ETH Zürich
Institut für Theoretische Informatik

Semesterarbeit
Algorithmen zur Protein-Identifikation

Christian Schlup

Studiengang Informatik, 7. Semester
cschlup@student.ethz.ch

Betreuung durch:
Prof. Dr. Peter Widmayer, widmayer@inf.ethz.ch
Mark Cieliebak, cielieba@inf.ethz.ch
Zsuzsanna Lipták, zsuzsa@inf.ethz.ch

Wintersemester 01/02

Inhaltsverzeichnis

1 Einführung **1**
1.1 Ausgangslage 1
1.2 Aufgabenstellung 1
1.3 Arbeitsumfeld 2

2 Algorithmen **3**
2.1 Linsearch 3
2.2 Binsearch 3
2.3 Lookup 4
2.3.1 Einleitendes Beispiel 5
2.3.2 Beschreibung von Lookup 6
2.3.3 Erläuterung der Nicht-Umsetzbarkeit für praktische Zwecke 7
2.4 Cluster 7
2.4.1 Beschreibung von Cluster 7
2.4.2 Beispiel 8

3 Implementierung **10**
3.1 Design des Frameworks 10
3.2 Klassendiagramm 11

4 Userinterface **13**
4.1 Bedienung des Userinterfaces 13
4.2 Transkript einer Session 14

5 Testen der Algorithmen und Beurteilung von Cluster **15**
5.1 Einleitung 15
5.2 Auswahl der Testdaten und Beschreibung der Tests 15
5.3 Interpretation des Testresultates 15
5.4 Testresultate im Überblick 16

6 Referenzen **18**

Anhang

A Source Code
B Testdaten, Testlog

Kapitel 1

Einführung

1.1 Ausgangslage

In Proteomics werden Proteine und ihre Funktionsweisen untersucht. Ein Protein besteht aus vielen aneinander geketteten Aminosäuren, die der Einfachheit halber durch einen Buchstaben repräsentiert werden. Alle Erkenntnisse über Proteine werden in Datenbanken gespeichert, die sehr gross werden können. Wenn ein neues Protein gefunden wird, möchte man wissen, ob in der Datenbank bereits Informationen über dieses Protein gespeichert sind. Dazu muss man das Protein identifizieren. Protein-Sequenzierung (das Identifizieren der Amino-Säuren-Sequenz) ist aber sehr aufwändig. Eine andere Methode besteht darin, das Protein in kleine Teile zu zerlegen und diese zu "wiegen". Dies liefert einen "Fingerprint", nach dem man in den Datenbanken suchen kann *Quelle*: [1, Zitat].Hierfür gibt es die beiden einfachen Algorithmen *Linsearch* und *Binsearch*: Der Erstere ist sehr effizient bezüglich Speicherplatz, O(1), hat aber Laufzeit O(n), wobei n die Länge des Proteins ist; der Zweite erzeugt eine zusätzliche Datenstruktur, welche $O(n^2)$ Speicherplatz benötigt, hat aber die sehr gute Laufzeit O(log n). *Quelle:* [1]

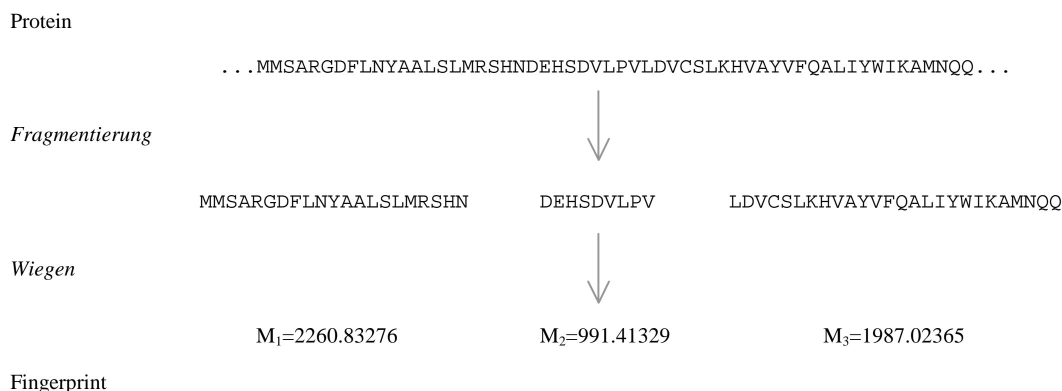


Abbildung 1: Fragmentierung eines Proteins und wiegen der Fragmente, *Quelle:* [1]

Man ist nun bestrebt, einen Algorithmus zu finden, der besser als einer dieser einfachen Algorithmen ist. Am Institut für Theoretische Informatik wurde ein Algorithmus *Lookup* entwickelt, der asymptotisch besser ist als Linsearch und Binsearch. Lookup wurde noch nicht implementiert und es kann keine Aussage darüber gemacht werden, wie sich dieser auf realen Daten verhält. Ebenso verhält es sich mit *Cluster*, einem weiteren an diesem Institut entwickelten Algorithmus.

1.2 Aufgabenstellung

Das ursprünglich definierte Ziel dieser Semesterarbeit war es, das tatsächliche Verhalten der Algorithmen Linsearch, Binsearch und Lookup auf realen Daten zu vergleichen und eine Aussage über deren Laufzeitverhalten zu machen, insbesondere über die Praxistauglichkeit von Lookup. Dies beinhaltet die Implementation der Algorithmen und das Testen auf realen Daten. Da sich nach wenigen Wochen schon abzeichnete, dass sich Lookup gegenüber Linsearch und Binsearch nicht gewinnbringend implementieren lässt, wurde darauf verzichtet und stattdessen Cluster implementiert.

1.3 Arbeitsumfeld

Ich wurde durch meine beiden Betreuer Mark Cieliebak und Zsuzsanna Lipták sorgfältig in das Thema eingeführt. Ausserdem stand mir das Paper [2] zur Verfügung. Die detaillierte Beschreibung der Algorithmen in diesem Paper erleichterte mir deren Implementation.
Die Implementierung der Algorithmen konnte ich zuhause auf meinem eigenen Rechner tätigen. Es handelt sich dabei um einen PC mit Pentium III 933MHz Prozessor und 128MB RAM Memory. Auf diesem Rechner ist das Betriebssystem *Windows XP home edition* installiert. Die Programmierung erfolgte in Java, Version 1.2.2.
Einmal wöchentlich traf ich mich mit Zsuzsa und Mark, um den Fortschritt und eventuelle Fragen bezüglich meiner Semesterarbeit zu besprechen.

Kapitel 2

Algorithmen

2.1 Linsearch

In Proteindatenbanken sind Proteine als aus Buchstaben bestehende Strings gespeichert. Jeder Buchstabe repräsentiert dabei eine Aminosäure. Der Algorithmus Linsearch vollzieht kein Preprocessing an diesen Proteinstrings. Unter Preprocessing versteht man die Aufbereitung der Daten in einer Weise, die später ein schnelleres Auffinden von Massen in einem Protein ermöglicht.
Nehmen wir an, wir suchen die Masse M in der Proteindatenbank. Linsearch durchsucht nun einen Proteinstring σ mit Hilfe von zwei Pointern l und r, die zu Beginn auf Position 1 stehen. Den Teilstring, der bei l beginnt und bei r endet, nennen wir $\sigma(l, r)$ und dessen Masse $\mu(\sigma(l, r))$. Es wird folgendermassen nach der Masse M gesucht: Ist M kleiner als $\mu(\sigma(l, r))$, bewegen wir r um eine Position nach rechts, wenn M grösser als $\mu(\sigma(l, r))$ ist, bewegen wir l um eine Position nach rechts. Falls zu einem Zeitpunkt gilt: $M = \mu(\sigma(l, r))$, ist die Masse gefunden und der Algorithmus gibt **yes** aus und terminiert. Wenn r das Ende des Proteinstrings erreicht hat und M kleiner als $\mu(\sigma(l, r))$ ist, bricht der Algorithmus ab und gibt **no** aus.
Für eine Proteinstringlänge von n ist die Laufzeit von Linsearch O(n) und der Speicherbedarf O(1). *Quelle*: [2, Übersetzung]

Beispiel: Wir betrachten das fiktive Aminosäurenalphabet $A = \{a, b, c\}$. Die Massen der einzelnen Aminosäuren a, b und c sind 1, 2 und 5. Als Proteinstring definieren wir σ = abbcaba. Die lineare Suche nach der Masse $M = 8$ bewegt zuerst den rechten Zeiger r bis Position 4, da M stets grösser ist als $\mu(\sigma(l, r))$ für $r = 1...3$. $\mu(\sigma(1, 4)) = 10$ und ist somit grösser als 8. Nun wird der Zeiger l bis Position 3 bewegt, was die aktuelle Masse auf 7 schrumpfen lässt und eine Weiterbewegung des rechten Zeigers r auslöst. Für $r = 5$ erhalten wir $\mu(\sigma(3, 5)) = 8$, was der gesuchten Masse M entspricht.

Abbildung 2: Lineare Suche nach $M = 8$, *Quelle*: [2]

2.2 Binsearch

Der Algorithmus Binsearch führt zuerst ein Preprocessing auf dem Proteinstring aus, was später ein effizientes Auffinden aller in diesem Proteinstring vorkommenden Teilmassen ermöglicht.
Das Preprocessing besteht darin, dass alle Teilmassen eines Proteinstrings im Voraus berechnet und sortiert werden. Diese Daten werden in einem Array abgespeichert, welches mittels binärer Suche nach einer bestimmten Masse in Zeit O(log n) durchsucht werden kann. Der Speicherbedarf für die aufbereiteten Daten beträgt $O(n^2)$. *Quelle*: [2, Übersetzung]

Beispiel: Wir betrachten das fiktive Aminosäurenalphabet $A = \{a, b, c\}$. Die Massen der einzelnen Aminosäuren a, b und c sind 1, 2 und 5 und als Proteinstring definieren wir σ = abbcaba.

Preprocessing: Zuerst wird für jeden möglichen Teilstring die Masse berechnet und in einem Array gespeichert:

σ = a b b c a b a

Teilstring	Masse =
a	1
a b	3
a b b	5
a b b c	10
a b b c a	11
a b b c a b	13
a b b c a b a	14
b	2
b b	4
b b c	9
b b c a	10
b b c a b	12
b b c a b a	13
b	2
b c	7
b c a	8
b c a b	10
b c a b a	11
c	5
c a	6
c a b	8
c a b a	9
a	1
a b	3
a b a	4
b	2
b a	3
a	1

Array: [1, 3, 5, 10, 11, 13, 14, 2, 4, 9, 10, 12, 13, 2, 7, 8, 10, 11, 5, 6, 8, 9, 1, 3, 4, 2, 3, 1]

Anschliessend werden die im Array gespeicherten Massen sortiert:
Array: [1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 5, 5, 6, 7, 8, 8, 9, 9, 10, 10, 10, 11, 11, 12, 13, 13, 14]

Eine Optimierung kann durch Entfernung der Duplikate aus diesem Array erreicht werden:
Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 13, 14]

Algorithmus: Das Array mit den abgespeicherten Massen kann in logarithmischer Zeit binär durchsucht werden. Wird die gesuchte Masse im Array gefunden, gibt der Algorithmus **yes** aus, andernfalls **no**.

2.3 Lookup

Lookup hat einige Gemeinsamkeiten mit Linsearch. Wie dort, werden auch bei Lookup zwei Pointer *l* und *r* durch den Proteinstring bewegt, allerdings geschieht dies nicht wie bei Linsearch in Einzelschritten sondern mit einer Schrittlänge von *c*(n). Wählen wir z. B. eine Schrittlänge von log n, können wir die Anzahl Schritte von O(n) auf O(n/log n) reduzieren, wobei jeder Schritt O(log log n) anstelle von konstanter Zeit benötigt. Dies ergibt die sublineare Laufzeit von O(n/log n * log log n) für Lookup. Die Idee des Algorithmus wird über ein Beispiel eingeführt. *Quelle:* [2, Übersetzung]

2.3.1 Einleitendes Beispiel

Wir betrachten das fiktive Aminosäurenalphabet $A = \{a, b, c\}$. Die Massen der einzelnen Aminosäuren a, b und c betragen 1, 2 und 5 und als Proteinstring definieren wir σ = abbcabccaabb. Die gesuchte Masse M sei 14 und für $c(n)$ wählen wir den konstanten Wert 3. σ wird in Blöcke der Grösse $c(n)$ aufgeteilt. Die Zeiger l und r werden nicht wie bei Linsearch in Einzelschritten über σ bewegt, sondern in Schritten der Länge $c(n)$.
Nehmen wir an, l zeige auf den Anfang des ersten Blocks während r auf das Ende des zweiten Blocks zeigt, wie in Abbildung 3 dargestellt. Wir sind nun an den Masse*änderungen* der aktuellen Submasse zwischen l und r interessiert, wenn wir die Zeiger um höchstens $c(n)$ nach rechts bewegen. Hätten wir eine Liste dieser Masseänderungen, könnten wir nach $M - \mu(\sigma(l, r))$ suchen. Zum Beispiel ist die aktuelle Submasse in Abbildung 3 $\mu(\sigma(l, r)) = 13$ und wir wollen wissen, ob wir durch Bewegung von l und r um höchstens 3 Positionen die Masse 14 - 13 = 1 erreichen können.

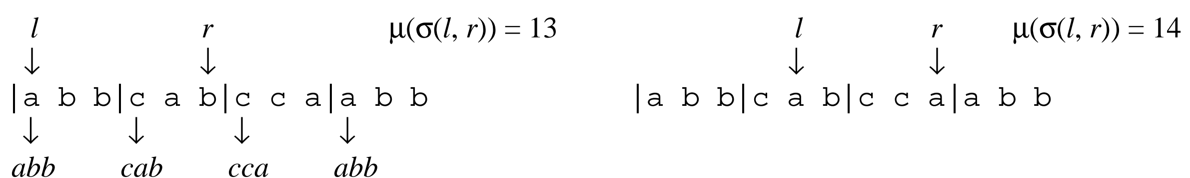


Abbildung 3: Lookup sucht nach M = 14, *Quelle:* [2]

Wir können diese Masseänderungen im Voraus berechnen und in einer $(c(n)+1) \times (c(n)+1)$ Matrix T speichern. Die (i, j)-Einträge von T speichern die Masseänderungen für Bewegungen von l und r um i - 1 und j - 1 Positionen nach rechts:

$$\mathrm{T}[abb, cca] = \begin{pmatrix} 0 & 5 & 10 & 11 \\ -1 & 4 & 9 & 10 \\ -3 & 2 & 7 & 8 \\ -5 & 0 & 5 & 6 \end{pmatrix}$$

Um die Suche nach einer bestimmten Masseänderung zu beschleunigen, speichern wir die Matrixeinträge zusätzlich in einem sortierten Array: S[*abb*,*caa*] = [-5, -3, -1, 0, 2, 4, 5, 6, 7, 8, 9, 10, 11]. Dies ermöglicht eine binäre Suche in logarithmischer Zeit der Arraygrösse. Im Fall unseres Beispieles ist die gesuchte 1 nicht im Array, somit müssen wir einen der beiden Zeiger zum nächsten Block bewegen.
Welchen Zeiger sollen wir bewegen? Wir wählen den Zeiger, der zuerst das Ende seines Blocks erreicht. Dazu vergleichen wir die Differenz $M - \mu(\sigma(l, r))$ mit dem Matrixeintrag $T(c(n), c(n))$ der einer Bewegung beider Zeiger um $c(n) - 1$ entspricht (Wert dieses Eintrags in T[*abb*, *caa*]: 7). Wenn die Differenz kleiner ist, bewegen wir den linken Zeiger zum nächsten Block, ansonsten den rechten. In unserem Beispiel ist die Differenz = 1, somit bewegen wir den linken Zeiger zum nächsten Block. Die aktuelle Submasse ändert sich um -5 (dem Minimum des Arrays) und wir erhalten für $\mu(\sigma(4, 6))$ = 13 - 5 = 8. Also suchen wir nach der neuen Differenz $M - \mu(\sigma(4, 6))$ = 14 - 8 = 6. Das sortierte Array für dieses Zeigerpaar ist S[*cab*, *cca*] = [-8, -6, -5, -3, -1, 0, 2, 3, 4, 5, 6, 10, 11] und als Matrix erhalten wir:

$$\mathrm{T}[cab, cca] = \begin{pmatrix} 0 & 5 & 10 & 11 \\ -5 & 0 & 5 & 6 \\ -6 & -1 & 4 & 5 \\ -8 & -3 & 2 & 3 \end{pmatrix}$$

Die gesuchte Differenz 6 ist im Array. In der Matrix wird diese Differenz erreicht, indem wir den linken Zeiger um eine und den rechten um drei Positionen weiterrücken. Der Algorithmus gibt die Positionen 5 und 9 für die Zeiger l und r aus und terminiert. *Quelle:* [2, Übersetzung]

2.3.2 Beschreibung von Lookup

Bis zur exakten Wahl von $c(n)$ in der Analyse nehmen wir $c(n) \approx \log n$ an. Einfachheitshalber nehmen wir an, dass $c(n)$ ein Teiler von n ist.

