int, int, int> eingelesen, der die Start- und Endzeit, sowie die Länge des Standes aus der Eingabe enthält.
Dieser Vector wird entsprechen nach seinem 2. Wert, also der Endzeit sortiert. Ist diese bei zwei Elementen gleich, so entscheidet, die 3. Größe (die Länge des Standes).

Die Gewichtung eines Standes ist später beim *Weighted Job Scheduling Problem* entscheidend und wurde hier auf $länge_{stand} * dauer_{stand}$ gesetzt.

So wird später beim Auswählen des Vorgängers, derjenige Stand ausgewählt der eher mehr Geld einbringt.

Nachdem der Vector sortiert wurde, wird ein Segmentbaum erstellt, der für jede Stunde bzw. für die Zeitintervalle speichert, wie viel maximal von der Straße schon verbraucht wurde, bzw. wie viel von der Straße noch übrig ist.
Durch die Verwendung eines Segmentbaums kann später die aktuelle Länge der Straße zu einem bestimmten Zeitabschnitt in $O(\log(T))$ verändert werden, bei *T* Zeitabschnitten.
Würde man ein lineares Array verwenden, so würde man $O(T)$ Zeitbenötigen, was sich bei größeren Zeiteingaben durchaus bemerkbar macht.

Wichtig ist beim Segmentbaum noch, dass dieser erst zu einer 2er Potenz aufgefüllt wird, falls das gesamte Zeitintervall keine Potenz von 2 ist, z.B. 9, *hier würde das Programm mit 5 leeren Feldern zu* $2^4$ *auffüllen.*

*Außerdem: Die Zeitangaben können beliebig eingegeben werden um müssen nicht im Bereich von 8 Uhr bis 18 Uhr liegen*.

Folgende while-Schleife wird solange wiederholt, bis sie mit *break* verlassen wird → *while(true)*:

- Im letzten Durchlauf verwendete Stände werden nun herausgefiltert und nicht zum neuen Array hinzugefügt, der alle nun verfügbaren Stände speichert.
  Dasselbe gilt auch für jene Stände deren Höhen nicht mehr in die Straße passen. Dies wird festgestellt, indem auf dem Segmentbaum eine Intervallabfrage für den zeitlichen Bereich von der Startzeit bis zur Endzeit des aktuellen Standes gestellt wird. Diese Abfrage gibt die maximale verbrauchte Länge in diesem Intervall zurück.

- Nun folgt die Implementierung der Lösung für das *Weighted Job Scheduling Problem* wie bereits in der Lösungsidee erwähnt.
  Hier wird jeden Stand mit dem Index 0 bis n (n steht für die Anzahl der Stände) jeder näheste Vorgänger bestimmt, der gerade so mit seiner Endzeit nicht die aktuelle Startzeit überschreite.

- Anschließend wird mit der rekursiven Methode *getOpt(i)* für jedes *i* der optimale Wert berechnet, so wie in der Lösungsidee erläutert.

- Via *Backtracking* lassen sich alle Stände herausfinden, die zur optimalen Lösung gehören.
  Für alle Stände *i* die zur Lösung gehören, wird im Segmentbaum *b* die verbrachte Straße im Intervall $[start_i; end_i]$ auf den Wert $b[start_i; end_i] = \max(b[start_i; end_i]) + länge_i$ gesetzt.
  Das heißt, dass das gesamte Intervall von *start_i* bis *end_i* im Segmentbaum *b* auf die Summe aus dem *bisherig maximalen Wert dieses Intervalls + die Länge des Standes i* gesetzt wird.

- Die whlie-Schleife beginnt von neuem.

- Die while-Schleife wird abgebrochen, falls entweder keine Stände mehr da sind, im letzten Durchlauf keine Stände verbraucht wurden und dabei auch keine übrig geblieben sind.

Am Ende des Programms werden noch die gesamten Mietennahmen ausgegeben, die sich aus der Summe der Lösungen der „aufeinandergestapelten" Lösungen des *Weighted Job Scheduling Problems* zusammensetzten, die wiederum die Gewichtungen für jeden Stand aus $länge_{stand} * dauer_{stand}$ sind.

# Beispiele

## Beispiel der BWINF Webseite

→ Die Ausgaben des Programms für die Aufgaben von der BwInf-Website sind in einem extra Ordner mitgesendet.

# Auswertung

Bei flohmarkt6.txt fällt bspw. auf, dass die Lösung des Programms nicht hundertprozentig optimal ist, da eine andere Anordnung 9382 [€] einbringen würde.
Aufgrund der Tatsache, dass mein Programm greedy arbeitet und mit der Heuristik (der Vorsortierung) dieses Problem schnell löst, sind die Abstriche in der Optimalität der Lösung zu verkraften.

