# 1. Lösungsidee

Ich möchte mit dem Wichtigsten und Allgemeinsten zuerst anfangen:
Der Kauf bzw. Verkauf von Gegenständen kostet nichts, deshalb ist die jeweilige Zusammensetzung der Gegenstände, die der Verein besitzt, nicht relevant für die Zusammensetzung des Inventars im nächsten Monat, da der Verein beliebig Gegenstände umtauschen kann. Wichtig ist einzig und allein, der Gesamtwert der Gegenstände, die der Verein besitzt.
Daraus folgt, dass man zur Bestimmung der Gegenstände, die der Schatzmeister von einem Monat auf den anderen kaufen oder verkaufen soll, nur aus der Gesamtmenge an Gegenständen die Teilmenge bestimmen muss, die am nächsten an den Wert der Gegenstände + Budget herankommt, also das Wenigste vom Monatsbudget verschwendet.

Dies ist schon ein wichtiger Schritt zur Lösung, da er das Problem sehr vereinfacht. Jetzt muss man nur noch die größte Teilmenge bestimmen, deren Gesamtwert den alten Gesamtwert + Budget nicht überschreitet.

Die einfachste Strategie ist es, alle Teilmengen zu bestimmen und sie nach ihrem Gesamtwert zu sortieren. Dies würde für das Aufgabenbeispiel ausgezeichnet funktionieren, aber bei den Beispielen im Internet gäbe es sehr große Probleme. Eines der Beispiele enthält 1000 Gegenstände. Die Anzahl der Teilmengen beträgt dann 2^1000, eine Zahl, die man sich nur schwer vorstellen kann und die es keines Falles erlauben würde, den Akquisitionsplan in ausreichender Zeit zu bestimmen.

Ich benutze aus Mangel an besseren Algorithmen, einen sehr einfachen Ansatz, der aber zumindest bei 4 der 5 Beispiele (Beispiel 0, 1, 3 und 4) optimale Lösungen liefert, da immer das Restgeld = 0 bleibt (der letzte Datensatz ist dabei egal) und bei dem anderen Beispiel ist das nicht so offensichtlich und konnte daher von mir nicht überprüft werden:
Ich verkaufe sozusagen alle Gegenstände und habe dadurch den Gesamtpreis + Budget Geld zur Verfügung.
Die Gesamtliste der Gegenstände wird dann nach dem Preis absteigend sortiert und ich versuche so viele, teure Gegenstände wie möglich zu kaufen, bis das Geld verbraucht ist, bzw. alle Gegenstände einen höheren Preis als das Restgeld haben.
Aus einem Vergleich der alten Liste mit der neuen Liste der Gegenstände, kann man feststellen, welche Gegenstände gekauft und welche verkauft worden sind:
alte Liste \ neue Liste ("ohne"): verkaufte Gegenstände
neue Liste \ alte Liste: gekaufte Gegenstände

Der Algorithmus ist dann beendet (bzw muss beendet werden, damit keine Endlosschleife beginnt), wenn keine neuen Gegenstände mehr gekauft werden, also das monatliche Budget nicht angegriffen wird, also Restgeld >= Budget ist.

Die nächste Frage, die sich stellt, ist: Findet der Algorithmus immer eine Lösung, wenn auch der oben beschriebene Brute-Force-Ansatz eine liefern würde?
Kurzer Beweis:
Nehmen wir an, der Brute-Force-Algorithmus findet eine Lösung und der obere nicht, dann würde das bedeuten, dass der Algorithmus wegen der oberen Abbruchbedingung eine Zustand erreicht, in dem zwar noch Gegenstände übrig geblieben sind, diese aber nicht gekauft werden können, weil Gesamtpreis + Budget nicht ausreicht. D.h. es wurde bisher die alte Liste an Gegenständen erworben und für günstigere reicht das Geld nicht mehr. Sei X der Preis des günstigsten Gegenstandes, der nicht gekauft werden kann. Dann müsste X > Budget sein, da aber mit dem teuersten Gegenstand beim Einkaufen angefangen wird, also keine Gegenstand günstiger als X erworben worden sein konnte, muss X auch der Preis des günstigsten Elementes sein. Da aber X > Budget wäre, und alle anderen Gegenstände teurer sind, kann auch der Brute-Force-Algorithmus keine Lösung

gefunden haben, da ja von Anfang an kein einziger Gegenstand erworben worden sein hätte können.