Preprocessing: Gegeben sei ein Proteinstring σ der Länge n. Zuerst berechnen wir $c(n)$. Dann erzeugen wir eine Tabelle T der Grösse $|A|^{c(n)} \times |A|^{c(n)}$. Jede Zeile und jede Spalte aus T wird durch einen String aus $A^{c(n)}$ indexiert. Für $I, J \in A^{c(n)}$ enthält der Tabelleneintrag $T[I, J]$ die Matrix und das sortierte Array wie oben beschrieben. Die Matrix enthält alle Differenzen $\mu(\text{präfix}(J)) - \mu(\text{präfix}(I))$, einschliesslich der leeren Präfixe. Die Tabelle T hängt somit nur von n und A ab, nicht aber vom String σ. Dann wird σ in Blöcke der Länge $c(n)$ aufgeteilt. Für jeden Block wird ein Zeiger zum Index I gespeichert, der für das Nachschlagen in der Tabelle T verwendet wird. Jeder dieser Indizes I repäsentiert einen String aus $A^{c(n)}$.

Algorithmus: Gegeben ist M, setze $l := 1$ und $r := 0$. Wiederhole die folgenden Schritte, bis entweder M gefunden wird oder $r > n$ gilt.

1. Nehmen wir an, l steht am Beginn des Blocks i und r steht am Ende des Blocks $j - 1$. Dann überprüfe, ob die Differenz $M - \mu(\sigma(l, r))$ im sortierten Array des Tabelleneintrags $T(I, J)$ gefunden werden kann.

2. Wenn $M - \mu(\sigma(l, r))$ im Array ist, durchsuche die Matrix des Tabelleneintrags $T(I, J)$ nach der Differenz $M - \mu(\sigma(l, r))$ und speichere die gefundenen Matrixindizes (k, l). Gib **yes** und den gefundenen Zeugen $i' := i * c(n) + k$, $j' := j * c(n) + (l - 1)$ aus, da $\mu(\sigma(i', j'))$ Masse M hat.

3. Wenn $M - \mu(\sigma(l, r))$ nicht im Array ist und $M - \mu(\sigma(l, r))$ kleiner als der Matrixeintrag an Position $(c(n), c(n))$ ist, inkrementiere l um $c(n)$ und setze $\mu(\sigma(l, r)) := \mu(\sigma(l, r)) + \min(\text{Array})$; ansonsten inkrementiere r um $c(n)$ und setze $\mu(\sigma(l, r)) := \mu(\sigma(l, r)) + \max(\text{Array})$.

Analyse: Zuerst leiten wir die Formeln für Speicherplatzbedarf und Laufzeit her, danach besprechen wir die Wahl von $c(n)$. Der Speicherplatzbedarf für die Tabelle T ist:

$$\begin{aligned} &\text{Anzahl Einträge} * (\text{Grösse einer Matrix} + \text{Grösse des sortierten Arrays}) \\ &= |A|^{2c(n)} * ((c(n) + 1)^2 + (c(n) + 1)^2) \\ &= O(|A|^{2c(n)} * c(n)^2). \end{aligned}$$

Der Platzbedarf, um für jeden Pointer einen Block zu speichern, beläuft sich auf:

$$\begin{aligned} &\text{Anzahl Blöcke} * \log(\text{Anzahl Elemente in } A^{c(n)}) \\ &= n / c(n) * \log(|A|^{c(n)}) \\ &= O(n). \end{aligned}$$

Für die Laufzeitberechnung beobachten wir, dass nach jeder Iteration (bestehend aus den Schritten 1 bis 3), entweder l oder r zum nächsten Block bewegt wird. Da jeder Zeiger höchstens $n / c(n)$ Mal vorrücken kann, erhalten wir höchstens $2 * n / c(n)$ Iterationen. Jede Iteration ausser die letzte benötigt Zeit $O(\log c(n)^2) + O(1)$. Die letzte Iteration beansprucht Zeit $O(c(n)^2)$.
Gesamthaft hat der Algorithmus einen Speicherbedarf von $O(n + |A|^{2c(n)} * c(n)^2)$ und eine Laufzeit von $O(n / c(n) * \log c(n) + n / c(n) + c(n)^2)$.

Wenden wir uns nun der Wahl von c(n) zu. Wenn wir $c(n) := \log_{|A|}n / 4$ wählen, erhalten wir für $|A|^{c(n)} = n^{¼}$. Dies ergibt einen Speicherbedarf von $O(n + n^{½} * \log^2 n) = O(n)$ und eine Laufzeit von O(n / log n * log log n), was einem linearen Speicherbedarf und einer sublinearen Laufzeit entspricht. Durch ein andere Wahl von c(n) werden nicht gleichzeitig Speicherbedarf und Laufzeit asymptotisch verbessert. *Quelle:* [2, Übersetzung]

2.3.3 Erläuterung der Nicht-Umsetzbarkeit für praktische Zwecke

Rufen wir uns an dieser Stelle in Erinnerung, dass das Alphabet *A* das Alphabet über den Aminosäuren ist. Von diesen Aminosäuren gibt es in der Praxis 20 häufig in Proteinen vorkommende, also $|A| = 20$. Es wird vorgeschlagen, $c(n) := \log_{|A|}n / 4$ zu wählen, womit wir $|A|^{c(n)} = n^{¼}$ erhalten, was schliesslich einen Speicherbedarf von $O(n + n^{½} * \log^2 n) = O(n)$ ergibt.
Die Wahl von $\log_{|A|}n / 4$ liefert jedoch selbst bei sehr grossem n immer noch praxisferne Blockgrössen, da wir es hier mit einem 20*er* Logarithmus zu tun haben. Proteinstrings überschreiten in der Praxis kaum die Länge von 3000 Aminosäuren.
In nachfolgender Tabelle sind die Blockgrössen c(n) für verschiedene n aufgeführt:

Blocklängen für $c(n) := \log_{\|A\|}n / 4$	
Proteinstringlänge n	Blocklänge c(n)
10^3	0.58
10^6	1.15
10^9	1.73

Tabelle 1: Proteinstringlängen und ihre Blocklängen

Wählen wir aber c(n) so, dass grössere Blocklängen erzeugt werden, erhalten wir einen Speicherbedarf jenseits der technischen Möglichkeiten.
Die Konkurrenzfähigkeit von Lookup gegenüber Linsearch und Binsearch ist deshalb kaum gewährleistet. Als Konsequenz davon beschlossen wir, Lookup im Rahmen dieser Semesterarbeit nicht zu implementieren.

2.4 Cluster

2.4.1 Beschreibung von Cluster

Hinter dem Cluster-Algorithmus steckt die folgende Idee: Gesucht ist eine Masse *M*. Stellen wir uns vor, dass wir durch ein Orakel eine Menge *C* von Positionen mit den folgenden Eigenschaften erhalten haben: Wenn *M* eine Submasse vom Proteinstring σ ist, dann gibt es ein $i \in C$, das *M abdeckt*, d.h. $i \in C$ ist Anfangsindex eines Proteinfragments aus σ mit Masse *M*. Somit müsste nur noch überprüft werden, ob eine der Positionen aus *C* für die Masse *M* tatsächlich in Frage kommt. Dies kann mit binärer Suche erledigt werden, sofern wir für jede Indexposition i = 1,...,n, die Suffixmasse $\sigma(i, n)$ abgespeichert haben. Nachfolgend wollen wir das Preprocessing und den Algorithmus genauer betrachten:

Preprocessing: Die Menge *C* erhalten wir duch das Preprocessing. Zuerst berechnen wir alle Submassen und speichern für jede Submasse x alle Positionen i, die x abdecken. Anschliessend werden die Submassen aufsteigend sortiert. Als nächstes werden die Submassen in nicht überschneidende Intervalle $I_1,...,I_m$, geclustert. Dabei speichern wir für jedes I_k eine Menge C_k von Positionen mit der Eigenschaft, dass für jedes $x \in I_k$ x eine Submasse von σ ist, falls es ein $i \in C_k$ gibt, das x abdeckt.
Wir müssen vor dem Preprocessing entscheiden, wieviele Positionen pro Intervall gespeichert werden.
Betrachten wir das Erstellen der Intervalle mit den Positionen etwas genauer:

Nennen wir das Intervall, das gerade erstellt wird, I_k. Wenn die nächste Submasse x in der Liste aller Submassen bereits durch ein $i \in C_k$ abgedeckt ist, dann nehmen wir x ins aktuelle Intervall I_k auf. Andernfalls wählen wir eine Position i, die x abdeckt. Wenn die maximale Anzahl Positionen pro Intervall noch nicht erreicht ist, nehmen wir i in C_k und x in I_k auf, andernfalls beginnen wir mit x ein neues Intervall I_{k+1} und fügen i in die zu I_{k+1} gehörige neue Positionsmenge C_{k+1} ein.

Algorithmus: Gesucht ist eine Masse M. Wir suchen nun zuerst binär nach einem Intervall k mit der Eigenschaft $M \in I_k$. Existiert kein solches Intervall, gibt der Algorithmus **no** aus, ansonsten überprüfen wir für jedes $i \in C_k$, ob i M abdeckt. Falls wir ein solches i finden, gibt Cluster **yes** und den bei i beginnenden Zeugen aus, andernfalls **no**.

Analyse: Nehmen wir an, wir haben m Intervalle und für jedes Intervall speichern wir c Indizes. Ferner benötigt das Speichern der Suffixmassen $O(n)$ Speicherplatz. Dies ergibt einen gesamten Speicherbedarf von $O(m * c + n)$. Für die Laufzeit der Intervallsuche erhalten wir $O(\log m)$, zusätzlich benötigen wir Zeit $O(c * \log n)$, um die gespeicherten Indizes des gefundenen Intervalls zu überprüfen. Somit erhalten wir die Laufzeit $O(\log m + c * \log n)$. *Quelle:* [2, Übersetzung]

2.4.2 Beispiel

Wir betrachten das fiktive Aminosäurenalphabet $A = \{a, b, c\}$. Die Massen der einzelnen Aminosäuren a, b und c sind 1, 2 und 5. Als Proteinstring definieren wir σ = ccaccb. Die gesuchte Masse M sei 6.

Preprocessing, Schritt 1: Wie bei Binsearch, werden zuerst für alle möglichen Teilstrings die Massen berechnet. Neben der Masse speichern wir auch noch die Startposition des Teilstrings im Proteinstring, also die Position, die die jeweilige Masse abdeckt:

σ = c c a c c b

Teilstring		Masse		Startposition
c	Masse =	5	Startposition =	1
c c		10		1
c c a		11		1
c c a c		16		1
c c a c c		21		1
c c a c c b		23		1
c		5		2
c a		6		2
c a c		11		2
c a c c		16		2
c a c c b		18		2
a		1		3
a c		6		3
a c c		11		3
a c c b		13		3
c		5		4
c c		10		4
c c b		12		4
c		5		5
c b		7		5
b		2		6

Schritt 2: Die Massen werden zusammen mit den Positionen, durch die sie abgedeckt werden, gespeichert. Anschliessend werden diese Daten bezüglich Massen aufsteigend sortiert:

Masse:	Positionen:
1	{3}
2	{6}
5	{1, 2, 4, 5}
6	{2, 3}
7	{5}
10	{1, 4}
11	{1, 2, 3}
12	{4}
13	{3}
16	{1, 2}
18	{2}
21	{1}
23	{1}

Schritt 3: Im letzten Schritt des Preprocessings werden die Massen in Intervalle geclustert. Das Beispiel zeigt die Clusterung für $c = 2$ und $c = 4$.

$c = 2$

Intervall:	Positionen:
[1, 2]	{3, 6}
[5, 6]	{1, 2}
[7, 11]	{5, 1}
[12, 13]	{4, 3}
[16, 23]	{1, 2}

$c = 4$

Intervall:	Positionen:
[1, 11]	{3, 6, 1, 5}
[12, 23]	{4, 3, 1, 2}

Algorithmus: Wir betrachten den Algorithmus anhand des Beispiels mit zwei gespeicherten Positionen pro Intervall: Zuerst werden die Intervalle binär nach der Masse $M = 6$ durchsucht. Der Algorithmus findet das Intervall [5, 6] mit den dazugehörigen Positionen {1, 2}. Nun wird überprüft, ob die Position 1 die Masse 6 abdeckt. Da dies nicht der Fall ist, geht der Algorithmus zur nächsten Position, in unserem Falle 2, weiter. 2 deckt die Masse 6 ab und der Algorithmus gibt **yes** und den an der Position 2 beginnenden Zeugen aus.

