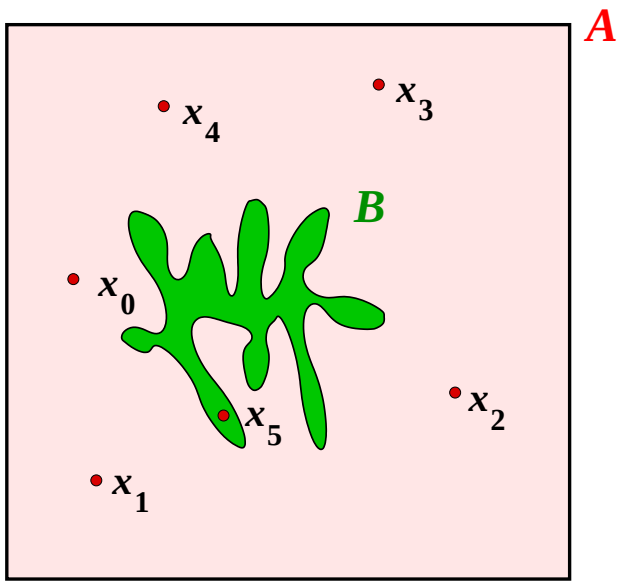
4 Randomisierte Suche

4.1 Suche mit wenigen Zufallsbits

Wir stellen uns folgendes Grundproblem vor: Gegeben ist eine endliche Menge A mit einer nichtleeren Teilmenge $B \subseteq A$.



Wir suchen (irgend)ein Element von B und haben hierfür nur die folgenden Operationen zur Verfügung:

- wir können alle Elemente der Menge A systematisch aufzählen;
- wir können ein Element von A zufällig wählen;
- für gegebenes $x \in A$ können wir einen Test „ist $x \in B$?“ durchführen.

Algorithmus 4.1.1 (Zufällige Suche)**.**
INPUT: Menge A mit unbekannter Teilmenge B
METHODE:
1 **repeat**
2 `x` := zufälliges Element von A
3 **until** `x` $\in B$;
4 **return** `x`.

Die Anzahl der durchgeführten Tests bestimmt die Kosten für die Erzeugung eines Elements von B.

Wir nehmen an, dass über die Lage von B in A keinerlei Information vorliegt.

Beispiel: Wir wollen im Bereich der k-Bit-Zahlen eine Primzahl finden.

$$A = \{n \in \mathbb{N} \mid 2^{k-1} \leq n < 2^k\}.$$
$$B = \{p \in A \mid p \text{ ist Primzahl}\}.$$

Wie in der Vorlesung „Grundlagen der Kryptographie“ im Detail besprochen wird, gibt es einen sehr effizienten (randomisierten) Primzahltest, der es also erlaubt, Zahlen $n \in A$ darauf zu testen, ob sie in B sind. Ein Verfahren, direkt eine Primzahl in A zu konstruieren, kennt man dagegen nicht. Das Verhältnis $\varrho = |B|/|A|$ ist in diesem Fall $\Theta(1/k)$, wie wir später noch genauer diskutieren.

Wenn wir A aufzählen, könnte es passieren, dass wir zuerst alle Elemente von $A - B$ sehen, dann erst ein Element von B. Dies ist in eigentlich allen Situationen unbefriedigend. Man stelle sich vor, dass etwa jedes zweite Element von A zu B gehört. Dann würde man im schlimmsten Fall $\frac{1}{2}|A| + 1$ Tests durchführen, bis man das erste Element von B findet. Attraktiver ist ein randomisiertes Vorgehen, das wir als nächstes beschreiben. Dieses würde in der beschriebenen Situation im Schnitt nur 2 Versuche brauchen.

Wir erkennen, dass dieser Algorithmus endlos lange läuft, wenn $B = \emptyset$ ist; diesen Fall müssen wir also ausschließen.

Wir analysieren die erwartete Laufzeit, die hier im wesentlichen durch die Anzahl der Schleifendurchläufe bestimmt ist. Mit

$$\varrho := \frac{|B|}{|A|}$$

(gesprochen: „rho“) bezeichnen wir die „Dichte“ von B in A. Seien $X_0, X_1, X_2, \ldots$ die gewählten Elemente (Zufallsobjekte in A), und sei Z die Anzahl der Schleifendurchläufe (Zufallsvariable). Diese Zufallsvariable ist offensichtlich geometrisch verteilt[1] Wir haben:

$$\mathbf{Pr}(Z \geq k+1) = \mathbf{Pr}(X_0 \notin B \wedge \ldots \wedge X_{k-1} \notin B) = (1-\varrho)^k. \tag{1}$$

Nach Fakt 2.2.4 aus den „Wahrscheinlichkeitstheoretischen Grundlagen“ ist

$$\mathbf{E}(Z) = \sum_{k\geq 0} \mathbf{Pr}(Z \geq k+1) = \sum_{k\geq 0} (1-\varrho)^k = \frac{1}{1-(1-\varrho)} = \frac{1}{\varrho}. \tag{2}$$

Wir wollen in diesem Abschnitt auch Zufallsexperimente als Ressource betrachten. Um fair zu messen, benutzen wir die Anzahl der Zufallsbits als Maß (nicht die Anzahl der Zufallszahlen, die eine Vielzahl von Bits auf einen Schlag liefern). Um eine Zahl aus A zufällig zu wählen, müssen wir eine $\lceil \log |A| \rceil$-Bit-Zahl (Index in A) wählen.

Proposition 4.1.2. *Zufällige Suche wie in Algorithmus 4.1.1 beschrieben kostet im erwarteten Fall $1/\varrho$ Tests „ist $x \in B$?“; die erwartete Zahl von Zufallsbits ist $\lceil \log |A| \rceil / \varrho$.*

Wenn die Dichte ϱ klein ist, wie etwa im Fall der Suche nach einer k-ziffrigen Primzahl, kann die Zahl der Zufallsbits erheblich sein.

Wir überlegen im folgenden, wie wir mit relativ wenigen Zufallsbits fast ebenso effizient randomisiert suchen können. Eine etwas allgemeinere Interpretation der Situation ist die eines „*special purpose*-Pseudozufallszahlengenerators“. Eigentlich benötigen wir eine lange Folge $X_0, X_1, \ldots,$ von zufälligen Elementen von A. Um Zufallsbits zu sparen, wählen wir einen kurzen zufälligen „*seed*“ (das deutsche „Samenkorn“ ist ungebräuchlich) und erzeugen daraus algorithmisch eine ganze Folge von Elementen von A. Diese Folge ist nicht mehr vollständig zufällig, sondern nur „pseudozufällig“. Im konkreten Fall der Suche nach einem Element von B in A können wir das Verhalten unseres Verfahrens genau analysieren.

Für unsere Konstruktion nehmen wir an, dass A eine algebraische Struktur hat, nämlich dass A ein endlicher Körper ist, zum Beispiel $A = \mathbb{Z}_p$ für eine Primzahl p.[2] Die

[1] Eine Zufallsvariable $Z\colon \Omega \to \mathbb{N}$ heißt *geometrisch verteilt mit Parameter* $p \in [0,1]$, wenn gilt: $\mathbf{Pr}(X=0) = 0$ und $\mathbf{Pr}(X=i) = (1-p)^{i-1}p$, für $i \geq 1$.