Um eine hundertprozentig optimale Lösung erreichen zu können, müsste man nahezu alle Möglichkeiten ausprobieren, was bei einer großen Eingabe sehr sehr lange dauern würde.

# Quellcode

```
#include <bits/stdc++.h>
using namespace std;

typedef vector<int> vi;
typedef vector<vi> vvi;
typedef tuple<int, int, int> tiii;

#define what(x) cerr<<#x<<" is "<<x<<endl;

const int INF = numeric_limits<int>::max()/2;

// gesamte Anzahl aller Stände, die sich angemeldet haben
int N;

// Länge der Straße
int maxL;
```

```
// Größe des Segmentbaumes, die im Laufe des Programms noch ermittelt wird
int sizeSegmentTree;

// neutrales Element, das im Segmentbaum benötigt wird,
// dessen assoziative Operation 'max' ist.
// analog dasselbe auch für den min_segment_tree mit INF als neutrales Element
const int max_neutral_element = -INF, min_neutral_element = INF;

// Vector, in dem der Segmentbaum gespeichert wird (eindimensional)
// und dessen dazugehörige, bereits ausgeführte Operationen des jeweiligen Knotens
// z.B. werden später alle Elemente in einem Intervall um einen bestimmen Wert erhöht werden
müssen;
// dies wird im Array operation für alle direkten und indirekten Kinderknoten zwischengespeichert

// Es wird ein Segmentbaum für das Minimum der Intervalle und eines für das Maximum benötigt
vi max_segment_tree, operation;

// neutrale Operation für die apply Methode, die addiert --> 0 ist neutral
int id = 0;

vi starting_times, ending_times, weights;

vi optimal, pre;


// Wert des Einkommens (ganz am Ende des Programms)
int INCOME;

bool sortBySecondElseThird(const tiii& a, const tiii& b) {
    int a1, a2, a3, b1, b2, b3;
    tie(a1, a2, a3) = a;
```

```
    tie(b1, b2, b3) = b;

    if (a2 == b2) {
        return a3 < b3;
    }
    return a2 < b2;
}

// Methode zum bestimmen des am weitesten rechten Index, an dem der Wert value in das Array arr
// eingefügt werden müsste, damit es sortiert bleiben würde

// mit binärer Suche in log(n) Zeit, da das Array arr bereits sortiert ist
int binary_search(const vi& arr, const int value) {
    // Zu Beginn: obere Grenze N-1
    // untere Grenze 0
    int up = N-1, low = 0;

    // solange obere Grenze noch oberhalb
    // der unteren Grenze ist, läuft die Such noch;
    // ansonsten wird der Index zurückgegeben, wo value eingefügt werden muss
    while (low < up) {

        // Mittelwert neu bilden
        int mid = floor((low + up)/2);

        if (value < arr[mid]) {
            up = mid;
        }
        else {
            low = mid + 1;
        }
```

```
    }

    // dasselbe Element wurde nicht in arr gefunden,
    // daher wird der Index zurückgegeben, wo es eingefügt werden müsste
    return low-1;
}




// Hilfsmethoden für den Segmentbaum
int right(int i) {
    return 2*i+1;
}
int left(int i) {
    return 2*i;
}
int parent(int i) {
    return floor(i/2);
}

// Erstellen des Segmentbaums
// initiale Arrays für Segmentbaum mit maximalen Intervallwerten
void build(const vi& max_arr) {
    // 2-fache Größe die für den Segmentbaum berechnet wurde,
    // wird für den Segmentbaum benötigt
    max_segment_tree.resize(2*sizeSegmentTree);
    operation.resize(2*sizeSegmentTree);

    // hintere Hälfte des Segmentbaums initialisieren,
    // welcher alle Werte des übergebenen Arrays arr enthält
```

```
    for (int i = sizeSegmentTree; i < sizeSegmentTree*2; ++i) {
        max_segment_tree[i] = max_arr[i - sizeSegmentTree];

        // Initialisierung zu Beginn mit dem neutralen Element id
        operation[i] = id;
    }

    for (int i = sizeSegmentTree-1; i > 0; --i) {
        // Maximum bzw. Minimum von den 'vorausgehenden Knoten' l und r wird gebildet
        // und in i gespeichert
        max_segment_tree[i] = max(max_segment_tree[left(i)], max_segment_tree[right(i)]);

        // neutrales Element id wird zugewiesen
        operation[i] = id;
    }
}


// folgende Methode aktualisiert beide Segmentbäume (min und max):
// In der Apply-Methode des Segementbaums werden alle Elemente in dem übergebenen Bereich (von
l bis r)
// auf den übergebenen Wert gesetzt
// da die Operation 'Gleichsetzen' kommutativ ist, wird im Segmentbaum Lazy Propagation benötigt
