Übungen zur Algorithmischen Bioinformatik I

Hausarbeit 2

Xiheng He

Mai 2021

(a) **Komplexität**

(i) Erklären Sie den Beweis der unteren Schranke fur Sortieralgorithmen mit Vergleichsoperationen!
Gegeben sei genau n unterschiedliche Zahlen. In best-case wurden sie alle schon sortiert damit wird dies nicht betrachtet. In worst-case sind alle nicht sortiert dann nehmen wir an, dass die Algorithmen den richtige Ergebnis liefern nur dann wenn sie x Schritte berechnet haben nämlich 2^x Vergleichsoperationen durchführen müssen, weil für jeden Schritt zwei Elemente verglichen werden deshalb es nur zwei mögliche Fälle für jeden Vergleich gibt.
Ferner sind theoritsch auch $n!$ Permutationen für solche Sotierprobleme vorhanden und Algorithmen benötigen höchstens 2^x Vergleichsoperationen um die einzige richtige Permutation herauszufinden. Deshalb folgt:

$$
\begin{aligned}
2^x \geq n! \Longrightarrow x &\geq \log(n!) \\
&= \log n + \log(n-1) + \log(n-2) + \cdots + \log 2 \\
&= \sum_{i=2}^{n} \log i \\
&= \sum_{i=2}^{n/2-1} \log i + \sum_{i=n/2}^{n} \log i \\
&\geq 0 + \sum_{i=n/2}^{n} \log \frac{n}{2} \qquad\qquad (\sum_{i=n/2}^{n} \log i \geq \sum_{i=n/2}^{n} \log \frac{n}{2}) \\
&\geq \frac{n}{2} \cdot \log \frac{n}{2} \\
&= \frac{n}{2} \cdot \frac{\log n}{\log 2} \\
&= \Theta(n \log n) \\
&\Longrightarrow x = \Omega(n \log n)
\end{aligned}
$$

Daraus können wir Schlussfolgerungen ziehen, dass die untere Schranke für Sortieralgorithmen $\Omega(n \log n)$ ist.

(ii) Wieso ist das Sortier-Problem ein "closed problem"?
Wie es in (i) schon gezeigt haben Sortieralgorithmen einerseits eine inhärente Laufzeit-Komplexität (z.B. Mergesorts is $O(n \log n)$) und andererseits eine untere Schranke $\Omega(n \log n)$. Deshalb ist Sortier-Problem "closed problem".

(iii) Es gibt lineare Algorithmen für das Sortier-Problem. Welche? Wieso? Welche zusätzlichen Annahmen müssen dazu gemacht werden?

Lineare Algorithmen: Bucketsort,Countingsort, Radixsort

Weil solche lineare Sortieralgorithmen nicht total auf Vergleichssortierung basiert sind. Die Laufzeiten der Algorithmen können zwar durch erhöhten Platzbedarf reduziert werden jedoch können diese Algorithmen nur unter bestimmen Umständen linear laufen.

Annahme: für alle Sortieralgorithmen, die auf Vergleich und Austausch der Elemente basieren, bleiben die untere Schranke in $\Omega(n \log n)$.

(iv) Was weiss man über das average-case Verhalten der jeweiligen Algorithmen?

Durch Laufzeitanalyse in average-case können wir diese Algorithmus evaluieren. Im best-case erhalten wir die beste Leistung dieses Algorithmus, die obere Schranke der Algorithmusleistung darstellt und im worst-case erhalten wir die schlechteste Leistung dieses Algorithmus, der auch untere Schranke darstellt. Manche Algorithmen die ausgezeichnete Laufzeit im best-case besitzen, funktionieren in tatsächlichen Gebrauch nicht immer ideal denn die Leistungen im worst-case sind jedoch mittelmäßig. Wenn ein Algorithmus in average-case läuft ziemlich wie in best-case und worst-case können wir einen solchen Algorithmus als "Stabil" evaluieren. Zusammengefasst, können wir durch average-case Verhalten die "typische" Leistung des Algorithmus erhalten.

(b) **Implementierung**

Implementieren Sie mehrere Sortieralgorithmen Ihrer Wahl inklusive (vielleicht von jeder Klasse eine?). Es gibt viele verfugbare Implementierungen, libraries bzw. tools für diese Algorithmen. Es ist vielleicht instruktiv eigene mit Standard-Implementierungen zu vergleichen.