[2] Sollte dies nicht der Fall sein, nummerieren wir die Elemente von A durch, stellen uns also $A = \{0, \ldots, |A|-1\}$ vor, suchen eine Primzahl $p \in [|A|, 2|A|]$ (eine solche existiert nach einem Satz der Zahlentheorie) und arbeiten mit $A' = \mathbb{Z}_p$. Die neue Dichte $\varrho' = \frac{|B|}{|A'|}$ unterscheidet sich höchstens um den Faktor 2 von ϱ.

Algorithmus 4.1.3 (Paarweise unabhängige Suche)**.**
INPUT: Menge $A = \mathbb{Z}_p$ (Körper) mit unbekannter, nichtleerer Teilmenge B
METHODE:
1 Wähle r und s aus $\mathbb{Z}_p$ zufällig;
2 Falls $r = 0$, setze $r := 1$; // erzwingt kompletten Durchlauf durch A
3 $\mathtt{x} := (s - r) \bmod p$;
4 **repeat**
5 $\quad \mathtt{x} := (\mathtt{x} + r) \bmod p$;
6 **until** $\mathtt{x} \in B$;
7 **return** x.

Idee ist, zwei Elemente r und s in $\mathbb{Z}_p$ zufällig zu wählen und dann die Folge

$$X_0 = r \cdot 0 + s, X_1 = r \cdot 1 + s, X_2 = r \cdot 2 + s, X_3 = r \cdot 3 + s, \ldots \tag{3}$$

(Arithmetik in $\mathbb{Z}_p$) in A zu testen. Eine kleine Komplikation liegt darin, dass $r = 0$ sein könnte. Dann würde man immer wieder nur den Eintrag s testen. Dies fangen wir mit einer Sonderbehandlung ab, die die Elemente von A in der Reihenfolge $s, s+1, s+2, \ldots, p-1, 0, 1, \ldots, s-1$ betrachtet. Ansonsten erhalten wir eine Folge von mehr oder weniger zufälligen Elementen in A. Wenn wir in $\mathbb{Z}_p$ arbeiten, können wir programmiertechnisch die Elemente $X_0, X_1, \ldots$ ohne Multiplikation, nur durch iterierte Addition, erzeugen. Wir erhalten Algorithmus 4.1.3.

Analyse: Korrektheit. Wir stellen zunächst fest, dass beim Schleifendurchlauf r auf jeden Fall einen Wert $\neq 0$ hat. Dies führt dazu, dass die Folge $(X_i)_{i=0,\ldots,p-1}$, mit $X_i = r \cdot i + s$, jedes Element von $A = \mathbb{Z}_p$ genau einmal berührt. (Man muss nur überlegen, dass für jedes $a \in A$ die Gleichung $r \cdot i + s = a$ genau eine Lösung $i = (a - s) \cdot r^{-1}$ hat.) Also wird ein Element von B gefunden, und die Schleife endet erfolgreich. Für die Analyse stellen wir uns vor, dass der Wert $r = 0$ einfach beibehalten wird (damit die Mathematik glatter funktioniert). Algorithmus 4.1.3 findet ein Element von B keinesfalls später als dieser ideale Algorithmus.

Anzahl der Zufallsbits: Wir wählen zwei Elemente von A zufällig; es werden also $2\lceil \log |A| \rceil$ Zufallsbits benötigt.

Rechenzeit: Wir analysieren, was wir über die Anzahl der Schleifendurchläufe sagen können. Wir definieren: $X_i := r \cdot i + s$, $i = 0, 1, \ldots, p-1$, und

$$Y_i := \begin{cases} 1, & \text{falls } X_i \in B, \\ 0, & \text{andernfalls.} \end{cases}$$

Proposition 4.1.4. *Jede Zufallsvariable $X_i, 0 \leq i \leq p-1$, ist in A uniform verteilt, und $X_i, 0 \leq i \leq p-1$, sind* ***paarweise unabhängig****, d. h.: $i \neq j \Rightarrow X_i, X_j$ unabhängig.*

Beweis: Für festes i haben wir $\mathbf{Pr}(X_i = a) = p^{-1}$, weil es für jedes r genau ein s mit $r \cdot i + s = a$ gibt, nämlich $s = a - r \cdot i$. Es gilt

$$\mathbf{Pr}(X_i = a \wedge X_j = b) = \mathbf{Pr}(r \cdot i + s = a \wedge r \cdot j + s = b).$$

Nun hat das Gleichungssystem

$$\begin{pmatrix} i & 1 \\ j & 1 \end{pmatrix} \cdot \begin{pmatrix} r \\ s \end{pmatrix} = \begin{pmatrix} a \\ b \end{pmatrix}$$

genau eine Lösung $\binom{r}{s}$ über dem Körper $\mathbb{Z}_p$, weil die Matrix $\binom{i\,1}{j\,1}$ Determinante $i - j \neq 0$ hat. Das bedeutet, dass jeder Wert $\binom{a}{b}$ mit derselben Wahrscheinlichkeit p^{-2} angenommen wird. Damit:

$$\begin{aligned} &\mathbf{Pr}(X_i = a \wedge X_j = b) \\ &\quad = \mathbf{Pr}(r \cdot i + s = x \wedge r \cdot j + s = y) \\ &\quad = p^{-2} \\ &\quad = \mathbf{Pr}(r \cdot i + s = a) \cdot \mathbf{Pr}(r \cdot j + s = b), \end{aligned}$$

was die Unabhängigkeit beweist. □

Korollar 4.1.5. *Die Zufallsvariablen Y_i, $0 \leq i \leq p-1$, haben Erwartungswert $E(Y_i) = \varrho$ und sind ebenfalls paarweise unabhängig.*

Beweis: Weil X_i uniform verteilt ist, ist $\mathbf{E}(Y_i) = \mathbf{Pr}(Y_i = 1) = \mathbf{Pr}(X_i \in B) = \varrho$. Wenn $i \neq j$, dann sind X_i und X_j unabhängig, also auch Y_i und Y_j als Funktionen von X_i bzw. X_j. □

Nun definieren wir, für $0 \leq k \leq p$:

$$Z_k := Y_0 + \cdots + Y_{k-1}.$$

Das ist die Anzahl der Erfolge in den ersten k Runden (unter der Annahme, dass man auch nach einem Treffer weitermacht). Beachte: $Z_0 = 0$. □

Lemma 4.1.6. $\mathbf{E}(Z_k) = k\varrho$ *und* $\mathbf{Var}(Z_k) = k\varrho(1-\varrho)$.