Kapitel 3

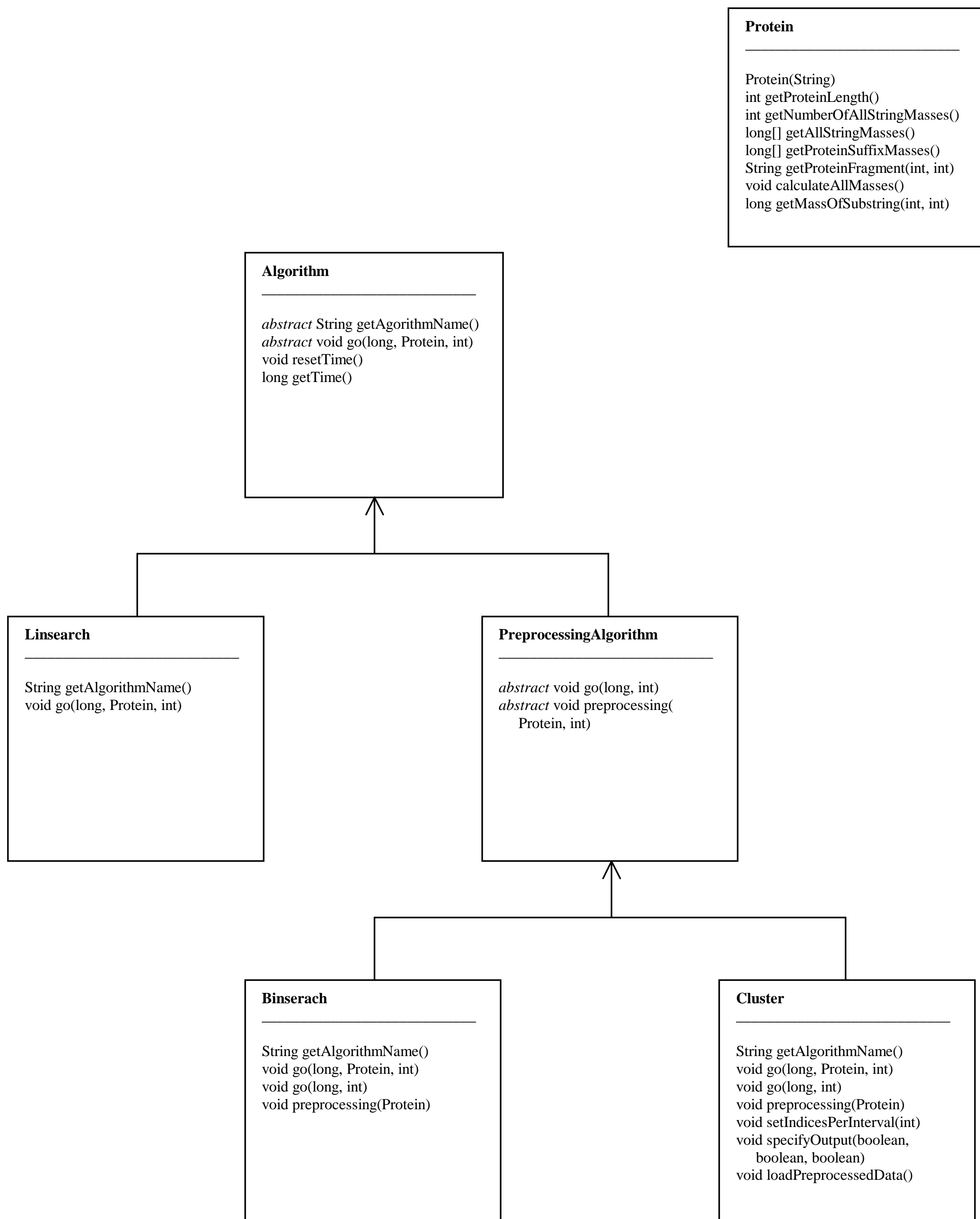
Implementierung

3.1 Design des Frameworks

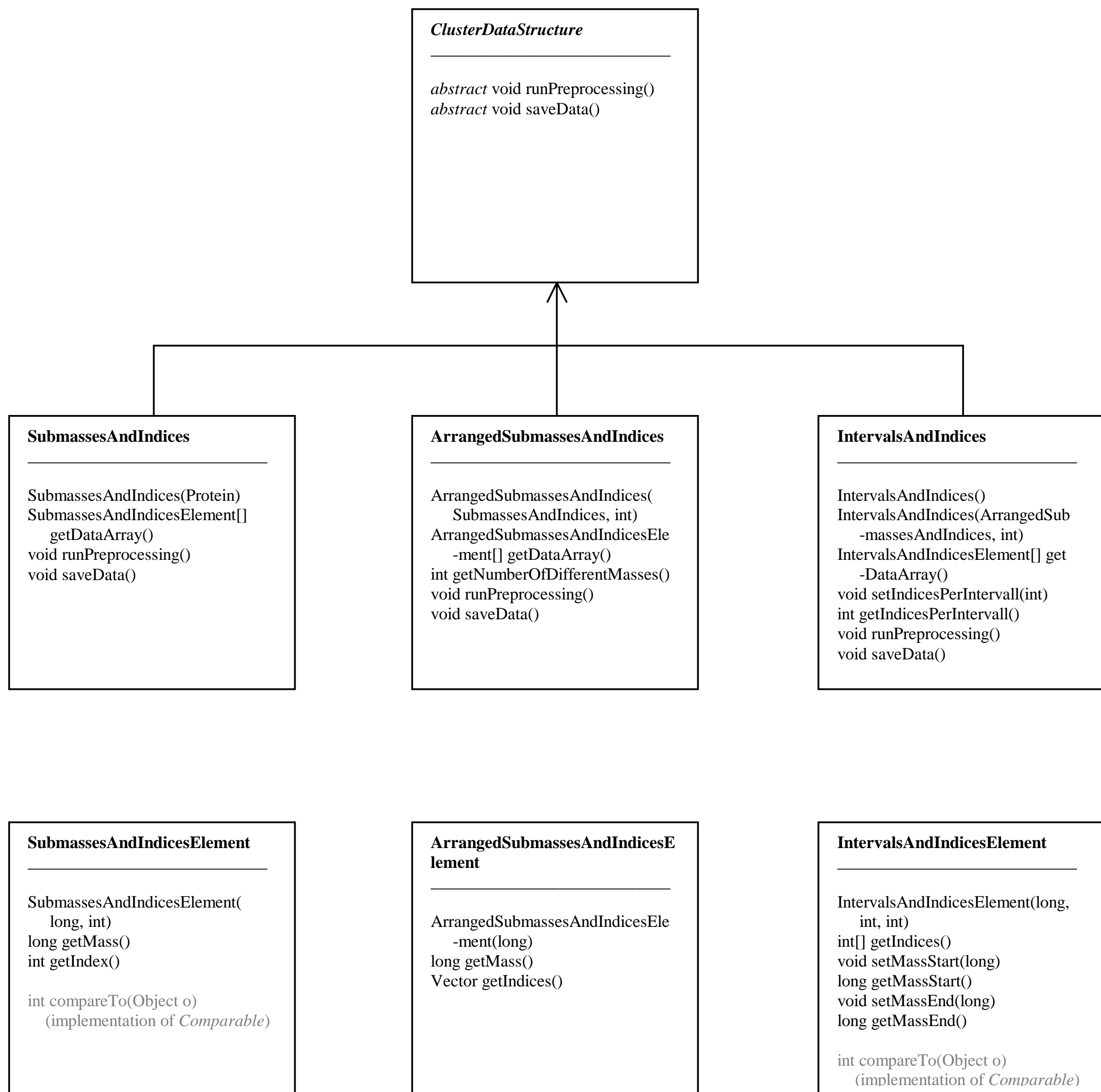
Das Design des Frameworks wurde so ausgelegt, dass es später um weitere Algorithmen erweitert werden kann. Hier soll ein Überblick über die Klassenhierarchie und ein Einblick in die *wichtigsten* Eigenschaften der einzelnen Klassen vermittelt werden.
Die Klasse *Algorithm* ist Superklasse für jeden Algorithmus. Sie implementiert Methoden für die Zeitmessung, die von den Algorithmen für das Messen der Laufzeit und des Preprocessings verwendet werden. Ausserdem muss jede Klasse, die *Algorithm* erweitert, die Methode *go* implementieren, welche einen Algorithmus mit einem Proteinstring und einer darin zu suchenden Masse startet.
Die abstrakte Klasse *PreprocessingAlgorithm* ist nur von jenen Algorithmen zu implementieren, die ein Preprocessing ausführen. Sie bietet via *go*-Methode die Möglichkeit, eine Masse ohne erneutes Preprocessing des zuletzt verwendeten Proteinstrings zu suchen.
Die Klassen *Linsearch*, *Binsearch* und *Cluster* enthalten die Algorithmen. *Cluster* bietet zudem folgende Methoden an: Mit *setIndicesPerInterval* kann die Anzahl Indizes pro Intervall festgelegt werden. Die Methode *specifyOutput* erlaubt es, die verschiedenen Stufen des Preprocessings in Outputfiles zu schreiben, was die Datenanalyse erleichtert. Mit der Methode *loadPreprocessedData* können nach einem Neustart des Programms zuvor gespeicherte Preprocessingdaten wieder geladen werden.
Eine Instanz der Klasse *Protein* wird für jeden neuen Proteinstring erzeugt. Diese Klasse, die Basisoperationen über dem Proteinstring anbietet, wird von jedem Algorithmus verwendet.
Die nachfolgenden Klassen dienen der Speicherung der Preprocessingdaten des Cluster Algorithmus: *ClusterDataStructure* dient als Interface für die Klassen *SubmassesAndIndices*, *ArrangedSubmasses-AndIndices* und *IntervalsAndIndices*, welche jeweils die Daten der Preprocessingschritte eins bis drei bearbeiten. Diese drei Klassen bieten Methoden an, um das Preprocessing der entsprechenden Stufe zu starten (*runPreprocessing*), die preprocessten Daten in ein File zu schreiben (*saveData*) und diese Daten in Form eines Arrays zu liefern (*getDataArray*). Die Arrays enthalten Instanzen der Klassen *SubmassesAndIndicesElement*, *ArrangedSubmassesAndIndicesElement* und *IntervalsAndIndices-Element*. Werden Daten in ein File geschrieben, erhält dieses den Namen der Preprocessing Klasse und den Suffix *.out*, z. B. heisst das File des ersten Preprocessingschrittes *SubmassesAndIndices.out*.
Weitere Informationen können dem Klassendiagramm in 3.2 oder dem Source Code in Anhang A entnommen werden.

3.2 Klassendiagramm

Algorithm Klassen und *Protein* Klasse:



Klassen und Interfaces, die für das Cluster Preprocessing benötigt werden:



Kapitel 4

Userinterface

4.1 Bedienung des Userinterfaces

Um die Algorithmen zu starten, benötigt man einen Computer, auf dem die Java Virtual Machine, Version 1.2.2 oder höher, installiert ist. Neben den verschiedenen Klassen des Programmframeworks muss ein File mit Testdaten bereitgestellt werden. Dieses File muss sich im selben Verzeichnis wie das Programm befinden und in einer einzigen Zeile den gesamten Proteinstring in Grossbuchstaben enthalten.
Mittels Userinterface können die Algorithmen gestartet werden. Das Userinterface wird in der Kommandozeile mit dem Befehl "`java Main`" gestartet. Folgende Punkte müssen beachtet werden:

- Es wird immer zuerst der Algorithmusname eingegeben, darauf folgen die Parameter.
- Als Algorithmusname kann alternativ auch nur dessen Anfangsbuchstabe eingegeben werden.
- Gewisse Parameter sind zwingend, andere fakultativ.
- Die Reihenfolge der Parametereingabe ist nicht relevant.
- Nach dem "`p=`" muss unmittelbar der Parameter folgen (ohne Spaces dazwischen).
- Das Programm wird durch die Eingabe von "`exit`" terminiert.
- Das Userinterface wurde für ein Windows-System programmiert. Es zeigte sich, dass auf gewissen anderen Systemen (z. B. UNIX) am Ende der Eingabezeile noch ein Space eingegeben werden muss, damit diese richtig übernommen wird.

Beschreibung der einzelnen Parameter:

Parameter	Beschreibung	Linsearch	Binsearch	Cluster
mass `m=`	Eingabe der zu suchenden Masse *Bsp*: `m=391.5347`	zwingend	zwingend	zwingend

protein
`p=`
	Hier wird das File angegeben, in dem das zu bearbeitende Protein gespeichert ist. Dieses File muss das Protein in einer einzigen Zeile speichern.
	Bsp:
	`protein=protein.txt`
	`protein=chloroplas`
	`protein=prot.txt+`
		zwingend	fakultativ; wird das Protein nicht angegeben, wird das zuletzt preprozesste Protein verwendet.	fakultativ; wird das Protein nicht angegeben, wird das zuletzt gepreprocesste Protein verwendet. Es besteht ausserdem die Möglichkeit, dem Filenamen ein "+" anzufügen, mit dieser Funktion wird ein gespeichertes, preprocesstes Protein von Disk geladen. Achtung: Das vor dem "+" angegebene Protein muss mit dem zuletzt preprocessten und gespeicherten übereinstimmen!

rounds `r=`	Hier kann die Anzahl Durchläufe (mit gleicher Masse und gleichem Protein) des Algorithmus spezifiziert werden. Dient der höheren Genauigkeit von Zeitmessungen. *Bsp*: `rounds=10000`	fakultativ, default ist 1	fakultativ, default ist 1	fakultativ, default ist 1
indices `i=`	Spezifiziert die Anzahl Indices pro Intervall *Bsp*: `indices=10`	-	-	fakultativ, default ist 5. Diese Option ist nur von Bedeutung, wenn ein *neues* Protein preprocessed wird.

output
`o=`
	Spezifiziert, welche Preprocessing-daten als File auf die Disk geschrieben werden. Das erste Flag bezieht sich auf SubmassesAndIndices, das zweite auf ArrangedSubmassesAndIndices und das dritte auf IntervalsAndIndices. Die Filenamen haben die Endung *.out*
	Bsp:
	`output=001`
	`output=011`
		-	-	fakultativ, default ist 000 d.h. kein Fileoutput wird erzeugt, die Daten werden lediglich im Memory gehalten.
				Diese Option ist nur von Bedeutung, wenn ein *neues* Protein preprocessed wird.

Beispiele:

```
c p=prot.txt m=505.682 i=10 o=111
Cluster p=prot.txt+ m=218.2334 r=10000
Binsearch protein=chlorop.txt m=336.3508
B m=514.6252
linsearch p=mitoch m=137.1412
```

4.2 Transkript einer Session

```
C:\>java Main

+-----------------------------------------------------------+
|          Algorithms For Protein Identification             |
+-----------------------------------------------------------+

l p=prot250.txt m=437.4126
Linsearch
Protein length: 250
mass found.  l=111, r=114  witness: HQDG, rounds: 1
time algorithm: 0 ms
------------------------------------------------------------
Binsearch m=300.2714 p=prot250.txt r=1000000
Binsearch
Protein length: 250
number of masses calculated: 31375
time preprocessing: 1210 ms
mass found, rounds: 1000000
Time Algorithm: 1480 ms
------------------------------------------------------------
C p=prot250.txt m=300.2714 o=001 i=10
Cluster
Protein length: 250
 level1, number of submasses calculated: 31375
 submasses and indices created, time: 3840 ms
  level2, number of different submasses: 28305
  submasses and indices arranged, time: 1320 ms
   level3, number of intervals: 2825, indices per interval: 10
   intervals and indices created, time: 2200 ms
total time preprocessing: 7360 ms
mass found.  l=112, r=114  witness: QDG, rounds: 1
time algorithm: 60 ms
------------------------------------------------------------
c m=437.4126
Cluster
mass found.  l=111, r=114  witness: HQDG, rounds: 1
time algorithm: 0 ms
------------------------------------------------------------
b m=437.4126
Binsearch
mass found, rounds: 1
Time Algorithm: 0 ms
------------------------------------------------------------
exit
------------------------------------------------------------
good bye!

C:\>
```

Kapitel 5

Testen der Algorithmen und Beurteilung von Cluster

5.1 Einleitung

Ziel dieser Semesterarbeit ist es unter anderem, konkrete Aussagen über die Praxistauglichkeit von Cluster zu machen. Hierbei müssen Linsearch, Binsearch und Cluster auf geeigneten realen Daten vergleichend laufen gelassen werden. Am Schluss steht die Interpretation der Testresultate.
Wir rufen uns in Erinnerung: Linsearch hat einen Speicherbedarf von O(1) und eine Laufzeit von O(n). Binsearch hat einen Speicherbedarf von $O(n^2)$ und eine Laufzeit von O(log n). Hat ein Algorithmus weniger Speicherbedarf als $O(n^2)$, nennen wir ihn *skinny*, braucht er weniger Zeit als O(n), ist er *speedy*. Linsearch und Binsearch sind jeweils nur skinny *oder* speedy. Wir suchen einen Algorithmus, der skinny *und* speedy ist. Die Frage ist, ob Cluster diese Anforderung erfüllt.