Zumindest findet dieser Algorithmus also immer eine Lösung, wenn es sie gibt. Mir ist leider kein Beweis eingefallen, mit dem gezeigt werden könnte, das der Algorithmus optimale Lösungen liefert und ich bin mir dessen auch nicht sicher. Dennoch scheint er ein guter Kompromiss zwischen Leistung und Ergebnis zu sein.

## 2. Programm-Dokumentation

Das Programm verwendet STL und templates um die Quellcode-Größe klein zu halten und es funktioniert genau so wie in der Lösungsidee beschrieben. Die Funktion MultiSetWithoutMultiSet ist eine Hilfsfunktion, die einfach A \ B in STL-Code beschreibt.
FindOptimalSubMultiSet beinhaltet den oben beschriebenen Algorithmus und liefert den Betrag des nicht-verwendeten Restgeldes zurück.
In main behelfe ich mir eines Tricks um den neuen Gesamtwert der Gegenstände zu bestimmen:
Da ich den alten Gesamtwert + Budget kenne, ist der neue Gesamtwert = alter Gesamtwert + Budget - Restgeld.

Das Progamm entnimmt dem ersten Befehlszeilenparameter den Dateinamen des Datensatzes und liest sie den Bespielen im Internet zufolge aus:
1. Zeile: Budget
Folgende Zeilen: Preise der Gegenstände

## 3. Programm-Ablaufprotokoll:

```
[...]>"Aufgabe 1.exe" beispiel0.txt
1. Verkauft: nichts Gekauft: 790 Gesamtwert: 790 Verloren: 210
2. Verkauft: 790 Gekauft: 340 1320 Gesamtwert: 1660 Verloren: 130
3. Verkauft: 1320 Gekauft: 2100 Gesamtwert: 2440 Verloren: 220
4. Verkauft: 340 Gekauft: 1320 Gesamtwert: 3420 Verloren: 20
5. Verkauft: nichts Gekauft: 790 Gesamtwert: 4210 Verloren: 210
6. Verkauft: 790 1320 2100 Gekauft: 5200 Gesamtwert: 5200 Verloren: 10
7. Verkauft: nichts Gekauft: 790 Gesamtwert: 5990 Verloren: 210
8. Verkauft: 790 Gekauft: 340 1320 Gesamtwert: 6860 Verloren: 130
9. Verkauft: 1320 Gekauft: 2100 Gesamtwert: 7640 Verloren: 220
10. Verkauft: 340 Gekauft: 1320 Gesamtwert: 8620 Verloren: 20
11. Verkauft: nichts Gekauft: 790 Gesamtwert: 9410 Verloren: 210
12. Verkauft: nichts Gekauft: 670 Gesamtwert: 10080 Verloren: 330
13. Verkauft: nichts Gekauft: 340 Gesamtwert: 10420 Verloren: 660

[...]>"Aufgabe 1.exe" beispiel1.txt
1. Verkauft: nichts Gekauft: 1 Gesamtwert: 1 Verloren: 0
2. Verkauft: 1 Gekauft: 2 Gesamtwert: 2 Verloren: 0
3. Verkauft: nichts Gekauft: 1 Gesamtwert: 3 Verloren: 0
.
.
.
131068. Verkauft: 1 2 Gekauft: 4 Gesamtwert: 131068 Verloren: 0
131069. Verkauft: nichts Gekauft: 1 Gesamtwert: 131069 Verloren: 0
131070. Verkauft: 1 Gekauft: 2 Gesamtwert: 131070 Verloren: 0
131071. Verkauft: nichts Gekauft: 1 Gesamtwert: 131071 Verloren: 0

[...]>"Aufgabe 1.exe" beispiel2.txt
1. Verkauft: nichts Gekauft: 8135 Gesamtwert: 8135 Verloren: 1865
2. Verkauft: 8135 Gekauft: 7567 10211 Gesamtwert: 17778 Verloren: 357
3. Verkauft: 7567 10211 Gekauft: 26530 Gesamtwert: 26530 Verloren: 1248
```

```
.
.
.
197. Verkauft: 8135 Gekauft: 7567 10211 Gesamtwert: 1688315 Verloren: 357
198. Verkauft: nichts Gekauft: 8135 Gesamtwert: 1696450 Verloren: 1865
199. Verkauft: nichts Gekauft: 7362 Gesamtwert: 1703812 Verloren: 2638

[...]>"Aufgabe 1.exe" beispiel3.txt
1. Verkauft: nichts Gekauft: 50 Gesamtwert: 50 Verloren: 0
2. Verkauft: 50 Gekauft: 100 Gesamtwert: 100 Verloren: 0
3. Verkauft: nichts Gekauft: 50 Gesamtwert: 150 Verloren: 0
4. Verkauft: 50 Gekauft: 100 Gesamtwert: 200 Verloren: 0
.
.
.
1023. Verkauft: 1 Gekauft: 3 6 6 6 6 6 6 6 6 Gesamtwert: 51150 Verloren: 0
1024. Verkauft: 3 Gekauft: 2 5 5 5 5 5 5 5 5 5 6 Gesamtwert: 51200 Verloren: 0
1025. Verkauft: 2 Gekauft: 3 4 4 4 4 4 5 5 5 5 5 Gesamtwert: 51250 Verloren: 0
1026. Verkauft: nichts Gekauft: 1 1 1 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4
Gesamtwert: 51295 Verloren: 5

[...]>"Aufgabe 1.exe" beispiel4.txt
Es gibt keine Lösung für diesen Datensatz!
```