Beweis: Wegen der Linearität des Erwartungswertes gilt:

$$\mathbf{E}(Z_k) = \mathbf{E}(Y_0) + \cdots + \mathbf{E}(Y_{k-1}) = k\varrho.$$

Weiter folgt aus der paarweisen Unabhängigkeit (Fakt 2.5.6(b) und die folgende Bemerkung): $\mathbf{Var}(Z_k) = \sum_{0 \le i < k} \mathbf{Var}(Y_i)$. Nun ist Y_i 0-1-wertig, also

$$\mathbf{Var}(Y_i) = \mathbf{Pr}(Y_i = 1) \cdot (1-\varrho)^2 + \mathbf{Pr}(Y_i = 0) \cdot (-\varrho)^2 = \varrho(1-\varrho),$$

und daher $\mathbf{Var}(Z_k) = k\varrho(1-\varrho)$. □

Wir wenden nun die Chebychev-Cantelli-Ungleichung (Proposition 2.7.2) an.[3] Dies liefert:

$$\begin{aligned} \mathbf{Pr}(Z_k = 0) &\le \mathbf{Pr}(Z_k \le \mathbf{E}(Z_k) - \mathbf{E}(Z_k)) \le \frac{\mathbf{Var}(Z_k)}{\mathbf{Var}(Z_k) + (\mathbf{E}(Z_k))^2} \qquad (4) \\ &= \frac{k\varrho(1-\varrho)}{k\varrho(1-\varrho) + (k\varrho)^2} = \frac{1-\varrho}{1-\varrho+k\varrho} < \frac{1}{1+k\varrho}. \end{aligned}$$

Für $k \ge \frac{1}{\varrho}$ liefert dies die Schranke $\mathbf{Pr}(Z_k = 0) < \frac{1}{2}$. Es ist auch bemerkenswert, dass selbst unter Aufwendung von beliebig vielen Zufallsbits wie in Algorithmus 4.1.1 $\frac{1}{\varrho}$ Tests nur zu einer konstanten Erfolgswahrscheinlichkeit führen: Ungleichung (1) liefert für $k = \frac{1}{\varrho}$:

$$\mathbf{Pr}(Z_k = 0) \le (1-\varrho)^{1/\varrho} < e^{-1} \approx 0{,}368\,;$$

dabei ist für kleine ϱ die Abschätzung mit 0,368 recht genau. Fazit also: Bei der Suche in der beschriebenen Form ist es bei kleinen Dichten ϱ praktisch unschädlich, zum Erreichen einer konstanten Erfolgswahrscheinlichkeit paarweise unabhängige Suche zu benutzen und dadurch viele Zufallsbits einzusparen.

Wir können (4) nun weiter benutzen, um die erwartete Rundenzahl nach oben abzuschätzen. Dabei verwenden wir unser Wissen, dass es (im modifizierten Algorithmus) mehr als p Runden auf keinen Fall geben kann. Wir wissen nach Fakt 2.2.7:

$$\mathbf{E}(\text{Rundenzahl}) = \sum_{k \ge 0} \mathbf{Pr}(\text{es gibt } \ge k+1 \text{ Runden}) = \sum_{0 \le k < p} \mathbf{Pr}(Z_k = 0).$$

[3] In der Originalarbeit von Chor und Goldreich wurde die Chebychev-Ungleichung benutzt. Diese führt zu einer ähnlichen Abschätzung, die aber für kleine k nicht „zieht“. Insbesondere ist die Abschätzung $\mathbf{Pr}(Z_k = 0) \le \frac{1}{2}$ mit der Chebychev-Ungleichung erst mit $k \ge 2/\varrho$ zu erreichen.

Mit (4) folgt:

$$\mathbf{E}(\text{Rundenzahl}) \leq \sum_{0 \leq k < p} \frac{1}{1 + k\varrho}. \tag{5}$$

Wir schätzen ab:

$$\frac{1}{1 + k\varrho} \leq \int_{k-1}^{k} \frac{dt}{1 + t\varrho} \text{ , für } k \geq 1.$$

Mit (5) folgt:

$$\mathbf{E}(\text{Rundenzahl}) \leq 1 + \int_{0}^{p-1} \frac{dt}{1 + t\varrho} < 1 + \int_{0}^{p} \frac{dt}{1 + t\varrho}.$$

Da $\int \frac{dt}{1+t\varrho} = \ln(1 + t\varrho)/\varrho$, folgt

$$\mathbf{E}(\text{Rundenzahl}) < 1 + [\ln(1 + t\varrho)/\varrho]_0^p = 1 + \frac{\ln(1 + p\varrho)}{\varrho} = 1 + \frac{1}{\varrho} \cdot \ln(|B| + 1).$$

(Zur allgemeinen Technik der Abschätzung von Summen durch Integrale siehe Abschnitt 4.5 unten.) Das Ergebnis sollte man mit dem Wert $1/\varrho$ aus (2) für die vollständig zufällige Suche vergleichen. Einerseits ist es einleuchtend, dass der Faktor $1/\varrho$ vorkommt; weniger schön ist der logarithmische Faktor $\ln(1 + p\varrho) = \ln(|B| + 1)$ im Zähler.

Wir können Algorithmus 4.1.3 so modifizieren, dass immer noch wenige Zufallsbits benötigt werden, dass aber eine erwartete Testzahl von $O(1/\varrho)$ erreicht wird, ohne Abhängigkeit von $|B|$. Die Idee ist dabei, zwei voneinander unabhängige Suchpunktfolgen zu generieren, die (quasi)parallel abgearbeitet werden; in jeder Runde werden also zwei Punkte aus A getestet. Hierfür wählen wir zwei Paare (r_1, s_1) und (r_2, s_2) von zufälligen Koeffizienten. Dies liefert Algorithmus 4.1.7.

Analyse: Korrektheit. Man sollte sich von der scheinbaren Endlosschleife nicht irritieren lassen: da auf jeden Fall ein Element von B gefunden wird, wird die Schleife über eine der **return**-Anweisungen verlassen.

Anzahl der Zufallsbits: Wir wählen vier Elemente von A zufällig; es werden also $4\lceil \log |A| \rceil$ Zufallsbits benötigt.

Laufzeit: Wir schätzen die erwartete Rundenzahl wie folgt ab. – Wir definieren:

$$X_i^{(1)} := r_1 \cdot i + s_1 \text{ und } X_i^{(2)} := r_2 \cdot i + s_2, \text{ für } i = 0, 1, \ldots, p - 1,$$

sowie

Algorithmus 4.1.7 (Verdoppelte paarweise unabhängige Suche)**.**
INPUT: Menge $A = \mathbb{Z}_p$ (Körper) mit unbekannter, nichtleerer Teilmenge B
METHODE:

```
1   Wähle r_1, r_2, s_1, s_2 aus Z_p zufällig;
2   Setze r_1 := max{1, r_1} und r_2 := max{1, r_2};
3   x1 := (s_1 − r_1) mod p;
4   x2 := (s_2 − r_2) mod p;
5   repeat
6       x1 := (x1 + r_1) mod p;
7       if x1 ∈ B then return x1;
8       x2 := (x2 + r_2) mod p;
9       if x2 ∈ B then return x2.
```

$$Y_i^{(1)} := \begin{cases} 1, & \text{falls } X_i^{(1)} \in B, \\ 0, & \text{sonst,} \end{cases} \qquad \text{und} \qquad Y_i^{(2)} := \begin{cases} 1, & \text{falls } X_i^{(2)} \in B, \\ 0, & \text{sonst,} \end{cases}$$

und schließlich, für $1 \le k \le p$:

$$Z_k^{(1)} := Y_0^{(1)} + \cdots + Y_{k-1}^{(1)} \quad \text{und} \quad Z_k^{(2)} := Y_0^{(2)} + \cdots + Y_{k-1}^{(2)}.$$

Die Analyse für die Folge Y_i, $0 \le i \le p-1$, und ihre Partialsummen Z_k von oben gilt natürlich für jede der beiden Folgen $Y_i^{(h)}$, $0 \le i \le p-1$, und $Z_k^{(h)}$ (mit $h = 1, 2$) gleichermaßen. Zudem sind $Z_k^{(1)}$ und $Z_k^{(2)}$ unabhängig. Wir benutzen gleich die Version (4), die auf der Chebychev-Cantelli-Ungleichung beruht, und erhalten:

$$\mathbf{Pr}(\text{Runden } 0, 1, \ldots, k-1 \text{ erfolglos}) = \mathbf{Pr}(Z_k^{(1)} = 0 \wedge Z_k^{(2)} = 0) \le \frac{1}{(1 + k\varrho)^2}\,.$$