5.2 Auswahl der Testdaten und Beschreibung der Tests

Auf der Suche nach geeigneten Testdaten wurden diverse Proteindatenbanken durchsucht. Zwei Proteine wurden schliesslich für umfassende Tests ausgewählt. Die Bezeichnungen lauten: *AHT1_mitochondria.pep* und *ATH1_chloroplast.pep*. Die Proteine stammen aus [3]. Für einen Link auf diese Proteine sei auf Anhang B verwiesen. Proteine zählen in der Regel zwischen 100 und 3000 Aminosäuren. Für das Testen wurde aus den oben aufgeführten Proteinen je ein zufälliges Fragment mit 750 Aminosäuren extrahiert, da die Verarbeitung eines Fragments dieser Länge auf den zur Verfügung stehenden Computern bezüglich Speicherbedarf gerade noch möglich ist.
Um kein Zufallsresultat zu erhalten, wurden die Algorithmen mit verschiedenen Massen auf dem selben Protein mehrmals gestartet. Die Massen wurden folgendermassen ausgewählt: Aus jedem Proteinstring wurden zufällig je 5 Fragmente der Länge 5 und 10 ausgewählt und deren Massen berechnet. Weitere 10 Massen zwischen 0 und 10000 wurden mit einem Zufallsgenerator erzeugt. Somit suchte jeder Algorithmus in jedem Protein nach 20 verschiedenen Massen, zehnmal mit positivem und zehnmal mit negativem Resultat, da die zufälligen Massen nicht gefunden wurden. Cluster wurde ausserdem mit verschiedener Anzahl Indizes pro Intervall getestet: ½ log n, log n und 2 log n.
Um die Messgenauigkeit der Laufzeit zu erhöhen, wurden Linsearch und Cluster 10^5 Mal und Binsearch 10^6 Mal hintereinander ausgeführt. Die so erhaltene Zeit wurde anschliessend wieder durch die Anzahl Runden dividiert, was einen genauen Wert pro Durchgang ergab. Bei dieser Methode können ungewünschte Beschleunigungen durch Cacheoptimierung auftreten. Im vorliegenden Fall spielte dies jedoch keine entscheidende Rolle.
Die Testdaten und ein Log der Tests können im Anhang B eingesehen werden.

5.3 Interpretation des Testresultates

Die Auswertung der Testdaten legt den Schluss nahe, dass Cluster nicht den erwarteten Gewinn gegenüber den einfachen Algorithmen Linsearch und Binsearch bringt. Cluster scheint nicht *skinny* zu sein. Die zu speichernde Datenmenge bleibt gegenüber Binsearch praktisch unverändert. Die Hoffnung, dass mit steigender Anzahl Indizes pro Intervall deutlich weniger Daten gespeichert werden müssen, erfüllt sich nicht. Die Verdoppelung dieses Parameters macht sich bei Proteinen dieser Grössenordnung erst an der dritten Stelle nach dem Komma bemerkbar (vergl. Tabelleneintrag "*Relative Anzahl*"). Offenbar ist die Wahrscheinlichkeit, dass innerhalb eines Intervalls mehrere Massen einen gleichen Startindex haben, zu gering. Somit muss praktisch für jede auftretende Masse des Intervalls ein eigener Index gespeichert werden.

Weitere Resultate der Tests:

- Cluster läuft deutlich schneller als Linsearch. Das gemessene Geschwindigkeitsverhältnis stimmt mit dem der Laufzeitanalyse überein.
- Während es bei Binsearch an der Laufzeit nicht sichtbar ist, ob eine Masse gefunden wurde oder nicht, macht dies bei Cluster und Linsearch einen deutlichen Unterschied.
- Obwohl das Preprocessing von Binsearch und die erste Preprocessingstufe von Cluster ähnlich viele Daten aufbereiten, benötigt Cluster dafür deutlich mehr Zeit. Dies liegt vorwiegend daran, dass Binsearch nur primitive Datentypen speichert, während Cluster mit Objekten arbeitet.

5.4 Testresultate im tabellarischen Überblick

Die Algorithmuslaufzeiten sind jeweils das arithmetische Mittel aller 20 mit dieser Masse gemessenen Zeiten. Die Anzahl zu speichernder Daten ist im Falle von Binsearch die Anzahl aller möglichen Massen, also n * (n + 1) / 2. Bei Cluster bezieht sich dieser Wert auf die Anzahl gespeicherter Positionen, also ohne die Intervalldaten.

Protein *ATH1_chloroplast.pep:*

Laufzeiten der Algorithmen, Protein: *ATH1_chloroplast.pep*, 750 Aminosäuren extrahiert					
Algorithmus	Linsearch	Binsearch	Cluster		
			bei ½ log n Indizes pro Intervall	bei log n Indizes pro Intervall	bei 2 log n Indizes pro Intervall
Zeit, Masse gefunden (µs)	25.58	1.07	4.71	7.19	12.70
Zeit, Masse nicht gefunden (µs)	73.20	1.17	7.99	14.14	29.37
Zeit im Durch-schnitt (µs)	49.39	1.12	6.35	10.66	21.03

Anzahl zu speichernder Daten, Protein: *ATH1_chloroplast.pep*, 750 Aminosäuren extrahiert					
Algorithmus	Linsearch	Binsearch	Cluster		
			bei ½ log n Indizes pro Intervall	bei log n Indizes pro Intervall	bei 2 log n Indizes pro Intervall
Absolute Anzahl	0	281625 (ohne Duplikate: 248767)	248755	248720	248580
Relative Anzahl	0	1	0.883	0.883	0.883

Zeit Preprocessing, Protein: *ATH1_chloroplast.pep*, 750 Aminosäuren extrahiert					
Algorithmus	Linsearch	Binsearch	Cluster		
			bei ½ log n Indizes pro Intervall	bei log n Indizes pro Intervall	bei 2 log n Indizes pro Intervall
Zeit in Sekunden	-	0.68	5.30	4.56	4.74

Protein *ATH1_mitochondria.pep:*

Laufzeiten der Algorithmen, Protein: *ATH1_mitochondria.pep*, 750 Aminosäuren extrahiert					
Algorithmus	Linsearch	Binsearch	Cluster		
			bei ½ log n Indizes pro Intervall	bei log n Indizes pro Intervall	bei 2 log n Indizes pro Intervall
Zeit, Masse gefunden (µs)	34.83	1.07	5.26	8.31	16.30
Zeit, Masse nicht gefunden (µs)	73.25	1.16	7.10	14.07	29.45
Zeit im Durch-schnitt (µs)	54.04	1.12	6.18	11.19	22.88

Anzahl zu speichernder Daten, Protein: *ATH1_mitochondria.pep*, 750 Aminosäuren extrahiert					
Algorithmus	Linsearch	Binsearch	Cluster		
			bei ½ log n Indizes pro Intervall	bei log n Indizes pro Intervall	bei 2 log n Indizes pro Intervall
Absolute Anzahl	0	281625 (ohne Duplikate: 256211)	256200	256150	256000
Relative Anzahl	0	1	0.910	0.910	0.909

Zeit Preprocessing, Protein: *ATH1_mitochondria.pep*, 750 Aminosäuren extrahiert					
Algorithmus	Linsearch	Binsearch	Cluster		
			bei ½ log n Indizes pro Intervall	bei log n Indizes pro Intervall	bei 2 log n Indizes pro Intervall
Zeit in Sekunden	-	0.28	5.07	4.48	4.73

Kapitel 6

Referenzen

[1] Aufgabenstellung DIPLOMARBEIT/SEMESTERARBEIT: **Algorithmen zur Protein-Identifikation**

Mark Cieliebak, Zsuzsanna Lipták

[2] **Algorithmic Complexity of Protein Identification: Combinatorics of Weighted Strings**
TECHNICAL REPORT no. 361, ETH Zurich, Dept. of Computer Science