void apply(int l, int r, int x, int y=id, int a=1, int b=sizeSegmentTree, int i=1) {
    max_segment_tree[i] = y;

    y += operation[i];
    // y = operation[i];
    operation[i] = id;

    if (l <= a and b <= r) {
```

```
        operation[i] = x;
        y = x;
        max_segment_tree[i] = x;
        return;
    }

    // aktueller Knoten nicht mehr im angegebenen Intervall
    if (r < a or b < l) {
        operation[i] = y;
        return;
    }

    // Mittelwert bilden
    int m = (a + b)/2;

    // linker und rechter Kinderknoten werden rekursiv aufgerufen,
    // damit deren Wert auch aktualisiert werden,
    // solange sie auch im angebenen Intervall befinden
    apply(l, r, x, y, a, m, left(i));
    apply(l, r, x, y, m+1, b, right(i));

    max_segment_tree[i] = max(max_segment_tree[left(i)], max_segment_tree[right(i)]);

    return;
}

// Abfrage im Segmentbaum nach dem Ergebnis im Intervall von l bis r
// --> Abfrage des maximalen Elements darin
// Verwendung der binären Suche --> log(n) Zeit

// Zu Beginn des rekursiven Aufrufs bei der Wurzel des Segmentbaums
```

```
// anschlie0end wird der Baum herabgestiegen,
// um Segmente zu finden, die das Intervall [a; b] aufteilen

int max_seg_query(const int& l, const int& r, int a=1, int b=sizeSegmentTree, int i=1) {
    if (l == r and l == 0) {
        return 0;
    }

    // Aktueller Knoten i steht für den Bereich [a; b]

    // Falls [a; b] ein Teilsegment des gesuchten Bereichs ist
    // -> Rückgabe des aktuellen Werts des Knotens
    if (l <= a and b <= r) {
        return max_segment_tree[i];
    }

    // aktueller Knoten für den Bereich [a; b] nicht mehr im angegebenen Intervall
    if (r < a or b < l) {
        return max_neutral_element;
    }

    // Mittelwert bilden
    int m = (a + b) /2;

    // maximaler Wert des linken und rechten Kinderknoten ermitteln,
    // dabei muss auch noch der Wert von operation[i] hinzugefügt werden
    return operation[i] + max(max_seg_query(l, r, a, m, left(i)), max_seg_query(l, r, m+1, b, right(i)));
}

int getOpt(const int& i) {
    if (i == -1) return 0;
```

```
    else if (0 <= i and i < optimal.size()) {
        return optimal[i];
    }

    else {
        // den maximalen Wert zurückgeben,
        // entweder den aktuellen Wert zusammen mit dem Wert des passenden Vorgängers von i
        // oder i einfach übernehmen, d.h. der akutelle Stand i wird nicht gesetzt
        return max(weights[i] + getOpt(pre[i]), getOpt(i - 1));
    }
}


int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    // maximale Straßenlänge ist frei wählbar
    // cout<<"> Bitte geben Sie die Straßenlänge in Meter ein\n"<<flush;
    // cin >> maxL;

    // oder:
    maxL = 1000;

    INCOME = 0;

    cin >> N;

    // Speicher für benutzte Stände
    vi used_elements;
```

```
used_elements.resize(N, false);

vector<tiii> arr(N);
for (int i = 0; i < N; ++i) {
    // Beginn, Ende und Länge werden eingelesen
    int a, b, c;
    cin >> a >> b >> c;

    arr[i] = {a, b, c};
}

// Sortieren nach ihrem Endzeitpunkt
// bei gleichem Endzeitpunkt, ist die Länge entscheidend
sort(arr.begin(), arr.end(), sortBySecondElseThird);

// die Start- und Endzeiten werden in Arrays gespeichert,
// nachdem sie sortiert wurden
// vi starting_times(N), ending_times(N);
starting_times.resize(N);
ending_times.resize(N);
weights.resize(N);

int minStartTime = INF, maxEndTime = 0;

for (int i = 0; i < N; ++i) {
    int start, end, length;
    tie(start, end, length) = arr[i];

    // Einkommen durch den Stand wird berechnet
```

```
    int weight = (end - start) * length;

    starting_times[i] = start;
    ending_times[i] = end;
    weights[i] = weight;

    minStartTime = min(minStartTime, start);
    maxEndTime = max(maxEndTime, end);
}

// + + + Segmentbaum + + + //

// Unterteilung Stundenweise, beginnend von minStartTime bis maxEndTime
// somit ist intervallSize = 1 (Stunde)
int intervallSize = 1;


// Array speichert die aktuelle Höhe zum aktuellen Intervall
int sizeTimeInterval = (maxEndTime - minStartTime) / intervallSize;


// es kann sein, das sizeTimeInterval keine 2er Potenz ist,
// daher werden noch zusätzliche leere Speicherplätze mit hinzugefügt,