Wie oben können wir nun den Erwartungswert der Rundenzahl abschätzen (zur Ab-

schätzung der Reihe durch das Integral vgl. wieder Abschnitt 4.5):

$$\begin{aligned}
\mathbf{E}(\text{Rundenzahl}) &= \sum_{k\geq 0} \mathbf{Pr}(\text{es gibt mindestens } k+1 \text{ Runden}) \\
&= \sum_{k\geq 0} \mathbf{Pr}(\text{die ersten } k \text{ Runden sind erfolglos}) \\
&\leq \sum_{k\geq 0} \frac{1}{(1+k\varrho)^2} \\
&= 1 + \sum_{k\geq 1} \frac{1}{(1+k\varrho)^2} \\
&< 1 + \int_0^\infty \frac{dt}{(1+t\varrho)^2} \\
&= 1 + \Big[-\frac{1/\varrho}{1+t\varrho}\Big]_0^\infty \\
&= 1 + \frac{1}{\varrho}.
\end{aligned}$$

Als Schranke für die erwartete Anzahl von Tests ergibt sich damit $2 + 2/\varrho$, was sich neben $1/\varrho$ für die völlig zufällige Suche sehr gut ausmacht.

4.2 Suche in sortierten einfach verketteten linearen Listen

In diesem Abschnitt untersuchen wir ein elementares Suchproblem: Gegeben ist eine *angeordnete* (einfach verkettete) lineare Liste mit Einträgen $x_1 < x_2 < \cdots < x_n$ aus einem geordneten Universum $(U, <)$. (Man stelle sich als Einträge Zahlen vor.) Wir wollen in dieser Liste nach einem Eintrag x suchen. Normalerweise bleibt nichts

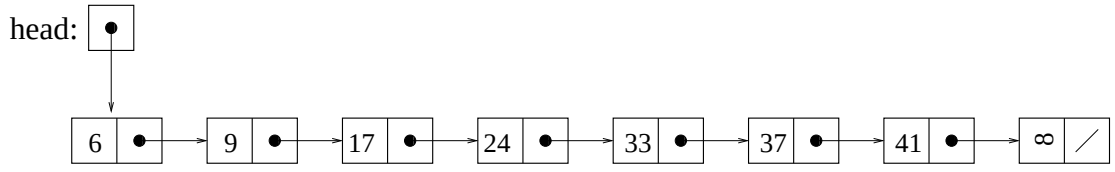


Abbildung 1: Geordnete lineare Liste

anderes übrig, als die Liste linear von vorne bis hinten zu durchmustern, bis das Element x gefunden wurde *oder* bis das Ende der Liste erreicht wird *oder* (wegen der Sortierung!) bis ein Eintrag $y > x$ gefunden wird. Der Aufwand für diese Suche ist proportional zur Anzahl der durchgeführten Schlüsselvergleiche. Diese beträgt k, wobei $k \leq n$ minimal ist mit $x_k \geq x$. (Die Möglichkeit, dass x größer ist als alle Listeneinträge, umgehen wir, indem wir ein letztes Listenelement mit einem künstlichen Schlüssel „∞“ anhängen. Ab hier sei also $x < x_n$ vorausgesetzt.) Im schlechtesten und im mittleren Fall ist der Suchaufwand $\Theta(n)$.

Diese lineare Schranke wollen wir schlagen. Hierfür muss eine besondere Voraussetzung erfüllt sein: Es muss möglich sein, ein Element der linearen Liste uniform zufällig auszuwählen. Dies ist bei den gewöhnlichen Listen-Strukturen, wie sie von den Programmiersprachen bereitgestellt werden, nicht der Fall. Allerdings ist es leicht, diese Voraussetzung herzustellen. Wir nennen drei Möglichkeiten.

- In Abbildung 2 ist die erste Möglichkeit dargestellt. Man könnte zur Implementierung der Liste vom Programm aus ein Array `A: array of listelem[1..`m`]` von Listenelementen kreieren und beim Einfügen in die Liste durch geeignete Verwaltung dafür sorgen, dass die benutzten Listenelemente immer einen vollen Anfangsabschnitt dieses Arrays bilden. Wenn auch Löschungen durchgeführt werden sollen, ist entweder eine doppelte Verzeigerung der Liste nötig oder man muss für eine Löschung zweimal suchen.

- Abbildung 3 zeigt eine zweite Möglichkeit. Eine lineare Liste lässt sich auch „zu Fuß“ mit Hilfe eines Arrays `value`, das die Daten enthält, und eines zweiten Arrays `next`, das die Zeiger als explizite Zahlen in $\{1, \ldots, n\}$ enthält, realisieren. Die Implementierung von Einfüge- und Lösch-Operationen ist eine Übungsaufgabe für die ersten Semester.

- Eine dritte Möglichkeit ist, ein Array `p[1..`n`]` zu halten, das einen Zeiger auf jedes Listenelement enthält. (Diese Lösung kostet natürlich zusätzlichen Platz.)

Wir stellen uns für das folgende vor, dass (Zeiger auf) Listenelemente zufällig gewählt werden können. Die Idee für den Algorithmus ist einfach: Man wählt zufällig eine Reihe (L viele) von Listenelementen $x_{i_1}, \ldots, x_{i_L}$ (Wiederholungen spielen dabei keine Rolle) und sucht unter diesen den größten Eintrag x_{i_0} heraus, der $< x$ ist. Von diesem Listenelement startend sucht man x oder das erste Element $> x$ in der Liste. Leichte Komplikationen werden nur dadurch verursacht, dass x kleiner als alle Einträge in der Liste sein könnte.

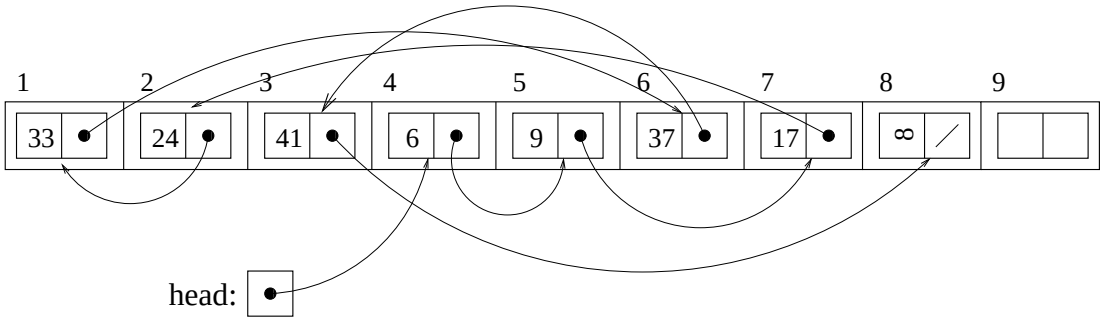


Abbildung 2: Geordnete lineare Liste, Listenelemente in Array

Algorithmus 4.2.1 (Randomisierte Suche in linearer Liste)**.**

INPUT: Liste (Start bei `head`) mit Einträgen $x_1 < \cdots < x_n$, Suchschlüssel $x < x_n$.