Mark Cieliebak, Thomas Erlebach, Zsuzsanna Lipták, Jens Stoye, Emo Welzl

[3] **TIGR *Arabidopsis thaliana* Database**

http://www.tigr.org/tdb/e2k1/ath1/

Anhang A

Source Code

Protein	I
Algorithm	III
Linsearch	IV
PreprocessingAlgorithm	V
Binsearch	VI
Cluster	VIII
ClusterDataStructure	XI
SubmassesAndIndices	XII
SubmassesAndIndicesElement	XIII
ArrangedSubmassesAndIndices	XIV
ArrangedSubmassesAndIndicesElement	XVI
IntervalsAndIndices	XVII
IntervalsAndIndicesElement	XX
Main	XXII

```
import java.util.*;
import java.text.*;

/**
 * Every instance of this Class houses a different protein
 * (= sequence of aminoacids).
 * This class also contains the hard coded static array of
 * aminoacid masses.
 *
 * @author   Christian Schlup
 */
public class Protein {

  private static final long average_mass[] = { 7107880,        0, 10314480, 11508860, 12911550,
                                              14717660,  5705200, 13714120, 11315950,        0,
                                              12817420, 11315950, 13119860, 11410390,        0,
                                               9711670, 12813080, 15618760,  8707820, 10110510,
                                                     0,  9913260, 18621330,        0, 16317600};


  private String proteinString;
  private int    proteinLength;
  private long   proteinSuffixMasses[];
  private long   allStringMasses[];
  private int    numberOfAllStringMasses;


 /**
  * Class constructor. Calculates the number of possible
  * fragments of this protein.
  */
  public Protein(String p) {
    proteinString = p;
    proteinLength = proteinString.length();
    numberOfAllStringMasses = getProteinLength() * (getProteinLength() + 1) / 2;
    preprocessProteinString();
  }



 /**
  * Gets the length of this protein (number of aminoacids).
  *
  * @return  length of the protein
  */
  public int getProteinLength() {
    return proteinLength;
  }



 /**
  * Gets the number of all possible fragments of this protein.
  *
  * @return  number of fragments
  */
  public int getNumberOfAllStringMasses() {
    return numberOfAllStringMasses;
  }



  /**
   * Gets a specified (by index) Fragment of this protein including boundaries.
   * @param from  startindex
   * @param to    endindex
   * @return      proteinfragment
   */
  public String getProteinFragment(int from, int to) {
    return proteinString.substring(from, to + 1);
  }



 /**
  * Gets the array that hosts the masses of all possible fragments
  * of this protein.
  *
  * @return  array of masses
  */
  public long[] getAllStringMasses() {
    return allStringMasses;
```

```java
  }
 /**
  * Gets the array that hosts the masses of all proteinfragments
  * that starts at the arrayindex and ends at the last index
  * (suffix masses).
  *
  * @return  array of masses
  */
  public long[] getProteinSuffixMasses() {
    return proteinSuffixMasses;
  }



  /**
   * Calculates the masses of all possible proteinfragments and saves
   * them in this <code>allStringMasses[]</code> array
   */
  public void calculateAllMasses() {
    long massSuffixes[] = new long[proteinLength + 1];
    for (int i = 0; i < proteinLength; i++) {
      massSuffixes[i] = getMassOfSubstring(i, proteinLength - 1);
    }
    massSuffixes[proteinLength] = 0;

    allStringMasses = new long[numberOfAllStringMasses];
    int overAllCounter = 0;
    for (int i = 0; i < proteinLength; i++) {
      for (int j = (i + 1); j < (proteinLength + 1); j++) {
        allStringMasses[overAllCounter] = massSuffixes[i] - massSuffixes[j];
        overAllCounter++;
      }
    }
  }



  /**
   * Gets the mass of a proteinfragment.
   * @param from  startindex
   * @param to    endindex
   * @return      mass
   */
  public long getMassOfSubstring(int from, int to) {
    return proteinSuffixMasses[from] - proteinSuffixMasses[to + 1];
  }



  /**
   * Gets the aminoacid mass specified in this <code>average_mass[]</code> array.
   *
   * @param ch  symbol of the aminoacid
   * @return    mass
   */
  private long getAverageAminoMass(Character ch) {
    return average_mass[ch.hashCode() - 65];
  }



  /**
   * Does some preprocessing on this protein. In the first loop it saves the
   * mass of every aminoacid of this protein, in the second loop the masses
   * starting at every index and ending at the last index (suffix masses) are
   * stored in this array.
   */
  private void preprocessProteinString() {
    proteinSuffixMasses = new long[proteinLength + 1];
    for (int i = 0; i < proteinLength; i++) {
      proteinSuffixMasses[i] = getAverageAminoMass(new Character(proteinString.charAt(i)));
    }
    for (int i = (proteinLength - 2); i >= 0; i--) {
      proteinSuffixMasses[i] = proteinSuffixMasses[i] + proteinSuffixMasses[i + 1];
    }
    proteinSuffixMasses[proteinLength] = 0;
  }

}
```

```java
/**
 * Algorithm is the abstract superclass for all algorithms.
 *
 * @author   Christian Schlup
 */
public abstract class Algorithm {

  private long time;


  /**
   * Gets the name of the algorithm.
   *
   * @return  algorithm name
   */
  public abstract String getAlgorithmName();



  /**
   * Runs the algorithm with a new mass and a new protein.
   *
   * @param mass     mass that we are looking for in the protein
   * @param protein  protein
   * @param rounds   defines how many times the algorithm searches the same
   *                 mass on the same protein. alows to get an more accurate
   *                 time on how long the algorithm takes to get a result.
   */
  public abstract void go(long mass, Protein protein, int rounds);



  /**
   * Resets the time.
   */
  public void resetTime() {
    time = System.currentTimeMillis();
  }



  /**
   * Gets the milliseconds since the time was reset.
   *
   * @return  time in milliseconds
   */
  public long getTime() {
    return (System.currentTimeMillis() - time);
  }

}
```

```java
/**
 * Linsearch is a simple algorithm that finds the mass of
 * a sequence of aminoacids in a protein. During runtime
 * it calculates masses of all possible proteinfragments
 * until the searched mass is found. Time O(n), storage
 * space O(1).
 *
 * @author   Christian Schlup
 */
public class Linsearch extends Algorithm {


  /**
   * Gets the name of the algorithm.
   *
   * @return  algorithm name
   */
  public String getAlgorithmName() {
    return new String("Linsearch");
  }



  /**
   * Runs the algorithm with a new mass and a new protein and outputs the
   * result on the screen.
   *
   * @param mass     mass that we are looking for in the protein
   * @param Protein  protein
   * @param rounds   defines how many times the algorithm searches the same
   *                 mass on the same protein. alows to get an more accurate
   *                 time on how long the algorithm takes to get a result.
   */
  public void go(long mass, Protein protein, int rounds) {
    int  left  = 0;
    int  right = 0;
    long currentMass = 0;
    resetTime();

    for (int lap = 0; lap < rounds; lap++) {
      left = 0;
      right = 0;
      int proteinLength = protein.getProteinLength();
      currentMass = protein.getMassOfSubstring(left, right);
      while ((mass != currentMass) &&
            ((right != (proteinLength - 1)) || (mass <= currentMass)) &&
            ((right != (proteinLength - 1)) || (left != (proteinLength - 1)))) {

        if ((mass > currentMass) || (left == right)) {
          right++;
        }
        if (mass < currentMass) {
          left++;
        }
        currentMass = protein.getMassOfSubstring(left, right);
      }
    }

    if (mass == currentMass) {
      System.out.println("mass found.  " + "l=" + left + ", r=" + right +
                         "  witness: " + protein.getProteinFragment(left, right) + ", rounds: " + rounds);
    } else {
      System.out.println("mass not found, rounds: " + rounds);
    }
    System.out.println("time algorithm: " + getTime() + " ms");
  }

}
```

```java
/**
 * Superclass for all preprocessing algorithms. Allows them to run
 * the algorithm with different masses on the same protein without
 * having to preprocess it again.
 *
 * @author   Christian Schlup
 */
public abstract class PreprocessingAlgorithm extends Algorithm {


  /**
   * Starts the algorithm with a new mass.
   *
   * @param mass     mass that we are looking for in the protein
   * @param rounds   defines how many times the algorithm searches the same
   *                 mass on the same protein. alows to get an more accurate
   *                 time on how long the algorithm takes to get a result.
   */
  public abstract void go(long mass, int rounds);



  /**
   * Calculates the masses of all possible proteinfragments and
   * sorts those masses.
   *
   * @param Protein  protein
   */
  public abstract void preprocessing(Protein protein);

}
```

```
import java.util.*;

/**
 * Binsearch is a simple algorithm that finds the mass of
 * a sequence of aminoacids in a protein. During preprocessing
 * it stores the masses of all possible proteinfragments.
 * The algorithm then does binary search over those masses.
 * Time O(log n), storage space O(n^2).
 *
 * @author   Christian Schlup
 */

public class Binsearch extends PreprocessingAlgorithm {

  private Protein protein;
  private boolean proteinSet = false;


  /**
   * Gets the name of the algorithm.
   *
   * @return  algorithm name
   */
  public String getAlgorithmName() {
    return new String("Binsearch");
  }



  /**
   * Runs the algorithm with a new mass and a new protein.
   *
   * @param mass     mass that we are looking for in the protein
   * @param protein  protein
   * @param rounds   defines how many times the algorithm searches the same
   *                 mass on the same protein. alows to get an more accurate
   *                 time on how long the algorithm takes to get a result.
   */
  public void go(long mass, Protein p, int rounds) {
    preprocessing(p);
    runAlgorithm(mass, rounds);
  }



  /**
   * Runs the algorithm with a new mass.
   *
   * @param mass     mass that we are looking for in the protein
   * @param rounds   defines how many times the algorithm searches the same
   *                 mass on the same protein. alows to get an more accurate
   *                 time on how long the algorithm takes to get a result.
   */
  public void go(long mass, int rounds) {
    if (proteinSet) {
      runAlgorithm(mass, rounds);
    } else {
      System.err.println("Please enter a protein for Binsearch.");
    }
  }



  /**
   * Calculates the masses of all possible proteinfragments and
   * sorts those masses.
   *
   * @param Protein  protein
   */
  public void preprocessing(Protein p) {
    protein = p;
    proteinSet = true;

    resetTime();
    protein.calculateAllMasses();
    Arrays.sort(protein.getAllStringMasses());
    System.out.println("number of masses calculated: " + (protein.getAllStringMasses()).length);
    System.out.println("time preprocessing: " + getTime() + " ms");
  }
```

```
  /**
   * Runs the algorithm that searches the stored fragmentmasses for a mass.
   * The results will be displayed on the screen.
   *
   * @param mass     mass that we are looking for in the fragments
   * @param rounds   defines how many times the algorithm searches the same
   *                 mass on the same protein. alows to get an more accurate
   *                 time on how long the algorithm takes to get a result.
   */
  private void runAlgorithm(long mass, int rounds) {
    resetTime();
    for (int lap = 0; lap < rounds - 1; lap++) {  // artifical overhead
      Arrays.binarySearch(protein.getAllStringMasses(), mass);
    }
    if ((mass <= 0) || (Arrays.binarySearch(protein.getAllStringMasses(), mass) < 0)) {
      System.out.println("mass not found, rounds: " + rounds);
    } else {
      System.out.println("mass found, rounds: " + rounds);
    }
    System.out.println("Time Algorithm: " + getTime() + " ms");
  }

}
```

```java
import java.util.*;

/**
 * Cluster is a speedy and skinny algorithm that finds the
 * mass of a sequence of aminoacids in a protein. While
 * preprocessing it forms intervals of masses and stores
 * for each interval several indices whith the following
 * property: If a mass M lies within an interval C there is
 * a substring of the protein that starts in I (= an index
 * of interval C) and has mass M. So the algorithm has
 * to check whether one of the positions in C works for M
 *
 * @author   Christian Schlup
 */
public class Cluster extends PreprocessingAlgorithm {

  private Protein protein;
  private boolean proteinSet = false;

  private SubmassesAndIndices         submassesAndIndices;
  private ArrangedSubmassesAndIndices arrangedSubmassesAndIndices;
  private IntervalsAndIndices         intervalsAndIndices;

  private int indicesPerInterval = 0;

  private boolean printSubmassesAndIndices = false;
  private boolean printArrangedSubmassesAndIndices = false;
  private boolean printIntervalsAndIndices = false;


  /**
   * Gets the name of the algorithm.
   *
   * @return  algorithm name
   */
  public String getAlgorithmName() {
    return new String("Cluster");
  }



  /**
   * Runs the algorithm with a new mass and a new protein.
   *
   * @param mass     mass that we are looking for in the protein
   * @param Protein  protein
   * @param rounds   defines how many times the algorithm searches the same
   *                 mass on the same protein. alows to get an more accurate
   *                 time on how long the algorithm takes to get a result.
   */
  public void go(long mass, Protein p, int rounds) {
    preprocessing(p);
    runAlgorithm(mass, rounds);
  }



  /**
   * Runs the algorithm with a new mass.
   *
   * @param mass     mass that we are looking for in the protein
   * @param rounds   defines how many times the algorithm searches the same
   *                 mass on the same protein. alows to get an more accurate
   *                 time on how long the algorithm takes to get a result.
   */
  public void go(long mass, int rounds) {
    if (proteinSet) {
      runAlgorithm(mass, rounds);
    } else {
      System.err.println("Please enter a protein for Cluster.");
    }
  }



  /**
   * Sets the number of indices for every interval for the preprocessing.
   *
   * @param i  number of indices per interval
   */
  public void setIndicesPerInterval(int i) {
    indicesPerInterval = i;
```

```
  }
  /**
   * Specifies which preprcessed data will be saved to disk.
   *
   * @param s  specifies if SubmassesAndIndices is saved to disk
   * @param a  specifies if ArrangedSubmassesAndIndices is saved to disk
   * @param i  specifies if IntervalsAndIndices is saved to disk
   */
  public void specifyOutput(boolean s, boolean a, boolean i) {
    printSubmassesAndIndices = s;
    printArrangedSubmassesAndIndices = a;
    printIntervalsAndIndices = i;

  }




  /**
   * Proceeds several preprocessing steps to finaly get the
   * massintervals with their indices.
   *
   * @param Protein  protein
   */
  public void preprocessing(Protein p) {
    protein = p;
    proteinSet = true;

    resetTime();
    buildSubmassesAndIndices();
    long timeSubmassesAndIndices = getTime();
    System.out.println(" submasses and indices created, time: " + timeSubmassesAndIndices + " ms");

    resetTime();
    buildArrangedSubmassesAndIndices();
    long timeArrangedSubmassesAndIndices = getTime();
    System.out.println("  submasses and indices arranged, time: " + timeArrangedSubmassesAndIndices + "
ms");

    resetTime();
    buildIntervalsAndIndices();
    long timeIntervalsAndIndices = getTime();
    System.out.println("   intervals and indices created, time: " + timeIntervalsAndIndices + " ms");

    System.out.println("total time preprocessing: " + (timeSubmassesAndIndices +
timeArrangedSubmassesAndIndices + timeIntervalsAndIndices) + " ms");
  }



  /**
   * Loads a file containing all the preprocessed Data.
   */
  public void loadPreprocessedData(Protein p) {
    protein = p;
    proteinSet = true;
    intervalsAndIndices = new IntervalsAndIndices();
  }


  /**
   * Runs the algorithm that searches the intervals for this mass. If an
   * interval is matched, it searches the indices of that interval for
   * this mass. It outputs the results on the screen.
   *
   * @param mass     mass that we are looking for in the fragments
   */
  private void runAlgorithm(long mass, int rounds) {
    resetTime();

    int found = 0;
    int intervalIndex;
    int indexCounter;
    int testedIndex = 0;
    int indicesPerInterval = intervalsAndIndices.getIndicesPerInterval();

    int right;
    int left;
    int middle = 0;

    for (int lap = 0; lap < rounds; lap++) {
      found = 0;
```

```
      intervalIndex = Arrays.binarySearch(intervalsAndIndices.getDataArray(), new Long(mass));
      if (intervalIndex >= 0) {  // interval found
        indexCounter = 0;
        while ((indexCounter < indicesPerInterval) && (found == 0)) {  // check every index
          testedIndex = ((intervalsAndIndices.getDataArray())[intervalIndex].getIndices())[indexCounter];
          left = testedIndex;
          right = protein.getProteinLength();

          while ((left <= right) && (found == 0)) {  // binary search for this mass
            middle = (left + right) / 2;
            if (mass == (protein.getProteinSuffixMasses())[testedIndex] -
(protein.getProteinSuffixMasses())[middle]) {
              found = 1;
            } else if (mass > (protein.getProteinSuffixMasses())[testedIndex] -
(protein.getProteinSuffixMasses())[middle]) {
              left = middle + 1;
            } else {
              right = middle - 1;
            }
          }
          indexCounter++;
        }
      }
    }
    if (found == 1) {
      System.out.println("mass found.  " + "l=" + testedIndex + ", r=" + --middle + "  witness: " +
protein.getProteinFragment(testedIndex, middle) + ", rounds: " + rounds);
    } else {
      System.out.println("mass not found, rounds: " + rounds);
    }
    System.out.println("time algorithm: " + getTime() + " ms");
  }



  /**
   * First step of Preprocessing the protein. Builds an array
   * from the protein that contains all masses of proteinfragments
   * and for every mass it's index.
   */
  private void buildSubmassesAndIndices() {
    submassesAndIndices = new SubmassesAndIndices(protein);
    submassesAndIndices.runPreprocessing();
    if (printSubmassesAndIndices) {
      submassesAndIndices.saveData();
    }
  }



  /**
   * Second step of Preprocessing the protein. Eliminates multiple
   * occurence of masses and combines every different mass with all
   * of it's indices.
   */
  private void buildArrangedSubmassesAndIndices() {
    arrangedSubmassesAndIndices = new ArrangedSubmassesAndIndices(submassesAndIndices,
protein.getNumberOfAllStringMasses());
    arrangedSubmassesAndIndices.runPreprocessing();
    if (printArrangedSubmassesAndIndices) {
      arrangedSubmassesAndIndices.saveData();
    }
  }



  /**
   * Third step of Preprocessing the protein. Builds from the Array
   * with the different masses and indices the intervals with their indices.
   */
  private void buildIntervalsAndIndices() {
    intervalsAndIndices = new IntervalsAndIndices(arrangedSubmassesAndIndices,
protein.getProteinLength());
    intervalsAndIndices.setIndicesPerInterval(indicesPerInterval);
    intervalsAndIndices.runPreprocessing();
    if (printIntervalsAndIndices) {
      intervalsAndIndices.saveData();
    }
  }

}
```

```
/**
 * Superclass of the datastructure used by the Cluster
 * algorithm.
 *
 * @author   Christian Schlup
 */
public interface ClusterDataStructure {


  /**
   * Runs the preprocessing for a new protein.
   */
  public abstract void runPreprocessing();



  /**
   * Saves the preprocessed data in a file.
   */
  public abstract void saveData();

}
```

```java
import java.util.*;
import java.io.*;

/**
 * This class hosts the first preprocessing level.
 * Builds an array from the protein that contains all
 * masses of proteinfragments and for every mass it's index.
 *
 * @author   Christian Schlup
 */
public class SubmassesAndIndices implements ClusterDataStructure {

  private Protein protein;
  private SubmassesAndIndicesElement arrayOfSubmassesAndIndices[];


  /**
   * Class constructor.
   *
   * @param  Protein
   */
  public SubmassesAndIndices(Protein p) {
    protein = p;
  }



  /**
   * Gets the array that contains the preprocessed data. This
   * array is used for further preprocessing.
   *
   * @return  array containing submasses and indices
   */
  public SubmassesAndIndicesElement[] getDataArray() {
    return arrayOfSubmassesAndIndices;
  }



  /**
   * Runs the preprocessing. Builds an array from the protein that
   * contains all masses of proteinfragments and for every mass it's
   * index.
   */
  public void runPreprocessing() {
    protein.calculateAllMasses();
    int proteinLength = protein.getProteinLength();
    arrayOfSubmassesAndIndices = new SubmassesAndIndicesElement[protein.getNumberOfAllStringMasses()];
    int  overallCounter = 0;
    for (int i = 0; i < proteinLength; i++) {
      for (int j = i; j < proteinLength; j++) {
        arrayOfSubmassesAndIndices[overallCounter] = new
SubmassesAndIndicesElement((protein.getAllStringMasses())[overallCounter], i);
        overallCounter++;
      }
    }
    Arrays.sort(arrayOfSubmassesAndIndices);
    System.out.println(" level1, number of submasses calculated: " +
protein.getNumberOfAllStringMasses());
  }



  /**
   * Saves the preprocessed data in a file. The filename is
   * <code>SubmassesAndIndices.out</code>
   */
  public void saveData() {
    try {
      PrintWriter out = new PrintWriter(new FileWriter("SubmassesAndIndices.out"));
      for (int i = 0; i < arrayOfSubmassesAndIndices.length; i++) {
        out.println(arrayOfSubmassesAndIndices[i].getMass() + ", " +
arrayOfSubmassesAndIndices[i].getIndex());
      }
      out.close();
    } catch(IOException e) {
      System.err.println(e.getMessage());
    }
  }

}
```

```java
/**
 * This class hosts an element of the first preprocessig level.
 * Each element contains a mass and it's index.
 *
 * @author   Christian Schlup
 */
class SubmassesAndIndicesElement implements Comparable {

  private long mass;
  private int  index;


  /**
   * Class constructor.
   *
   * @param  mass
   * @param  index
   */
  SubmassesAndIndicesElement(long m, int i) {
    mass  = m;
    index = i;
  }



  /**
   * Gets the mass of this element.
   *
   * @return  mass
   */
  public long getMass() {
    return mass;
  }



  /**
   * Gets the index of this element.
   *
   * @return  index
   */
  public int getIndex() {
    return index;
  }



  /**
   * This class must be comparable to another instance of this class.
   * Method for the <code>interface Comparable</code>
   *
   * @param   another instance of this class
   * @return  result of comparison: a negative integer, zero, or a positive
   *          integer as this object is less than, equal to, or greater than
   *          the specified object
   */
  public int compareTo(Object o) {
    SubmassesAndIndicesElement cs = (SubmassesAndIndicesElement)o;
    int result = (new Long(mass)).compareTo(new Long(cs.mass));
    if (result == 0) {
      result = (new Integer(index)).compareTo(new Integer(cs.index));
    }
    return result;
  }

}
```

```
import java.io.*;

/**
 * This class hosts the second preprocessing level.
 * Eliminates multiple occurence of masses and combines every
 * different mass with all of it's indices.
 *
 * @author   Christian Schlup
 */
public class ArrangedSubmassesAndIndices implements ClusterDataStructure {

  private SubmassesAndIndices submassesAndIndices;