(i) (simple): z.B. INSERTION-SORT, **BUBBLE-SORT**, ...

(ii) (rekursiv): z.B. **MERGE-SORT**, QUICK-SORT, ...

(iii) (linear): z.B. COUNTING-SORT, **BUCKET-SORT**, RADIX-SORT, ...

(c) **Benchmark**

Evaluieren und vergleichen Sie die Effizienz Ihrer Implementierungen (verwenden Sie geeignete timing Funktionen, achten Sie auf startup times z.B. der Java VM).

(i) z.B. durch Timing auf unterschiedlich großen input Files.

Für große Eingabe ist **BUCKET-SORT** optimal, für kleine Eingabe sind **BUBBLE-SORT** und **BUCKET-SORT** optimal, da **MERGE-SORT** die Eingabe zuerst teilen muss. Für beliebige Eingabegröße läuft **MERGE-SORT** stabilste.

input size	$n = 10^2$	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$
BUBBLE-SORT	0ms	11ms	159ms	12742ms	1264231ms
MERGE-SORT	4ms	1ms	2ms	23ms	212ms
BUCKET-SORT	0ms	0ms	1ms	2ms	26ms

(ii) Wie passt das Timing zur theoretischen Laufzeitanalyse

BUBBLE-SORT: $\frac{1264231}{12742} = 99.2 \approx (\frac{10^6}{10^5})^2 = 100$

MERGE-SORT: $\frac{23}{2} = 11.5 \approx \frac{10^5 \cdot \log 10^5}{10^4 \cdot \log 10^4} = 12.5$

BUCKET-SORT: $\frac{26}{2} = 13 \approx \frac{10^6 + k_1}{10^5 + k_2}$ $(k_1 > k_2)$

Die tatsächliche Laufzeit ist unterschiedlich von der theroetische Laufzeit Komplexität jedoch ungefähr gleich.

(iii) Können die linearen Algorithmen die Effizienz deutlich verbessern? In welchen Fällen?
Die Laufzeit Komplexität des Bucketsort Algorithmus in worst-case liegt in $O(n^2)$ und in average-case liegt in $O(n+k)$ wobei k die Anzahl der Buckets entspricht. Deshalb kann dieser Algorithmus näturlich deutlich verbessert werden wenn es bestimmt wird, für welche Fälle sich zum Einsatz eignet. Bucketsort eignet sich hauptsächlich für gleichmäßig verteilte numerische Arrays. In diesem Fall kann eine maximale Effizienz erzielt werden.

(iv) Vergleichen Sie Ihre Implementierung mit dem bekannten Unix-tool sort!

- Unix kann alphabetisch und absteigend sortieren. Meine eigene Implementierung kann nur Nummern aufsteigend sortieren.
- Unix kann im Vergleich zu meiner Implementierung auch Groß- und Kleinschreibung ignorieren und die Sortierergebnisse de-duplizieren.
- Unix-Sort verwendet auch externen Speicherplatz, um die Effizienz der Sortierung zu optimieren.

(v) Was können Sie über das Laufzeitverhalten von Unix sort schließen. Mit welchem Algorithmus könnte sort implementiert sein?
Unix sort benutz externen Speicherplatz für Optimierung der Effizienz und der Algorithmus muss unter allen Umständen stabil sein und immer eine optimale Effizienz anbieten. D.h, die Laufzeit muss in worst-case und best-case (oder average-case) ungefähr gleich sein und er muss mindestens in $O(n \log n)$ da $O(n^2)$ nicht optimal und lineare Sortieralgorithmen nicht immer die beste Effizienz bieten. Laufzeit in $O(n \log n)$ und der Algorithmus stabil, vermutlich wurde mit Merge-Sort implementiert.
(Im Verzeichnis coreutils kann man ein sort.c Datei finden. Eine der Funktionen heißt MAX_MERGE.)