METHODE:

```
1   // Fall x ≤ x_1 abfangen:
2   if x ≤ head.key then return head;
3   // Phase 1: Suche zufällig Schlüssel < x
                  – der kleinste Schlüssel ist auf jeden Fall geeignet – :
4   best ← head; bestkey ← head.key;
5   repeat L times
6     temp ← (Zeiger auf) zufälliges Listenelement;
7     if temp.key < x and bestkey < temp.key
8     then best ← temp; bestkey ← temp.key;
9   // Phase 2: Suche ersten Schlüssel ≥ x:
10  temp ← best;
11  repeat temp ← temp.next until temp.key ≥ x;
12  // endet wegen x_n = ∞
13  return temp.
```

Analyse: Korrektheit. Nach unserer Annahme über ein sehr großes Element „$x_n = \infty$“ am Ende ist die Liste garantiert nicht leer. Zeile 2 fängt den Fall ab, dass $x \leq x_1$ ist (kleinster Listeneintrag). Ab Zeile 4 ist also sicher, dass $x_1 < x < x_n$ ist, und dass `temp.key` $< x$. Die Schleife in Zeilen 5 bis 7 ändert nichts an der Eigenschaft `temp.key` $< x$. In Zeile 10 wird das erste Listenelement gesucht und gefunden, das einen Schlüssel $\geq x$ enthält. Ein solches Listenelement muss es geben, da $x_n > x$ ist.

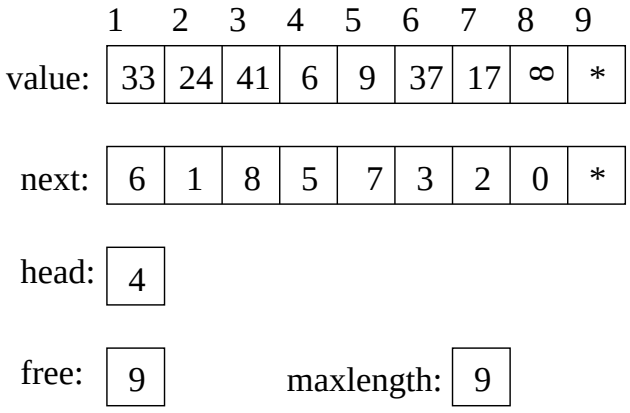


Abbildung 3: Geordnete lineare Liste in 2 Arrays

Analyse: Zeitbedarf. Wir können annehmen, dass $x > x_1$ ist. Inklusive der vorbereitenden Zeile 2 (1 Schlüsselvergleich) benötigt Phase 1 höchstens $1 + 2L$ Schlüsselvergleiche. Die Anzahl der Schlüsselvergleiche in Phase 2 ist eine Zufallsvariable. Wenn x_k der kleinste Schlüssel ist, der $\geq x$ ist, dann wird auf jeden Fall x mit x_k verglichen. Die Anzahl der weiteren Vergleiche nennen wir Y_2. Wir haben (siehe Abbildung 4):

$Y_2 \geq j$
$\Leftrightarrow$ in Phase 2 werden $x_{k-j}, \ldots, x_{k-1}$ mit x verglichen
$\Leftrightarrow$ in Phase 1 werden $x_{k-j}, \ldots, x_{k-1}$ nicht gewählt.

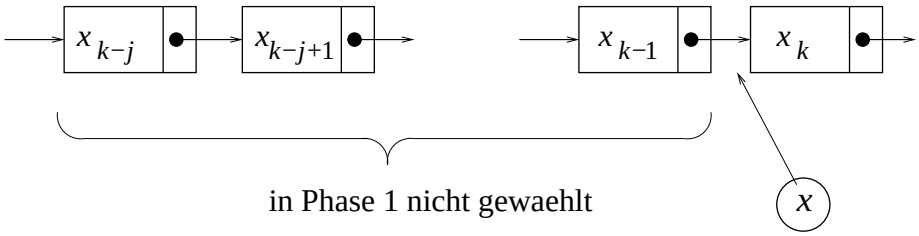


Abbildung 4: Listeneinträge, die nicht gewählt werden dürfen, wenn $Y_2 \geq j$ sein soll

Die Wahrscheinlichkeit, dass $x_{k-j}, \ldots, x_{k-1}$ in keinem der L Versuche in Phase 1

gewählt werden, ist

$$\left(\frac{n-j}{n}\right)^L < e^{-jL/n}.$$

Damit:

$$\mathbf{E}(Y_2) = \sum_{j\geq 1} \mathbf{Pr}(Y_2 \geq j) \leq \sum_{j\geq 1} e^{-jL/n} = \frac{e^{-L/n}}{1-e^{-L/n}} = \frac{1}{e^{L/n}-1} < \frac{1}{(1+\frac{L}{n})-1} = \frac{n}{L}.$$

Wenn Y die Anzahl aller Vergleiche im Algorithmus bezeichnet, erhalten wir:

$$\mathbf{E}(Y) \leq 1 + 2L + 1 + \frac{L}{n} = 2 + 2L + \frac{n}{L}.$$

Wir setzen nun L so fest, dass dieser Ausdruck minimal wird. Standardmethoden (setze Ableitung von $2x + n/x$ gleich 0, was $x = \sqrt{n/2}$ liefert) ergeben als fast optimale Wahl $L = \lceil\sqrt{n/2}\rceil$. Damit wird

$$\mathbf{E}(Y) \leq 2 + 2\left\lceil\sqrt{n/2}\right\rceil + \frac{n}{\left\lceil\sqrt{n/2}\right\rceil} \leq 4 + 2\sqrt{2}\cdot\sqrt{n}.$$

Proposition 4.2.2. *Suchen in einer angeordneten linearen Liste mit n Einträgen, bei der zufälliges Wählen eines Listenelements möglich ist, benötigt bei Verwendung von Algorithmus 4.2.1 im erwarteten Fall Zeit $O(\sqrt{n})$.*

Bemerkung: Wenn man anstelle der in einem Array organisierten Liste direkt ein sortiertes Array benutzt, kann man mit $\log n$ Vergleichen suchen (binäre Suche).

In unserer Listenstruktur sind jedoch auch **Einfügungen** (und Löschungen) in erwarteter Zeit $O(\sqrt{n})$ durchführbar, was in einem sortierten Array nicht der Fall ist.

4.3 Quickselect

Für diesen Abschnitt sei U eine durch die Relation $<$ totalgeordnete Menge (z.B. $U = \mathbb{N}$).

Definition 4.3.1. *Wenn $S = (a_1, \ldots, a_n)$ mit $a_1 \leq \cdots \leq a_n$ eine Liste mit n Einträgen aus U ist, und $1 \leq k \leq n$, dann heißt a_k* **das Element von Rang k** *in S. Das Element a von Rang $\lceil n/2\rceil$ heißt der* ***Median*** *von S.*

Beispiel: In $S = (1, 4, 7, 11, 15, 27, 30)$ ist 4 das Element vom Rang 2 und 11 ist der Median. In $S = (1, 4, 7, 15, 27, 30)$ ist 30 das Element vom Rang 6, und 7 ist der Median.

Beachte: Wenn n ungerade ist, ist der Median das „mittlere“ Element, wenn man S sortiert anordnet; wenn n gerade ist, sitzt der Median unmittelbar links neben der Mitte.