  private ArrangedSubmassesAndIndicesElement arrayOfArrangedSubmassesAndIndices[];
  private int numberOfAllStringMasses;
  private int differentMassesCounter;


  /**
   * Class constructor.
   *
   * @param  submasses and indices = first level of preprocessing
   * @param  number of all possible fragments of the protein
   */
  public ArrangedSubmassesAndIndices(SubmassesAndIndices s, int p) {
    submassesAndIndices = s;
    numberOfAllStringMasses = p;
  }



  /**
   * Gets the number of different submasses of proteinfragments.
   *
   * @return  number of different submasses
   */
  public int getNumberOfDifferentMasses() {
    return differentMassesCounter;
  }



  /**
   * Gets the array that contains the preprocessed data. This
   * array is used for further preprocessing.
   *
   * @return  array containing submasses and indices
   */
  public ArrangedSubmassesAndIndicesElement[] getDataArray() {
    return arrayOfArrangedSubmassesAndIndices;
  }



  /**
   * Runs the preprocessing. Builds a new array from the last
   * preprocessed data that eliminates multiple occurence of masses
   * and combines every different mass with all of it's indices.
   */
  public void runPreprocessing() {
    arrayOfArrangedSubmassesAndIndices = new ArrangedSubmassesAndIndicesElement[numberOfAllStringMasses];
    int i = 0;
    long currentMass = 0;
    differentMassesCounter = 0;
    while(i < numberOfAllStringMasses) {
      currentMass = (submassesAndIndices.getDataArray())[i].getMass();
      arrayOfArrangedSubmassesAndIndices[differentMassesCounter] = new
ArrangedSubmassesAndIndicesElement(currentMass);

      while((i < numberOfAllStringMasses) && (currentMass ==
(submassesAndIndices.getDataArray())[i].getMass())) {
        arrayOfArrangedSubmassesAndIndices[differentMassesCounter].getIndices().add(new
Integer((submassesAndIndices.getDataArray())[i].getIndex()));
        i++;
      }
      differentMassesCounter++;
    }
    System.out.println("  level2, number of different submasses: " + differentMassesCounter);
  }
```

```
  /**
   * Saves the preprocessed data in a file. The filename is
   * <code>ArrangedSubmassesAndIndices.out</code>
   */
  public void saveData() {
    try {
      PrintWriter out = new PrintWriter(new FileWriter("ArrangedSubmassesAndIndices.out"));
      for (int i = 0; i < differentMassesCounter; i++) {
        out.println(arrayOfArrangedSubmassesAndIndices[i].getMass() + " " +
arrayOfArrangedSubmassesAndIndices[i].getIndices());
      }
      out.close();
    } catch(IOException e) {
      System.err.println(e.getMessage());
    }
  }

}
```

```java
import java.util.*;

/**
 * This class hosts an element of the second preprocessing level.
 * Each element contains a mass and it's indices.
 *
 * @author   Christian Schlup
 */
class ArrangedSubmassesAndIndicesElement {

  private long   mass;
  private Vector indices;


  /**
   * Class constructor.
   *
   * @param  mass
   */
  ArrangedSubmassesAndIndicesElement(long m) {
    mass    = m;
    indices = new Vector();
  }



  /**
   * Gets the mass of this element.
   *
   * @return  mass
   */
  public long getMass() {
    return mass;
  }



  /**
   * Gets a vector with all the indices for this mass.
   *
   * @return  vector of indices.
   */
  public Vector getIndices() {
    return indices;
  }

}
```

```java
import java.util.*;
import java.io.*;

/**
 * This class hosts the third (and last) preprocessing level.
 * Builds from the Array with the different masses and indices the
 * intervals and their indices.
 *
 * @author   Christian Schlup
 */
public class IntervalsAndIndices implements ClusterDataStructure {

  private int indicesPerInterval = 10;   // default

  private ArrangedSubmassesAndIndices arrangedSubmassesAndIndices;
  private int proteinLength;
  private IntervalsAndIndicesElement[] arrayOfIntervalsAndIndices;
  private Vector intervalsAndIndices;


  /**
   * Class constructor that loads a file containing the preprocessed Data.
   */
  public IntervalsAndIndices() {
    loadData();
  }



  /**
   * Class constructor.
   *
   * @param  arranged submasses and indices = second level of preprocessing
   * @param  length of the protein
   */
  public IntervalsAndIndices(ArrangedSubmassesAndIndices a, int p) {
    arrangedSubmassesAndIndices = a;
    proteinLength = p;
  }



  /**
   * Gets the array that contains the preprocessed data. This
   * array is used for the algorithm.
   *
   * @return  array containing intervals of masses and indices
   */
  public IntervalsAndIndicesElement[] getDataArray() {
    return arrayOfIntervalsAndIndices;
  }



  /**
   * Sets the number of indices per interval.
   *
   * @param number of indices
   */
  public void setIndicesPerInterval(int i) {
    indicesPerInterval = i;
  }



  /**
   * Gets the number of indices per interval.
   *
   * @return number of indices
   */
  public int getIndicesPerInterval() {
    return indicesPerInterval;
  }



  /**
   * Runs the preprocessing. Builds from the Array with the different masses
   * and indices the intervals with their indices.
   */
  public void runPreprocessing() {
    intervalsAndIndices = new Vector();
```

```
    int indicesPerIntervalCounter = 0;
    IntervalsAndIndicesElement currentElement = new
IntervalsAndIndicesElement((arrangedSubmassesAndIndices.getDataArray())[0].getMass(), indicesPerInterval,
proteinLength);
    for (int i = 0; i < arrangedSubmassesAndIndices.getNumberOfDifferentMasses(); i++) {
      long currentSubmass = (arrangedSubmassesAndIndices.getDataArray())[i].getMass();
      int  currentSubmassIndices[] = new
int[(arrangedSubmassesAndIndices.getDataArray())[i].getIndices().size()];

      for (int j = 0; j < (arrangedSubmassesAndIndices.getDataArray())[i].getIndices().size(); j++) {
        currentSubmassIndices[j] =
((Integer)(arrangedSubmassesAndIndices.getDataArray())[i].getIndices().get(j)).intValue();
      }

      int j = 0;
      while ((j < indicesPerInterval) && (Arrays.binarySearch(currentSubmassIndices,
(currentElement.getIndices())[j]) < 0)) {
        j++;
      }
      if (j != indicesPerInterval) {  //  the current submass is already covered by an index
        currentElement.setMassEnd(currentSubmass);
      } else {
        int choosenIndex = currentSubmassIndices[0];
        if (indicesPerIntervalCounter < indicesPerInterval) {  // current submass is not yet covered and
we can still add indices for this interval
          currentElement.setMassEnd(currentSubmass);
        } else {  // current submass is not yet covered and we can't add any further indices: create new
interval
          intervalsAndIndices.add(currentElement);
          currentElement = new IntervalsAndIndicesElement(currentSubmass, indicesPerInterval,
proteinLength);
          indicesPerIntervalCounter = 0;
        }
        (currentElement.getIndices())[indicesPerIntervalCounter] = choosenIndex;
        indicesPerIntervalCounter++;
      }
    }
    intervalsAndIndices.add(currentElement);  // save the last interval
    System.out.println("   level3, number of intervals: " + intervalsAndIndices.size() + ", indices per
interval: " + indicesPerInterval);
    vectorToArray();
  }



  /**
   * Copies the intervalsAndIndices Vector to an array making sure, the
   * Array contains IntervalsAndIndicesElements and not just Objects.
   */
  private void vectorToArray() {
    /* copy this vector to an array */
    arrayOfIntervalsAndIndices = new IntervalsAndIndicesElement[intervalsAndIndices.size()];
    for (int i = 0; i < intervalsAndIndices.size(); i++) {
      arrayOfIntervalsAndIndices[i] = (IntervalsAndIndicesElement)intervalsAndIndices.get(i);
    }
    intervalsAndIndices.clear();
  }



  /**
   * Saves the preprocessed data in a file. The filename is
   * <code>IntervalsAndIndices.out</code>
   */
  public void saveData() {
    try {
      PrintWriter out = new PrintWriter(new FileWriter("IntervalsAndIndices.out"));
      out.println(indicesPerInterval);
      for (int i = 0; i < arrayOfIntervalsAndIndices.length; i++) {
        out.println("[" + arrayOfIntervalsAndIndices[i].getMassStart() + ", " +
arrayOfIntervalsAndIndices[i].getMassEnd() + "]");
        for (int j = 0; j < indicesPerInterval; j++) {
          out.println((arrayOfIntervalsAndIndices[i].getIndices())[j]);
        }
      }
      out.close();
    } catch(IOException e) {
      System.err.println(e.getMessage());
    }
  }
```

```
  /**
   * Loads a former preprocessed datafile. The filename is
   * <code>IntervalsAndIndices.out</code>
   */
  private void loadData() {
    try {
      BufferedReader in = new BufferedReader(new FileReader("IntervalsAndIndices.out"));
      intervalsAndIndices = new Vector();

      int indexOfIndices = 0;
      int positionOfComma;
      String comma = new String(",");
      String bracket = new String("]");
      String currentRecord;
      IntervalsAndIndicesElement currentIntervalsAndIndicesElement = null;

      if (in.ready()) {  // read number of indices per interval from file
        currentRecord = new String(in.readLine());  // first record: how many indices per interval
        indicesPerInterval = Integer.parseInt(currentRecord);

        if (in.ready()) {  // read intervals and indices from file
          currentRecord = new String(in.readLine());  // interval record
          while (in.ready()) {
            positionOfComma = currentRecord.indexOf(comma);

            if (positionOfComma > 0) {  // interval record
              currentIntervalsAndIndicesElement = new IntervalsAndIndicesElement(0, indicesPerInterval,
0);
              intervalsAndIndices.add(currentIntervalsAndIndicesElement);
              currentIntervalsAndIndicesElement.setMassStart(Long.parseLong(currentRecord.substring(1,
positionOfComma)));

currentIntervalsAndIndicesElement.setMassEnd(Long.parseLong(currentRecord.substring(positionOfComma + 2,
currentRecord.indexOf(bracket))));
              indexOfIndices = 0;
              currentRecord = new String(in.readLine());
            } else {  // index records
              while ((currentRecord.indexOf(comma) == -1) && (in.ready())) {
                (currentIntervalsAndIndicesElement.getIndices())[indexOfIndices] =
Integer.parseInt(currentRecord);
                indexOfIndices++;
                currentRecord = new String(in.readLine());
              }
            }
          }
          (currentIntervalsAndIndicesElement.getIndices())[indexOfIndices] =
Integer.parseInt(currentRecord);
        }
        vectorToArray();
      }
      in.close();
    } catch(IOException e) {
      System.err.println(e.getMessage());
    }
  }

}
```

```java
import java.util.*;

/**
 * This class hosts an element the third (and last) preprocessing
 * level. Each element contains an interval of masses and their
 * indices.
 *
 * @author   Christian Schlup
 */
class IntervalsAndIndicesElement implements Comparable {

  private long   massStart;
  private long   massEnd;
  private int    indicesPerInterval;
  private int    indices[];


  /**
   * Class constructor.
   *
   * @param  mass
   * @param  number of allowed indices per interval
   * @param  Length of the protein. If the number of indices of the last
   *         interval is smaller than the allowed number of indices
   *         per interval, we fill them with the proteinlength. This leeds
   *         to an efficient binary search for the last interval if the
   *         mass was not found.
   *
   */
  IntervalsAndIndicesElement(long mass, int i, int proteinLength) {
    massStart          = mass;
    massEnd            = mass;
    indicesPerInterval = i;
    indices            = new int[indicesPerInterval];
    Arrays.fill(this.indices, proteinLength);  // I know that's bad. Any better ideas?
  }



  /**
   * Gets the array of indices for this interval.
   *
   * @return  array of indices
   */
  public int[] getIndices() {
    return indices;
  }



  /**
   * Sets the start mass of this interval.
   *
   * @param  start mass
   */
  public void setMassStart(long s) {
    this.massStart = s;
  }



  /**
   * Gets the start mass of this interval.
   *
   * @return  start mass
   */
  public long getMassStart() {
    return massStart;
  }



  /**
   * Sets the end mass of this interval.
   *
   * @param  end mass
   */
  public void setMassEnd(long e) {
    this.massEnd = e;
  }
```

```
  /**
   * Gets the end mass of this interval.
   *
   * @return  end mass
   */
  public long getMassEnd() {
    return massEnd;
  }



  /**
   * This class must be comparable to another instance of this class.
   * Method for the <code>interface Comparable</code>
   *
   * @param   another instance of this class
   * @return  result of comparison: a negative integer, zero, or a positive
   *          integer as this object is less than, equal to, or greater than
   *          the specified object
   */
  public int compareTo(Object o) {
    long m = ((Long)o).longValue();
    if (m > massEnd) {
      return -1;
    } else if (m < massStart) {
      return  1;
    } else {
      return  0;
    }
  }

}
```

```java
import java.io.*;

/**
 * This class is the userinterface for the algorithm framework.
 * It allows to run algorithms and to specifiy parameters.
 * First, specify an algorithm name. Choose between Linsearch,
 * Binsearch and Cluster. Then specify the parameter.
 * The following parameters are allowed:
 * - Mass, specified with m=
 * - Protein, specified with p=
 * - Number of rounds, specified with r=, default = 1
 * And only for the Cluster algorithm:
 * - Indices, specified with i=, default = 5. Sets the number of
 * indices per interval.
 * - Output, specified with o=, default = 000. Determines what
 * files to create during the preprocessing.
 *
 * @author   Christian Schlup
 */
public class Main {

  private Linsearch linsearch;
  private Binsearch binsearch;
  private Cluster   cluster;

  private Protein protein;

  private String  proteinFile;
  private boolean proteinSpecified;
  private long    mass;
  private boolean printSubmassesAndIndices;
  private boolean printArrangedSubmassesAndIndices;
  private boolean printIntervalsAndIndices;

  /**
   * Parameter names.
   */
  private String proteinString = new String("p=");
  private String massString    = new String("m=");
  private String roundsString  = new String("r=");
  private String indicesString = new String("i=");
  private String outputString  = new String("o=");

  private String spaceChar     = new String(" ");


  /**
   * The static main method creates an instance of this class.
   */
  public static void main(String argv[]) {
    Main main = new Main();
  }



  /**
   * The Class Constructor creates an instance of every algorithm and
   * starts the input loop, waiting for user input.
   */
  public Main() {
    linsearch = new Linsearch();
    binsearch = new Binsearch();
    cluster     = new Cluster();
    inputLoop();
  }



  /**
   * Runs by user input specified algorithms with their parameters.
   */
  private void inputLoop() {
    String inLine = new String("");

    System.out.println("\n+---------------------------------------------------------+");
    System.out.println("|           Algorithms For Protein Identification           |");
    System.out.println("+---------------------------------------------------------+\n");

    while (!inLine.startsWith("exi")) {
      proteinSpecified = false;
      inLine = new String(readString());
```

```
      inLine = inLine.substring(0, inLine.length() - 1).concat(spaceChar);  // adding a space makes it
easy to read a parameter at end of line

      /* Linsearch */
      if ((inLine.startsWith("Linsearch")) || (inLine.startsWith("linsearch")) || (inLine.startsWith("L"))
|| (inLine.startsWith("l"))) {
        System.out.println(linsearch.getAlgorithmName());
        if (proteinParameter(inLine)) {  // protein specified
          if (massParameter(inLine)) {  // mass specified
            protein = new Protein(readProtein());
            System.out.println("Protein length: " + protein.getProteinLength());
            linsearch.go(mass, protein, roundsParameter(inLine));
          } else {
            System.out.println("Error: no mass specified");
          }
        } else {
          System.out.println("Error: no protein specified");
        }

      /* Binsearch */
      } else if ((inLine.startsWith("Binsearch")) || (inLine.startsWith("binsearch")) ||
(inLine.startsWith("B")) || (inLine.startsWith("b"))) {
        System.out.println(binsearch.getAlgorithmName());
        if (massParameter(inLine)) {  // mass specified
          if (proteinParameter(inLine)) {  // protein specified: with preprocessing
            protein = new Protein(readProtein());
            System.out.println("Protein length: " + protein.getProteinLength());
            binsearch.go(mass, protein, roundsParameter(inLine));
          } else {  // no protein specified: use last protein, no preprocessing
            binsearch.go(mass, roundsParameter(inLine));
          }
        } else {
          System.out.println("Error: no mass specified");
        }

      /* Cluster */
      } else if ((inLine.startsWith("Cluster")) || (inLine.startsWith("cluster")) ||
(inLine.startsWith("C")) || (inLine.startsWith("c"))) {
        System.out.println(cluster.getAlgorithmName());
        if (massParameter(inLine)) {  // mass specified
          outputParameter(inLine);
          cluster.specifyOutput(printSubmassesAndIndices, printArrangedSubmassesAndIndices,
printIntervalsAndIndices);
          cluster.setIndicesPerInterval(indicesParameter(inLine));
          if (proteinParameter(inLine)) {  // protein specified
            if (proteinFile.endsWith("+")) {  // read preprocessed data
              proteinFile = proteinFile.substring(0, proteinFile.length() - 1);
              protein = new Protein(readProtein());
              System.out.println("Protein length: " + protein.getProteinLength());
              cluster.loadPreprocessedData(protein);
              cluster.go(mass, roundsParameter(inLine));
            } else {  // preprocess protein
              protein = new Protein(readProtein());
              System.out.println("Protein length: " + protein.getProteinLength());
              cluster.go(mass, protein, roundsParameter(inLine));
            }
          } else {  // no protein specified: use last protein, no preprocessing
            cluster.go(mass, roundsParameter(inLine));
          }
        } else {
          System.out.println("Error: no mass specified");
        }

      } else {
        if (!inLine.startsWith("exi")) {
          System.out.println("Error: no algorithm specified");
        }
      }
      System.out.println("------------------------------------------------------------");
    }
    System.out.println("good bye!");
  }



  /**
   * Reads a protein from harddisk and returns the protein as a string.
   *
   * @return  protein as a string
   */
  private String readProtein() {
    String proteinInputString = "";
```

```
    try {
      BufferedReader in = new BufferedReader(new FileReader(proteinFile));
      if (in.ready()) {
        proteinInputString = new String(in.readLine());
      }
      in.close();
    } catch(IOException e) {
      System.err.println(e.getMessage());
    }
    return proteinInputString;
  }



  /**
   * Extracts the protein parameter and gets the filename of the proteinfile.
   *
   * @param   inLine  user input
   * @return  true if the proteinparameter is set, false otherwise
   */
  private boolean proteinParameter(String inLine) {
    int startProteinString = inLine.indexOf(proteinString);
    if (startProteinString != -1) {
      int endProteinString = inLine.indexOf(spaceChar, startProteinString + proteinString.length());
      proteinFile = inLine.substring(startProteinString + proteinString.length(), endProteinString);
      proteinSpecified = true;
    } else {
      proteinSpecified = false;
    }
    return proteinSpecified;
  }



  /**
   * Extracts the mass parameter and sets the mass.
   *
   * @param   inLine  user input
   * @return  true if the mass parameter is set, false otherwise
   */
  private boolean massParameter(String inLine) {
    int startMassString = inLine.indexOf(massString);
    if (startMassString != -1) {
      int endMassString = inLine.indexOf(spaceChar, startMassString + massString.length());
      mass = (long)(100000 * Double.parseDouble(inLine.substring(startMassString + massString.length(),
endMassString)));
      return true;
    }
    return false;
  }



  /**
   * Extracts the rounds parameter and gets the number of rounds.
   * @param   inLine  user input
   * @return  number of rounds
   */
  private int roundsParameter(String inLine) {
    int startRoundsString = inLine.indexOf(roundsString);
    if (startRoundsString != -1) {
      int endRoundsString = inLine.indexOf(spaceChar, startRoundsString + roundsString.length());
      return Integer.parseInt(inLine.substring(startRoundsString + roundsString.length(),
endRoundsString));
    } else {
      return 1;  // default setting if rounds are not specified
    }
  }



  /**
   * Extracts the indices parameter and gets the number of indices.
   * Works only for the Cluster algorithm.
   *
   * @param   inLine  user input
   * @return  number of rounds
   */
  private int indicesParameter(String inLine) {
    int startIndicesString = inLine.indexOf(indicesString);
    if (startIndicesString != -1) {
      int endIndicesString = inLine.indexOf(spaceChar, startIndicesString + indicesString.length());
```

```
      return Integer.parseInt(inLine.substring(startIndicesString + indicesString.length(),
endIndicesString));
    } else {
      return 5;  // default setting if indices are not specified
    }
  }



  /**
   * Extracts the output parameter from the String inLine. The parameter has semantics as follows:
   * For ervery preprocessed data, an outputfile can be created. This can be specified by setting
   * a 0 or a 1. There are tree different proprocessed data levels, if an outputfile sould be created
   * for every level, the parameter is: output=111, if no output should be created, the parameter
   * can be specified as: ouput=000. If no output parameter is set, no output will be created.
   * This parameter works only for the Cluster algorithm.
   *
   * @param   inLine  user input
   */
  private void outputParameter(String inLine) {
    printSubmassesAndIndices = false;
    printArrangedSubmassesAndIndices = false;
    printIntervalsAndIndices = false;
    int startOutputString = inLine.indexOf(outputString);
    if (startOutputString != -1) {
      int endOutputString = inLine.indexOf(spaceChar, startOutputString + outputString.length());
      if (Integer.parseInt(inLine.substring(startOutputString + outputString.length(), startOutputString +
outputString.length() + 1)) == 1) {
        printSubmassesAndIndices = true;
      }
      if (Integer.parseInt(inLine.substring(startOutputString + outputString.length() + 1,
startOutputString + outputString.length() + 2)) == 1) {
        printArrangedSubmassesAndIndices = true;
      }
      if (Integer.parseInt(inLine.substring(startOutputString + outputString.length() + 2,
startOutputString + outputString.length() + 3)) == 1) {
        printIntervalsAndIndices = true;
      }
    }
  }



  /**
   * Reads a string from the console.
   *
   * @return  input string
   */
  private String readString() {
    int ch;
    String inLine = "";
    boolean done = false;
    while (!done) {
      try {
        ch = System.in.read();
        if (ch < 0 || (char)ch == '\n') {
          done = true;
        } else {
          inLine = inLine + (char) ch;
        }
      } catch(java.io.IOException e) {
        done = true;
      }
    }
    return inLine;
  }


}
```