## 4. Quellcode

```
#include <stdio.h>
#include <algorithm>
#include <set>
#include <map>
#include <iterator>

typedef signed long Currency;
typedef std::multiset< Currency > CurrencyMultiSet;
typedef std::map< Currency, CurrencyMultiSet > CurrencyMultiSetMap;

struct DataSet {
	Currency monthlyBudget;
	CurrencyMultiSet productPrices;

	DataSet( const char *filename );
};

DataSet::DataSet( const char *filename ) : monthlyBudget( 0 ) {
	FILE *file = fopen( filename, "r" );
	if( !file ) {
		return;
	}
	if( fscanf( file, "%i", &monthlyBudget ) == 1 ) {
```

```
		while( 1 ) {
			unsigned price;
			if( fscanf( file, "%i", &price ) == EOF )
				break;
			productPrices.insert( (Currency) price );
		}
	}
	fclose( file );
}

Currency FindOptimalSubMultiSet( const CurrencyMultiSet &elements, const Currency &limit, CurrencyMultiSet &subset ) {
	Currency moneyLeft = limit;
	for( CurrencyMultiSet::const_reverse_iterator iter = elements.rbegin() ; iter != elements.rend() ; iter++ ) {
		const Currency price = *iter;
		if( price <= moneyLeft ) {
			subset.insert( *iter );
			moneyLeft -= price;
		}
		if( moneyLeft < 0 ) {
			break;
		}
	}
	return moneyLeft;
}

CurrencyMultiSet MultiSetWithoutMultiSet( const CurrencyMultiSet &A, const CurrencyMultiSet &B ) {
	CurrencyMultiSet out;
	std::insert_iterator< CurrencyMultiSet > &insertIterator = std::inserter( out, out.begin() );
	std::set_difference( A.begin(), A.end(), B.begin(), B.end(), insertIterator );
	return out;
}

void OutputMultiSet( const CurrencyMultiSet &elements ) {
	if( !elements.empty() ) {
		for( CurrencyMultiSet::const_iterator iter = elements.begin() ; iter != elements.end() ; iter++ ) {
			const Currency price = *iter;
			printf( "%i ", price );
		}
	} else {
```

```
                printf( "nichts " );
        }
}

int main( int argc, char *argv[] ) {
        if( argc < 2 ) {
                return -1;
        }

        DataSet dataSet( argv[ 1 ] );

        unsigned iteration = 1;
        Currency total = 0;
        CurrencyMultiSet solution[2];
        while( 1 ) {
                CurrencyMultiSet &current = solution[ iteration % 2 ];
                CurrencyMultiSet &old = solution[ (iteration - 1) % 2 ];

                total += dataSet.monthlyBudget;
                current.clear();
                Currency wasted = FindOptimalSubMultiSet( dataSet.productPrices, total, current );
                total -= wasted;
                if( wasted < dataSet.monthlyBudget ) {
                        CurrencyMultiSet bought = MultiSetWithoutMultiSet( current, old );
                        CurrencyMultiSet sold = MultiSetWithoutMultiSet( old, current );
                        printf( "%i. Verkauft: ", iteration );
                        OutputMultiSet( sold );
                        printf( "Gekauft: " );
                        OutputMultiSet( bought );
                        printf( "Gesamtwert: %i Verloren: %i\n", total, wasted );
                } else {
                        if( iteration == 1 ) {
                                printf( "Es gibt keine Lösung für diesen Datensatz!\n" );
                        } // else we're done (see the proof in info.txt)
                        break;
                }
                iteration++;
        }
        return 0;
}
```