(vi) Wie vergleichen sich die Laufzeiten Ihrer Implementierungen mit Unix sort?
Wie in Java kann die Laufzeit durch Bash Skripten bestimmt werden. z.B
start=$(date +%s)
sort -n $file
end=$(date +%s)
take=$((end - start))

(d) **Bioinformatik-Anwendung**

Es gibt kaum ein Verfahren, dass so vielfältige Anwendungen in der Bioinformatik hat wie Sortieren. Oft werden Proteinsequenzen und -strukturen durchsucht und gescored und es mussen sortierte Listen der Ergebnisse ausgegeben oder weiterverarbeitet werden.

(i) Unix sort kann dabei eine ganze Reihe von Aufgaben erledigen: Wozu könnte z.B. das bekannte "Unix-Pattern": sort | uniq -c | sort -rn dienen? Diskutieren Sie Anwendungen und ähnlich nutzliche "Pattern". sort hat viele Optionen, cut -f sowie head, tail und grep können auch sehr dienlich sein (von awk ganz zu schweigen).

sort: sortieren Datei
uniq -c: zeigen die Anzahl der Vorkommen der Zeile sort -rn: sortieren absteigend nach Werte Es dient dazu z.B, zeigen alle wiederholten Zeilen oder Pattern (z.B subsequence, motif) in der Datei und ordnen diese nach Anzahl der Vorkommen.
z.B: ps -e -o "%C : %p : %z : %a" | sort –nr, das Pattern kann nach CPU-Auslastung absteigend sortieren.

(ii) Wie passt das Timing von sort zur theoretischen Laufzeitanalyse?
Die Laufzeit von Unix Sort liegt nahe bei $O(n \log n)$, ungefähr gleich zur theoretischen Laufzeitanalyse.

(iii) Es wurden schon SAM/BAM Files erwähnt. Beschreiben Sie kurz was SAM/BAM files sind, was der Unterschied von SAM und BAM ist, und wozu sie dienen! Warum sollten/müssen die sortiert werden?
Das SAM-Format (Sequence Alignment / Map) ist ein generisches Format zum Speichern großer Nukleotidsequenz-Alignments. Es besteht aus header section und alignment section. BAM ist das Binärformat von SAM, daher sind die beiden Formate identisch.

- SAM ist ein text-based Format und BAM ist das entsprechende Binärformat.
- BAM-Dateien sind normalerweise komprimiert daher benötigt weniger Speicherplatt und für die Arbeit mit Software effizienter als SAM.

(iv) Es kann nicht schaden, sich etwas (Achtung: das kann umfangreich werden, das Format ist recht komplex! Siehe https://samtools.github.io/hts-specs/SAMv1.pdf), also etwas uber SAM/BAM-Files zu informieren.
Fur Benchmarking arbeiten Sie mit .sam Files. Wie bekommen Sie .sam Files?

- Alignments können als .sam Files ausgegeben werden. z.B Mittels Alignment software bowtie2 (bowtie2 -1 aln1.fasta -2 aln2.fasta -S exp.sam) kann .sam File erzeugt werden.
- Wenn .bam File vorhanden ist, kann .sam File durch: samtools view exp.bam -O SAM > exp.sam konvertiert werden.

(v) Betrachten Sie hier (unkomprimierte) SAM-files (normalerweise speichert und arbeitet man meist mit geeigneten Tools auf BAM files!) und nutzen Sie diese fur das Benchmarking Ihrer Implementierungen und den Vergleich mit Standardtools wie Unix sort und **samtools**. Das Beispiel-File ist ungefähr 4GB groß und enthält ungefähr 16 Mio Zeilen. Es kann je nachdem (Zahlen, Sequenzen, Zeilen) auf einem Laptop in 10s, 1:20m, 10m sortiert werden (Achtung: hängt auch vom verfügbaren Hauptpeicher ab!).

- **Unix sort** und **samtools** können alphabetisch sortieren.
- **Unix sort** und **samtools** können z.B den Verbrauch des Internspeichers anpassen.
- **Unix sort** und **samtools** können effizienter als merge-sort Standardimplementierung sortieren.

(vi) Was können Sie über das Laufzeitverhalten von **samtools** sort schließen? Mit welchem Algorithmus könnte **samtools** sort implementiert sein?
Die Laufzeit solltet für große Eingaben möglichst effizient sein. Daher solltet sie nicht in $O(n^k)$ oder $O(n + k)$ sein. **samtools sort** könnte als heap-based merge sort implementiert sein.