Anhang B

Testdaten, Testlog

In diesem Anhang finden sich sowohl die Links auf die beiden für das Testen ausgesuchten Proteine als auch die Proteinstrings der Länge 750, mit denen die Algorithmen getestet wurden. Fett hervorgehoben sind die zufällig ausgewählten Fragmente, deren Masse im Voraus für die Tests berechnet wurde (siehe 5.2).

Aus: ftp://ftp.tigr.org/put/data/a_thaliana/ath1/SEQUENCES/ATH1_chloroplast.pep

chlorop750.txt

VYLVFTTNDFPWLTI**IVVFP**ISAGSLMLFLPHRGNKVNK**WYTICICILE**LLLTTYAFCYNFKMDDPLIQLSEDYK
WIDFFDFYWRMGIDGLSIGTILLTGFITTLATLAAFPVTRDSRFFHFLMLAMYSGQIGSFSSRDL**LLFFIMWELE**
LIPVYLLLSMWGGKKRLYSATKFILYTAGSSIFLL**IGVLG**ISLYGSNEPTLNLELLANKSYPVTLEILFYIGFLI
AFAVKSPIIPLHTWLPDTHGEAHYSTCMLLAGILLKMGAYGLVRINMELLPHAHSMFSPWL**LVVGT**IQIIYAAST
SPGQRNLKKRIAYSSVSHMGFIIIGISSITDPGLNGAILQIISHGFIGAALFFLAGTSYDRIRLVYLDEMGGMAI
SIPKIFTMFTILSMASLALPGMSGFIAEFIVFFGIITSQKYFLISKIFIIFVMAIGMILTPIYLLSMLRQMFYGY
KLINIKNFSFFDSGPRELFLSISILLPIIGI**GIYPD**FVLSLASDKVESILSNYFYGVMV**FQSFILGNLV**SLCMKI
INSVVVVGLYYGFLTTFSIGPSYLFLLRARVMDEGEEGTEKKVSATTGFIAGQLMMFISIYYAPLHLALGRPHTI
TVLALPYLLFHFFWNNHKHFFDYGSTTRNEMRNLRIQCVFLNNL**IFQLF**NHFILPSSMLARLVNIYMFRCNNKML
FVTSSFVGWLIGHILFMKWVG**LVLVWIQQNN**SIRSNVVIRSNKYKLDKEWSVENILEKTTRFCYNEAKKEYLPKI

Aus: ftp://ftp.tigr.org/put/data/a_thaliana/ath1/SEQUENCES/ATH1_mitochondria.pep

mitoch750.txt

MYLLIVFLSMLSSSVAGFFGRFLGS**ESVSR**FNLIIFLILLV**FSICLFRSLK**QYLGKRMTQWCYLALVCQISLFLV
LLRSHILAGFGTFSADVFTVFMGTFSVTGSSGGIVNHQDGASSEWFTYTSDMVEDSASSGRTSSSVNQPIPEEQA
WEREARAQEHDRISAEVETITSACENLEAAMVRKAQILLHQRGVTLGDPEDV**KRALQ**LALHDDWEHAIDDRKRHF
TVLRRNF**GTARCERWNP**FIDELRGLGNH**QVNAR**HYVDMSVLRRIPQDGTFHQEGPIHRLAKRRPR**FIASFDLSAA**
TDRWPVPVIYELMACLFGQTMASCIVNGALALNSCSLKSVTGRHDEVVFVAGQPLGYYGSWALFALSHHAIVWLA
ALRAYPHQTRPFLDYALLGDDIVIADRSVAKEYRSL**LDALQ**VDISDAKSIVSETGCLEFAKRFWVKIMSKDLSPV
SAKAVLESYFLVGTQQLAYKYKLSPKTCLRLNKAGYRVLGQMDTTLRPYPGVLVGFRRYLVSFMERLTRLNHFLV
NMRWDFYEGVIQAGY**IRNLQRELDH**TPAELLGSKLDLIFFRESLNLSTYVNNWYMQNLGVPGPVNFIEKYHDACF
SNYMKLMEIPSPLDQFEIVPLIPMHIGNFYFSFTNPSLFMLLTLSFFLLLIHFVTKKGGGNLVPNAWQSLVELLY
D**FVLNL**VKEQIGGLSGNVKQMFFPCILVTFLFLLFCNLQGMIPYSVPAGKAMLGRVVDAMGVP**IDGKGALSDH**EQ