// damit die Größe des Segmentbaums eine 2er Potenz
int potenz = ceil(log2(sizeTimeInterval));

// rest --> auffüllen
sizeSegmentTree = pow(2, potenz);

int rest = sizeSegmentTree - sizeTimeInterval;

// Erstellen des Segmentbaumes, mit einem leeren Array,
```

```
    // da noch keine Stände festgelegt wurden
    build(vi(sizeSegmentTree, 0));


    cout << "> Folgende Stände sind eingeplant (im Format Start- und Endzeitpunkt + deren Länge \n";

    int counterUsed = 0;
    int counterUnpossible = 0;

    bool firstTime = true;

    while(true) {

        counterUsed = 0;
        counterUnpossible = 0;
        vector<tiii> newArr;
        vi newStarting_times, newEnding_times, newWeights;
        for (int i = 0; i < N; ++i) {

            if (used_elements[i]) {
                counterUsed++;
            }
            else {
                int& length = get<2>(arr[i]);

                int usedheight = max_seg_query(starting_times[i]-minStartTime, ending_times[i]-
minStartTime-1);

                if (usedheight + length > maxL) {
                    // kein Platz mehr für das aktuelle Elemente
                    counterUnpossible++;
```

```
        }
        else {
            newArr.push_back(arr[i]);
            newStarting_times.push_back(starting_times[i]);
            newEnding_times.push_back(ending_times[i]);
            newWeights.push_back(weights[i]);
        }
    }
}

N -= counterUsed;
N -= counterUnpossible;

used_elements.clear();
used_elements.resize(N, false);

starting_times = newStarting_times;
ending_times = newEnding_times;
weights = newWeights;

if (N == 0) {
    break;
}
else if (!firstTime) {
    if (counterUnpossible <= 0 and counterUsed <= 0) {
        break;
    }
}

// + + + Weighted Job Scheduling Problem + + + //
```

```
// Array das für den einen Stand i denjenigen Stand pre[i] speichert,
// der vor dem akutellen Stand i endet,
// damit die Endzeit von pre[i] < starting_times[i] gilt
pre.resize(N);

for (int i = 0; i < N; ++i) {
    // Index des vorherigen Standes wird gesucht,
    // dessen Endzeit noch vor der Startzeit des aktuellen Stands ist
    int index_pre = binary_search(ending_times, starting_times[i]);
    // index verringern, damit das Element gefunden wird,
    // das noch vor der aktuellen Startzeit endet
    // index_pre --;

    pre[i] = index_pre;
}


// base case für Dynamische Programmierung (rekursiv)
// optimal[0] = 0;
for (int i = 0; i < N; ++i) {
    int opt_i = getOpt(i);
    optimal.push_back(opt_i);
}

// was nun geschieht:
// - eine optimale Reihe wurde gefunden,
//  -> diese wird nun gespeichert
// - und die Höhen im Segmentbaum werden aktualisiert

vi solution;
```

```
// Backtracking der Lösung; (Lösung = Anordnung der Stände)
for (int i = N-1; i >= 0;) {
    int following_getOpt;
    if (i > -1) following_getOpt = getOpt(i - 1);
    else following_getOpt = 0;

    if (weights[i] + getOpt(pre[i]) > following_getOpt) {
        solution.push_back(i);
        // cout << "sol\n";

        used_elements[i] = true;

        // es wird mit dem Vorgänger von i fortgesetzt
        i = pre[i];
    }
    else --i;
}

INCOME += optimal[N-1];
optimal.clear();


// Erhöhung der Höhen
for (int i : solution) {
    int start = starting_times[i];
    int end = ending_times[i];
    int weight = weights[i];

    int& length = get<2>(arr[i]);

    cout << start << " " << end << " " << length << "\n";
```

```
            int aktHeight = max_seg_query(start-minStartTime, end-minStartTime-1);
            if (aktHeight<0) aktHeight = 0;

            // Segmentbaum für das Intervall [start; end] auf length + aktHeight setzen
            // length entspricht der Breite des aktuellen Standes
            // das Ende wird nicht eingeschlossen, sprich nicht erhöht,
            // da es sich sonst mit der anschließenden Höhe eines Startzeitpunkt s, s=end, doppelt erhöht
werden würde,
            // obwohl dies in der Realität nicht der Fall wäre --> nachfolgender Stand mit end1=start2
            apply(start-minStartTime, end-minStartTime-1, length+aktHeight);

        }

        firstTime = false;

    }

    cout << ">> gesamtes Einkommen beträgt " << INCOME;

    return 0;
}
```