Gegeben ist nun ein Array `A[1..n]`, in dem die verschiedenen Einträge $a_1 < \cdots < a_n$ aus einem totalgeordneten Bereich $(U, <)$ in **irgendeiner Reihenfolge**, also ungeordnet, gespeichert sind.

Aufgabe: „Selection(k)“: Arrangiere die Einträge in `A[1..n]` so um, dass in `A[`k`]` das Element a_k von Rang k in $S = \{a_1, \ldots, a_n\}$ steht und dass die Einträge in `A[1..`$k-1$`]` kleiner als a_k und die in `A[`$k+1$`..n]` größer als a_k sind.

Der „Quickselect-Algorithmus“ ist ein randomisierter Algorithmus, der das Selektionsproblem in erwarteter Zeit $O(n)$ löst.

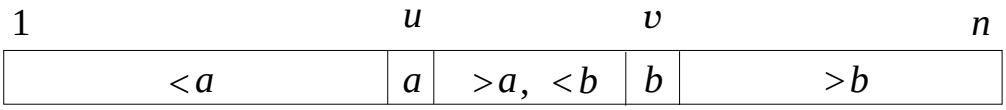
Algorithmus und Analyseprinzip finden sich in der Vorlesung „Algorithmen und Datenstrukturen“ vom SS 2016, Kapitel 8, auf den Druckfolien 56–68, siehe
`http://www.tu-ilmenau.de/iti/lehre/lehre-ss-2016/aud/`.

4.4 Mediansuche mit „Random Sampling“

In diesem Abschnitt betrachten wir nochmals das Selektionsproblem, und zwar den wichtigen Spezialfall $k = \lceil n/2 \rceil$: das ***Medianproblem***.

Zur Vereinfachung der Analyse nehmen wir an, dass $n/2$ und $n^{1/4}$ ganze Zahlen sind. (Andernfalls arbeitet man mit geeigneten gerundeten Werten.)

Die Idee ist, mit einem Zufallsverfahren zwei Elemente a und b in S zu finden derart, dass mit großer Wahrscheinlichkeit a kleiner als der Median und b größer als der Median ist. Dann kann man durch einmaliges Durchmustern aller Einträge in `A` und Vergleichen mit a und b die Einträge von `A` in drei Klassen einteilen und umarrangieren: In `A[1..`$u-1$`]` stehen die Einträge, die kleiner als a sind; in `A[`u`]` steht a; in `A[`$u+1$`..`$v-1$`]` stehen die Einträge, die zwischen a und b liegen; in `A[`v`]` steht b; in `A[`$v+1$`..n]` schließlich stehen die Einträge, die größer als b sind. Nun muss man nur noch die Einträge in `A[`$u+1$`..`$v-1$`]` sortieren, um den Median an seinen Platz zu schaffen und die anderen Elemente wie verlangt anzuordnen.



Wenn nur $v - u = o(n/\log n)$, wird der Aufwand für das Sortieren $o(n)$ sein.

Aber wie finden wir die gewünschten Einträge a und b? Die Intuition hinter dem Algorithmus ist folgende. Wir wählen

$$L = n^{3/4}$$

viele Einträge aus S rein zufällig, der – algorithmischen – Einfachheit halber mit Wiederholung. Das ist im Durchschnitt jeder $n^{1/4}$-te Eintrag. (Diese Zufallsauswahl nennt man „random sampling“.) Man kann sich vorstellen, dass diese zufällig gewählten Elemente $b_1 \leq \ldots \leq b_L$ einigermaßen gleichmäßig über den Bereich $\{1, \ldots, n\}$ der Ränge in S verstreut sind. Wir wählen nun aus der (Multi-)Menge $B = \{b_1, \ldots, b_L\}$ zwei Elemente a und b, die wir in der Nähe der Ränge $n/2 - n^{3/4}$ bzw. $n/2 + n^{3/4}$ vermuten. Da B um den Faktor $n^{1/4}$ „dünner“ als S ist, sollten wir hierfür die Elemente aus B vom Rang $(n/2 - n^{3/4})/n^{1/4} = \frac{1}{2}n^{3/4} - \sqrt{n}$ und $(n/2 + n^{3/4})/n^{1/4} = \frac{1}{2}n^{3/4} + \sqrt{n}$ nehmen. Um diese Elemente zu identifizieren, wird B einfach sortiert. Der Aufwand hierfür ist $O(n^{3/4} \log n) = o(n)$.

Es kann natürlich passieren, dass bei der Zufallsauswahl etwas schief geht. Es kann sein, dass a und b den Median nicht „einrahmen“ oder dass der Abstand $v - u$ zu groß ist. Mit Hilfe der Hoeffding-Schranke werden wir aber zeigen, dass dies nur mit verschwindend geringer Wahrscheinlichkeit passiert. Zudem kann diese Situation erkannt werden, so dass wir einen selbstverifizierenden Algorithmus mit sehr kleiner Versagenswahrscheinlichkeit erhalten (Abb. 5).

Es ist klar, dass der Algorithmus, wenn er den Fall 3 erreicht und den mittleren Abschnitt sortiert hat, die verlangte Umordnung „Median in die mittlere Position“ korrekt vorgenommen hat. Wir haben also einen selbstverifizierenden Algorithmus, den man durch Wiederholung in einen Las-Vegas-Algorithmus umbauen kann, wenn die Wahrscheinlichkeit für die Ausgabe „?“ genügend klein ist.

Laufzeitanalyse: Entscheidend ist die Zahl der Vergleiche. Für das Sortieren der $n^{3/4}$ Zufallselemente genügen $O(n^{3/4} \log n) = o(n)$ Vergleiche. Für die Separierung der drei Abteilungen „ $< a$“, „in $[a, b]$“ und „$> b$“ werden im schlechtesten Fall $2n$ Vergleiche benötigt. Wenn man für jedes x zufällig entscheidet, ob man x zuerst mit

Algorithmus 4.4.1 (Random Sample Median).
Input: Array `A[1..n]` (ungeordnet)
Methode:
(1) $L := n^{3/4}$;
(2) Wähle **zufällig** $j_1, \ldots, j_L$ aus $\{1, \ldots, n\}$;
(3) Sortiere $\texttt{A}[j_1], \ldots, \texttt{A}[j_L]$ aufsteigend;
Resultat: $b_1 \leq b_2 \leq \cdots \leq b_L$.
(4) $a := b_{L/2-\sqrt{n}}$;
(5) $b := b_{L/2+\sqrt{n}}$;
(6) transportiere die Einträge $< a$ nach $\texttt{A}[1..u-1]$;
(7) transportiere die Einträge $> b$ nach $\texttt{A}[v+1..n]$;
(8) transportiere a nach $\texttt{A}[u]$, b nach $\texttt{A}[v]$;
(9) transportiere die anderen Einträge in $[a, b]$ nach $\texttt{A}[u+1..v-1]$;
(10) **Fall 1:** $u > n/2$ oder $v < n/2$: **return** „?“;
(11) **Fall 2:** $v - u > 2D \cdot n^{3/4}$: **return** „?“;
(12) // D ist eine noch festzulegende Konstante
(13) **Fall 3:** Sonst. Sortiere $\texttt{A}[u+1..v-1]$, z.B. mit Mergesort.

Abbildung 5: Algorithmus Random Sample Median

a oder mit b vergleicht, ist mit hoher Wahrscheinlichkeit die Zahl von Vergleichen für Zeilen (6)–(9) sogar nur etwa $\frac{3}{2}n + o(n)$ (Übungsaufgabe!). Das Sortieren des mittleren Teilarrays in Fall 3 kostet wieder $O(n^{3/4} \log n) = o(n)$ Vergleiche.