TEST: chlorop750.txt

```
************************************************************Linsearch
mass found.  l=15, r=19  witness: IVVFP, rounds: 100000
time algorithm: 180 ms
mass found.  l=185, r=189  witness: IGVLG, rounds: 100000
time algorithm: 1953 ms
mass found.  l=187, r=191  witness: VLGIS, rounds: 100000
time algorithm: 1933 ms
mass found.  l=481, r=485  witness: GIYPD, rounds: 100000
time algorithm: 4897 ms
mass found.  l=221, r=226  witness: GFLIAF, rounds: 100000
time algorithm: 2243 ms
mass found.  l=39, r=48  witness: WYTICICILE, rounds: 100000
time algorithm: 451 ms
mass found.  l=140, r=149  witness: LLFFIMWELE, rounds: 100000
time algorithm: 1492 ms
mass found.  l=375, r=384  witness: SIPKIFTMFT, rounds: 100000
time algorithm: 3855 ms
mass found.  l=508, r=517  witness: VFQSFILGNL, rounds: 100000
time algorithm: 5158 ms
mass found.  l=338, r=349  witness: LQIISHGFIGAA, rounds: 100000
time algorithm: 3415 ms
mass not found, rounds: 100000
time algorithm: 7280 ms
mass not found, rounds: 100000
time algorithm: 7511 ms
mass not found, rounds: 100000
time algorithm: 7401 ms
mass not found, rounds: 100000
time algorithm: 7340 ms
mass not found, rounds: 100000
```

```
time algorithm: 7070 ms
mass not found, rounds: 100000
time algorithm: 7371 ms
mass not found, rounds: 100000
time algorithm: 7210 ms
mass not found, rounds: 100000
time algorithm: 7561 ms
mass not found, rounds: 100000
time algorithm: 7130 ms
mass not found, rounds: 100000
time algorithm: 7321 ms

************************************************************Binsearch
number of masses calculated: 281625
time preprocessing: 681 ms
mass found, rounds: 1000000
Time Algorithm: 1001 ms
mass found, rounds: 1000000
Time Algorithm: 1032 ms
mass found, rounds: 1000000
Time Algorithm: 871 ms
mass found, rounds: 1000000
Time Algorithm: 941 ms
mass found, rounds: 1000000
Time Algorithm: 772 ms
mass found, rounds: 1000000
Time Algorithm: 1101 ms
mass found, rounds: 1000000
Time Algorithm: 1132 ms
mass found, rounds: 1000000
Time Algorithm: 1151 ms
mass found, rounds: 1000000
Time Algorithm: 1082 ms
mass found, rounds: 1000000
Time Algorithm: 941 ms
mass not found, rounds: 1000000
Time Algorithm: 1222 ms
mass not found, rounds: 1000000
Time Algorithm: 1182 ms
mass not found, rounds: 1000000
Time Algorithm: 1151 ms
mass not found, rounds: 1000000
Time Algorithm: 1122 ms
mass not found, rounds: 1000000
Time Algorithm: 1152 ms
mass not found, rounds: 1000000
Time Algorithm: 1161 ms
mass not found, rounds: 1000000
Time Algorithm: 1182 ms
mass not found, rounds: 1000000
Time Algorithm: 1202 ms
mass not found, rounds: 1000000
Time Algorithm: 1161 ms
mass not found, rounds: 1000000
Time Algorithm: 1192 ms

************************************************************Cluster
 level1, number of submasses calculated: 281625
 submasses and indices created, time: 1743 ms
  level2, number of different submasses: 248767
  submasses and indices arranged, time: 2013 ms
   level3, number of intervals: 49751, indices per interval: 5
   intervals and indices created, time: 1542 ms
total time preprocessing: 5298 ms
mass found.  l=15, r=19  witness: IVVFP, rounds: 100000
time algorithm: 330 ms
mass found.  l=185, r=189  witness: IGVLG, rounds: 100000
time algorithm: 401 ms
mass found.  l=187, r=191  witness: VLGIS, rounds: 100000
time algorithm: 240 ms
mass found.  l=481, r=485  witness: GIYPD, rounds: 100000
time algorithm: 251 ms
mass found.  l=221, r=226  witness: GFLIAF, rounds: 100000
time algorithm: 450 ms
mass found.  l=39, r=48  witness: WYTICICILE, rounds: 100000
time algorithm: 621 ms
mass found.  l=140, r=149  witness: LLFFIMWELE, rounds: 100000
time algorithm: 841 ms
```

```
mass found.  l=375, r=384  witness: SIPKIFTMFT, rounds: 100000
time algorithm: 511 ms
mass found.  l=508, r=517  witness: VFQSFILGNL, rounds: 100000
time algorithm: 230 ms
mass found.  l=338, r=349  witness: LQIISHGFIGAA, rounds: 100000
time algorithm: 832 ms
mass not found, rounds: 100000
time algorithm: 861 ms
mass not found, rounds: 100000
time algorithm: 160 ms
mass not found, rounds: 100000
time algorithm: 871 ms
mass not found, rounds: 100000
time algorithm: 902 ms
mass not found, rounds: 100000
time algorithm: 881 ms
mass not found, rounds: 100000
time algorithm: 921 ms
mass not found, rounds: 100000
time algorithm: 791 ms
mass not found, rounds: 100000
time algorithm: 842 ms
mass not found, rounds: 100000
time algorithm: 861 ms
mass not found, rounds: 100000
time algorithm: 901 ms

************************************************************Cluster
 level1, number of submasses calculated: 281625
 submasses and indices created, time: 1793 ms
  level2, number of different submasses: 248767
  submasses and indices arranged, time: 2213 ms
   level3, number of intervals: 24872, indices per interval: 10
   intervals and indices created, time: 551 ms
total time preprocessing: 4557 ms
mass found.  l=15, r=19  witness: IVVFP, rounds: 100000
time algorithm: 941 ms
mass found.  l=185, r=189  witness: IGVLG, rounds: 100000
time algorithm: 1061 ms
mass found.  l=187, r=191  witness: VLGIS, rounds: 100000
time algorithm: 271 ms
mass found.  l=481, r=485  witness: GIYPD, rounds: 100000
time algorithm: 260 ms
mass found.  l=221, r=226  witness: GFLIAF, rounds: 100000
time algorithm: 1002 ms
mass found.  l=39, r=48  witness: WYTICICILE, rounds: 100000
time algorithm: 631 ms
mass found.  l=140, r=149  witness: LLFFIMWELE, rounds: 100000
time algorithm: 1542 ms
mass found.  l=375, r=384  witness: SIPKIFTMFT, rounds: 100000
time algorithm: 480 ms
mass found.  l=508, r=517  witness: VFQSFILGNL, rounds: 100000
time algorithm: 241 ms
mass found.  l=338, r=349  witness: LQIISHGFIGAA, rounds: 100000
time algorithm: 761 ms
mass not found, rounds: 100000
time algorithm: 1582 ms
mass not found, rounds: 100000
time algorithm: 150 ms
mass not found, rounds: 100000
time algorithm: 1563 ms
mass not found, rounds: 100000
time algorithm: 1592 ms
mass not found, rounds: 100000
time algorithm: 1582 ms
mass not found, rounds: 100000
time algorithm: 1572 ms
mass not found, rounds: 100000
time algorithm: 1433 ms
mass not found, rounds: 100000
time algorithm: 1572 ms
mass not found, rounds: 100000
time algorithm: 1502 ms
mass not found, rounds: 100000
time algorithm: 1592 ms

************************************************************Cluster
 level1, number of submasses calculated: 281625
```

```
 submasses and indices created, time: 1733 ms
  level2, number of different submasses: 248767
  submasses and indices arranged, time: 2203 ms
   level3, number of intervals: 12429, indices per interval: 20
   intervals and indices created, time: 801 ms
total time preprocessing: 4737 ms
mass found.  l=15, r=19  witness: IVVFP, rounds: 100000
time algorithm: 551 ms
mass found.  l=185, r=189  witness: IGVLG, rounds: 100000
time algorithm: 2043 ms
mass found.  l=187, r=191  witness: VLGIS, rounds: 100000
time algorithm: 2563 ms
mass found.  l=481, r=485  witness: GIYPD, rounds: 100000
time algorithm: 1172 ms
mass found.  l=221, r=226  witness: GFLIAF, rounds: 100000
time algorithm: 631 ms
mass found.  l=39, r=48  witness: WYTICICILE, rounds: 100000
time algorithm: 220 ms
mass found.  l=140, r=149  witness: LLFFIMWELE, rounds: 100000
time algorithm: 2434 ms
mass found.  l=375, r=384  witness: SIPKIFTMFT, rounds: 100000
time algorithm: 1522 ms
mass found.  l=508, r=517  witness: VFQSFILGNL, rounds: 100000
time algorithm: 1172 ms
mass found.  l=338, r=349  witness: LQIISHGFIGAA, rounds: 100000
time algorithm: 390 ms
mass not found, rounds: 100000
time algorithm: 2914 ms
mass not found, rounds: 100000
time algorithm: 2975 ms
mass not found, rounds: 100000
time algorithm: 2884 ms
mass not found, rounds: 100000
time algorithm: 2904 ms
mass not found, rounds: 100000
time algorithm: 2924 ms
mass not found, rounds: 100000
time algorithm: 3025 ms
mass not found, rounds: 100000
time algorithm: 2944 ms
mass not found, rounds: 100000
time algorithm: 2884 ms
mass not found, rounds: 100000
time algorithm: 3054 ms
mass not found, rounds: 100000
time algorithm: 2864 ms
```

TEST: mitoch750.txt

```
************************************************************Linsearch
mass found.  l=25, r=29  witness: ESVSR, rounds: 100000
time algorithm: 340 ms
mass found.  l=676, r=680  witness: FVLNL, rounds: 100000
time algorithm: 6970 ms
mass found.  l=411, r=415  witness: LDALQ, rounds: 100000
time algorithm: 4156 ms
mass found.  l=253, r=257  witness: QVNAR, rounds: 100000
time algorithm: 2614 ms
mass found.  l=182, r=186  witness: RKAQI, rounds: 100000
time algorithm: 1893 ms
mass found.  l=41, r=50  witness: FSICLFRSLK, rounds: 100000
time algorithm: 470 ms
mass found.  l=232, r=241  witness: GTARCERWNP, rounds: 100000
time algorithm: 2384 ms
mass found.  l=738, r=747  witness: IDGKGALSDH, rounds: 100000
time algorithm: 7541 ms
mass found.  l=540, r=549  witness: IRNLQRELDH, rounds: 100000
time algorithm: 5507 ms
mass found.  l=290, r=299  witness: FIASFDLSAA, rounds: 100000
time algorithm: 2955 ms
mass not found, rounds: 100000
time algorithm: 7340 ms
mass not found, rounds: 100000
time algorithm: 7521 ms
mass not found, rounds: 100000
```

```
time algorithm: 7381 ms
mass not found, rounds: 100000
time algorithm: 7250 ms
mass not found, rounds: 100000
time algorithm: 7030 ms
mass not found, rounds: 100000
time algorithm: 7421 ms
mass not found, rounds: 100000
time algorithm: 7180 ms
mass not found, rounds: 100000
time algorithm: 7451 ms
mass not found, rounds: 100000
time algorithm: 7140 ms
mass not found, rounds: 100000
time algorithm: 7541 ms

***********************************************************Binsearch
number of masses calculated: 281625
time preprocessing: 280 ms
mass found, rounds: 1000000
Time Algorithm: 1042 ms
mass found, rounds: 1000000
Time Algorithm: 1152 ms
mass found, rounds: 1000000
Time Algorithm: 1081 ms
mass found, rounds: 1000000
Time Algorithm: 1082 ms
mass found, rounds: 1000000
Time Algorithm: 981 ms
mass found, rounds: 1000000
Time Algorithm: 901 ms
mass found, rounds: 1000000
Time Algorithm: 1162 ms
mass found, rounds: 1000000
Time Algorithm: 1122 ms
mass found, rounds: 1000000
Time Algorithm: 1101 ms
mass found, rounds: 1000000
Time Algorithm: 1082 ms
mass not found, rounds: 1000000
Time Algorithm: 1162 ms
mass not found, rounds: 1000000
Time Algorithm: 1231 ms
mass not found, rounds: 1000000
Time Algorithm: 1172 ms
mass not found, rounds: 1000000
Time Algorithm: 1132 ms
mass not found, rounds: 1000000
Time Algorithm: 1131 ms
mass not found, rounds: 1000000
Time Algorithm: 1182 ms
mass not found, rounds: 1000000
Time Algorithm: 1182 ms
mass not found, rounds: 1000000
Time Algorithm: 1121 ms
mass not found, rounds: 1000000
Time Algorithm: 1152 ms
mass not found, rounds: 1000000
Time Algorithm: 1142 ms

***********************************************************Cluster
 level1, number of submasses calculated: 281625
 submasses and indices created, time: 1722 ms
  level2, number of different submasses: 256211
  submasses and indices arranged, time: 2884 ms
   level3, number of intervals: 51240, indices per interval: 5
   intervals and indices created, time: 461 ms
total time preprocessing: 5067 ms
mass found.  l=25, r=29  witness: ESVSR, rounds: 100000
time algorithm: 441 ms
mass found.  l=676, r=680  witness: FVLNL, rounds: 100000
time algorithm: 240 ms
mass found.  l=411, r=415  witness: LDALQ, rounds: 100000
time algorithm: 811 ms
mass found.  l=253, r=257  witness: QVNAR, rounds: 100000
time algorithm: 811 ms
mass found.  l=182, r=186  witness: RKAQI, rounds: 100000
time algorithm: 841 ms
```

```
mass found.  l=41, r=50  witness: FSICLFRSLK, rounds: 100000
time algorithm: 261 ms
mass found.  l=232, r=241  witness: GTARCERWNP, rounds: 100000
time algorithm: 831 ms
mass found.  l=738, r=747  witness: IDGKGALSDH, rounds: 100000
time algorithm: 371 ms
mass found.  l=540, r=549  witness: IRNLQRELDH, rounds: 100000
time algorithm: 250 ms
mass found.  l=290, r=299  witness: FIASFDLSAA, rounds: 100000
time algorithm: 410 ms
mass not found, rounds: 100000
time algorithm: 842 ms
mass not found, rounds: 100000
time algorithm: 831 ms
mass not found, rounds: 100000
time algorithm: 150 ms
mass not found, rounds: 100000
time algorithm: 140 ms
mass not found, rounds: 100000
time algorithm: 862 ms
mass not found, rounds: 100000
time algorithm: 881 ms
mass not found, rounds: 100000
time algorithm: 851 ms
mass not found, rounds: 100000
time algorithm: 871 ms
mass not found, rounds: 100000
time algorithm: 862 ms
mass not found, rounds: 100000
time algorithm: 811 ms

************************************************************Cluster
 level1, number of submasses calculated: 281625
 submasses and indices created, time: 1712 ms
  level2, number of different submasses: 256211
  submasses and indices arranged, time: 2213 ms
   level3, number of intervals: 25615, indices per interval: 10
   intervals and indices created, time: 551 ms
total time preprocessing: 4476 ms
mass found.  l=25, r=29  witness: ESVSR, rounds: 100000
time algorithm: 1022 ms
mass found.  l=676, r=680  witness: FVLNL, rounds: 100000
time algorithm: 230 ms
mass found.  l=411, r=415  witness: LDALQ, rounds: 100000
time algorithm: 1462 ms
mass found.  l=253, r=257  witness: QVNAR, rounds: 100000
time algorithm: 1362 ms
mass found.  l=182, r=186  witness: RKAQI, rounds: 100000
time algorithm: 811 ms
mass found.  l=41, r=50  witness: FSICLFRSLK, rounds: 100000
time algorithm: 821 ms
mass found.  l=232, r=241  witness: GTARCERWNP, rounds: 100000
time algorithm: 812 ms
mass found.  l=738, r=747  witness: IDGKGALSDH, rounds: 100000
time algorithm: 1111 ms
mass found.  l=540, r=549  witness: IRNLQRELDH, rounds: 100000
time algorithm: 261 ms
mass found.  l=290, r=299  witness: FIASFDLSAA, rounds: 100000
time algorithm: 420 ms
mass not found, rounds: 100000
time algorithm: 1432 ms
mass not found, rounds: 100000
time algorithm: 1462 ms
mass not found, rounds: 100000
time algorithm: 141 ms
mass not found, rounds: 100000
time algorithm: 1602 ms
mass not found, rounds: 100000
time algorithm: 1592 ms
mass not found, rounds: 100000
time algorithm: 1683 ms
mass not found, rounds: 100000
time algorithm: 1562 ms
mass not found, rounds: 100000
time algorithm: 1522 ms
mass not found, rounds: 100000
time algorithm: 1582 ms
mass not found, rounds: 100000
```

```
time algorithm: 1492 ms

**********************************************************Cluster
 level1, number of submasses calculated: 281625
 submasses and indices created, time: 1712 ms
  level2, number of different submasses: 256211
  submasses and indices arranged, time: 2214 ms
   level3, number of intervals: 12800, indices per interval: 20
   intervals and indices created, time: 801 ms
total time preprocessing: 4727 ms
mass found.  l=25, r=29  witness: ESVSR, rounds: 100000
time algorithm: 2233 ms
mass found.  l=676, r=680  witness: FVLNL, rounds: 100000
time algorithm: 1482 ms
mass found.  l=411, r=415  witness: LDALQ, rounds: 100000
time algorithm: 2604 ms
mass found.  l=253, r=257  witness: QVNAR, rounds: 100000
time algorithm: 1222 ms
mass found.  l=182, r=186  witness: RKAQI, rounds: 100000
time algorithm: 681 ms
mass found.  l=41, r=50  witness: FSICLFRSLK, rounds: 100000
time algorithm: 691 ms
mass found.  l=232, r=241  witness: GTARCERWNP, rounds: 100000
time algorithm: 1962 ms
mass found.  l=738, r=747  witness: IDGKGALSDH, rounds: 100000
time algorithm: 2264 ms
mass found.  l=540, r=549  witness: IRNLQRELDH, rounds: 100000
time algorithm: 2904 ms
mass found.  l=290, r=299  witness: FIASFDLSAA, rounds: 100000
time algorithm: 260 ms
mass not found, rounds: 100000
time algorithm: 2884 ms
mass not found, rounds: 100000
time algorithm: 2784 ms
mass not found, rounds: 100000
time algorithm: 3055 ms
mass not found, rounds: 100000
time algorithm: 3034 ms
mass not found, rounds: 100000
time algorithm: 2914 ms
mass not found, rounds: 100000
time algorithm: 3075 ms
mass not found, rounds: 100000
time algorithm: 2884 ms
mass not found, rounds: 100000
time algorithm: 2904 ms
mass not found, rounds: 100000
time algorithm: 3044 ms
mass not found, rounds: 100000
time algorithm: 2875 ms
```