Wahrscheinlichkeitsanalyse: Wir schätzen die Wahrscheinlichkeit dafür ab, dass nicht Fall 3 eintritt.

Mit welcher Wahrscheinlichkeit tritt **Fall 1** ein, d. h. ist $u > n/2$ oder $v < n/2$? Wir fragen also nach der Wahrscheinlichkeit $\mathbf{Pr}(A_1 \cup A_2)$ für die Ereignisse

$$A_1 := \{u > n/2\} \quad \text{und} \quad A_2 := \{v < n/2\}.$$

Wir betrachten nur A_1. Dazu erinnern wir uns, dass $a_1 < \cdots < a_n$ gilt. (Diese Anordnung wird vom Algorithmus natürlich nicht hergestellt.) Da über $j_1, \ldots, j_L$ Einträge zufällig gewählt werden, ist die Anordnung der Einträge in `A` aber unerheblich. Wenn A_1 eintritt, dann trifft die Zufallsauswahl $\texttt{A}[j_1], \ldots, \texttt{A}[j_L]$ die Menge $\{a_1, \ldots, a_{n/2}\}$ (erste Hälfte) strikt weniger als $(L/2 - \sqrt{n})$-mal. Andererseits erwarten wir genau $L/2$ Treffer. Diese Unterschreitung des Erwarungswertes ist sehr unwahrscheinlich,

wie man durch Anwendung der Hoeffding-Schranke sieht. Wir definieren:

$$\begin{aligned} X_i &:= \left\{ \begin{array}{ll} 1, & \text{falls } \mathtt{A}[j_i] \in \{a_1, \ldots, a_{n/2}\}, \\ 0, & \text{sonst,} \end{array} \right\}, \text{ für } 1 \leq i \leq L; \\ X &:= X_1 + \cdots + X_L, \\ m &:= \mathbf{E}(X) = L/2, \\ \delta &:= \sqrt{n}/m = 2n^{-1/4}. \end{aligned}$$

Dann haben wir, nach einer Form der Hoeffding-Schranke (Korollar 2.6.4, Formel (15) in Kapitel 2):

$$\begin{aligned} \mathbf{Pr}(A_1) &= \mathbf{Pr}(X < (1-\delta)m) \\ &\leq e^{-\delta^2 m/2} \\ &= e^{-n^{-1/2} \cdot L} \\ &= e^{-n^{1/4}}. \end{aligned}$$

Da man $\mathbf{Pr}(A_2)$ analog abschätzen kann, ergibt sich als Schranke für die Wahrscheinlichkeit, dass Fall 1 eintritt, $O(e^{-n^{1/4}})$, eine mit wachsendem n sehr rasch verschwindende Wahrscheinlichkeit.

Mit welcher Wahrscheinlichkeit tritt **Fall 2** ein, d.h. es ist $v - u > 2Dn^{3/4}$? B sei das Ereignis, dass Fall 2 eintritt, *nicht aber* Fall 1.

Wenn Fall 2 eintritt, aber nicht Fall 1, dann ist $u \leq n/2 \leq v$ und es tritt mindestens einer der beiden folgenden Fälle ein:
(i) $v - n/2 > D \cdot n^{3/4}$ (Ereignis B_1),
(ii) $n/2 - u > D \cdot n^{3/4}$ (Ereignis B_2).

(Wenn $v - n/2 \leq D \cdot n^{3/4}$ und $n/2 - u \leq D \cdot n^{3/4}$ wäre, würde $v - u \leq 2D \cdot n^{3/4}$ folgen.)

Wir zeigen, dass B_1 exponentiell kleine Wahrscheinlichkeit hat; für B_2 zeigt man dies analog.

Dass $v > n/2 + D \cdot n^{3/4}$ ist, bedeutet, dass $b = b_{L/2+\sqrt{n}}$ nicht in $\{a_1, \ldots, a_{n/2+D \cdot n^{3/4}}\}$ sitzt, d.h. dass die Zufallsauswahl $\mathtt{A}[j_1], \ldots, \mathtt{A}[j_L]$ die Menge $S_{\text{unten}} := \{a_1, \ldots, a_{n/2+D \cdot n^{3/4}}\}$ weniger als $L/2 + \sqrt{n}$ oft trifft. In S_{unten} erwarten wir aber $(n/2 + Dn^{3/4}) \cdot (L/n) = L/2 + D\sqrt{n}$ Treffer. Wir rechnen nun wieder mit Hilfe der Hoeffding-Ungleichung aus, dass eine Trefferzahl von $L/2 + \sqrt{n}$ für nicht zu kleine D extrem unwahrscheinlich ist.

Definiere:

$$\begin{aligned} X_i &:= \left\{ \begin{array}{ll} 1, & \text{falls } \texttt{A}[j_i] \in S_{\text{unten}}, \\ 0, & \text{sonst,} \end{array} \right\}, \text{ für } 1 \le i \le L; \\ X &:= X_1 + \cdots + X_L, \\ m &:= \mathbf{E}(X) = L \cdot \frac{|S_{\text{unten}}|}{n} = \frac{L}{2} + D\sqrt{n}, \\ \delta &:= \frac{m - (L/2 + \sqrt{n})}{m} = \frac{(D-1)\sqrt{n}}{L/2 + D\sqrt{n}}. \end{aligned}$$

Nun haben wir, wieder mit der Hoeffding-Schranke (Korollar 2.6.4 in Kap. 2):

$$\begin{aligned} \mathbf{Pr}(B_1) &\le \mathbf{Pr}(X \le m(1-\delta)) \\ &\le e^{-\delta^2 m/2} \\ &= e^{-(D-1)^2 \cdot n/(L+2D\sqrt{n})} \\ &< e^{-(D-1)^2 \cdot n^{1/4}/(1+2Dn^{-1/4})}. \end{aligned}$$

Für jedes feste $D > 1$ geht auch diese Wahrscheinlichkeit mit wachsendem n exponentiell rasch gegen 0. Man wählt z.B. $D = \frac{5}{2}$ und hat damit einen zu sortierenden „mittleren Bereich“ von maximal $5n^{3/4}$ Elementen und eine Wahrscheinlichkeit von höchstens $2e^{-2.25 \cdot n^{1/4}/(1+5n^{-1/4})} < e^{-n^{1/4}}$ für Fall 2 (für n genügend groß).

Wir fassen zusammen:

Satz 4.4.2. *Algorithmus 4.4.1 ist ein selbstverifizierender Algorithmus für das Medianproblem (einschließlich Umarrangieren). Die Anzahl der durchzuführenden Vergleiche ist $\frac{3}{2}n + O(n^{3/4} \log n)$, die Versagenswahrscheinlichkeit ist $O(e^{-n^{1/4}})$.*

Bemerkung 1: Beim Umbau zu einem Las-Vegas-Algorithmus gemäß Kapitel 3 ergibt sich eine *erwartete* Vergleichsanzahl von $\frac{3}{2}n + O(n^{3/4} \log n)$, da auch mit den zusätzlichen Faktoren $1/(1 - O(e^{-n^{1/4}})) = 1 + O(e^{-n^{1/4}})$ die Vergleichsanzahl in $\frac{3}{2}n + O(n^{3/4} \log n)$ bleibt.

Bemerkung 2: Unter den bekannten Median-Algorithmen gibt es keinen, der die Vergleichsanzahl des Random-Sampling-Algorithmus (im wesentlichen $\frac{3}{2}n$) schlägt.

Bemerkung 3: Eine Variante der Random-Sampling-Methode führt zu effizienten parallelen Sortieralgorithmen.

4.5 Ergänzung: Abschätzung von Summen durch Integrale

Im Abschnitt über zweifach unabhängige Suche (4.1) haben wir zweimal eine unendliche Reihe durch ein Integral abgeschätzt. Dahinter steckt eine einfache Technik, die es erlaubt, Summen mit monoton fallenden oder monoton wachsenden Summanden einfach und recht genau abzuschätzen. Aus den Ungleichungen lässt sich auch ablesen, wie groß der Schätzfehler maximal ist.

Lemma 4.5.1. *Sei* $f\colon [a,b] \to \mathbb{R}_0^+$ *eine Funktion, wobei* $a \in \mathbb{N}$ *und* $b \in \mathbb{N} \cup \{\infty\}$. *Dann gilt*:

(a) *Wenn* f *monoton wachsend ist, und* $a < b < \infty$:

$$\int_a^b f(x)\,dx + f(a) - f(b) \le \sum_{a \le i < b} f(i) \le \int_a^b f(x)\,dx.$$

(b) *Wenn* f *monoton fallend ist, und* $a < b < \infty$:

$$\int_a^b f(x)\,dx \le \sum_{a \le i < b} f(i) \le \int_a^{b-1} f(x)\,dx + f(a) \le \int_a^b f(x)\,dx + f(a).$$

(c) *Wenn* f *monoton fallend ist, gilt für die unendliche Reihe*:

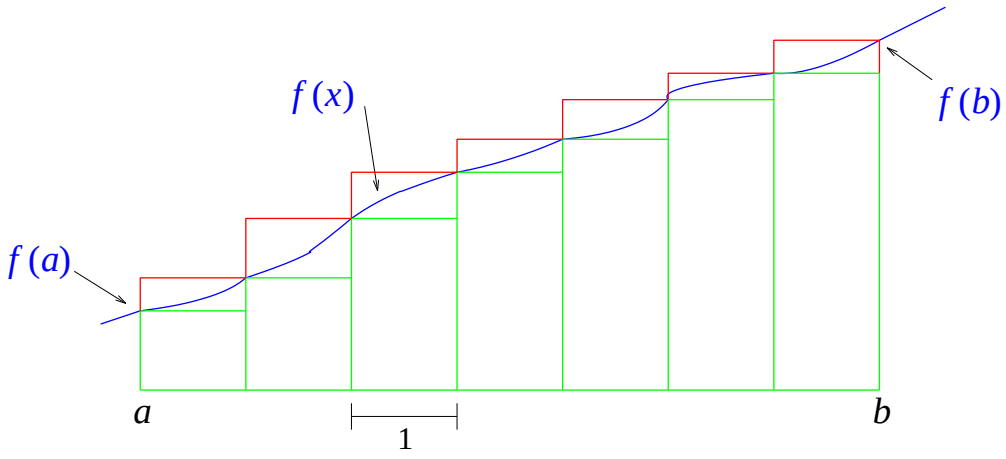
$$\int_a^\infty f(x)\,dx \le \sum_{i \ge a} f(i) \le \int_a^\infty f(x)\,dx + f(a).$$

Beweis: (a) Die Situation ist wie in der folgenden Skizze. Die roten Rechtecke deuten eine Obersumme für das Integral $\int_a^b f(x)\,dx$ an, die grünen Rechtecke eine Untersumme.

Weil f monoton wachsend ist, gilt:

$$\int_{i-1}^{i} f(x)\,dx \le f(i), \text{ für } a < i \le b, \text{ und } f(i) \le \int_i^{i+1} f(x)\,dx, \text{ für } a \le i < b. \qquad (6)$$

Wir summieren die erste Ungleichung über $a < i \le b$ und erhalten $\int_a^b f(x)\,dx \le \sum_{a<i\le b} f(i)$, also $\int_a^b f(x)\,dx - f(b) + f(a) \le \sum_{a\le i<b} f(i)$. Die zweite Ungleichung summieren wir über $a \le i < b$ und erhalten $\sum_{a\le i<b} f(i) \le \int_a^b f(x)\,dx$.



(b) Weil f monoton fallend ist, gilt:

$$\int_i^{i+1} f(x)\,dx \le f(i), \text{ für } a \le i < b, \text{ und } f(i) \le \int_{i-1}^{i} f(x)\,dx, \text{ für } a < i \le b. \qquad (7)$$

Wir summieren die erste Ungleichung in (7) über $a \le i < b$ und erhalten $\int_a^b f(x)\,dx \le \sum_{a\le i<b} f(i)$. Die zweite Ungleichung in (7) summieren wir über $a < i < b$ und erhalten $\sum_{a<i<b} f(i) \le \int_a^{b-1} f(x)\,dx \le \int_a^b f(x)\,dx$.

(c) Wenn $b = \infty$, summieren wir die erste Ungleichung in (7) über alle $i \ge a$ und die zweite über alle $i > a$, um die behaupteten Ungleichungen zu erhalten. □

Beispiele:
(i) Mit $f(x) = 1/x$ und $a = 1$ und $b = n + 1$ erhalten wir mit Lemma 4.5.1(b) obere und untere Schranken für $H_n = \sum_{1\le i\le n} \frac{1}{i}$:

$$\boxed{\ln n < \ln(n+1) = \int_1^{n+1} \frac{dx}{x} \le H_n \le 1 + \ln n.}$$

(ii) Mit $f(x) = 1/x^2$ und $b = \infty$ erhalten wir mit Lemma 4.5.1(c):

$$\int_a^{\infty} f(x)\,dx = \left[-1/x\right]_a^{\infty} = \frac{1}{a} \le \sum_{i\ge a} \frac{1}{i^2} \le \int_a^{\infty} f(x)\,dx + f(a) = \frac{1}{a} + \frac{1}{a^2}.$$

(iii) Mit $f(x) = \ln(x)$ und $a = 1$ und $b = n$ können wir Lemma 4.5.1(a) anwenden. Beachte, dass $\int \ln x \, dx = x(\ln x - 1) + C$, also $\int_1^n \ln(x)\, dx = n(\ln n - 1) + 1$. Wir erhalten:

$$n(\ln n - 1) + 1 - \ln n \leq \sum_{1 \leq i < n} \ln i \text{ , d.h. } n(\ln n - 1) + 1 \leq \sum_{1 \leq i \leq n} \ln i = \ln(n!).$$

Daraus schließen wir: $\boxed{n! \geq e^{n(\ln n - 1)+1} = e \cdot \left(\frac{n}{e}\right)^n.}$ Weiter ergibt sich (zweite Ungleichung in Lemma 4.5.1(a)):

$$\sum_{1 \leq i \leq n} \ln i \leq n(\ln n - 1) + 1 + \ln n \text{ , d.h. } \ln(n!) \leq n(\ln n - 1) + 1 + \ln n.$$

Daraus schließen wir: $\boxed{n! \leq e^{n(\ln n - 1)+1+\ln n} = (en) \cdot \left(\frac{n}{e}\right)^n.}$ Dies sind schon (für den geringen Aufwand) recht ordentliche Abschätzungen für $n!$.