4 Randomisierte Suche

Mit diesem Kapitel wenden wir uns dem Hauptteil der Vorlesung zu: Beispielalgorithmen und Analysetechniken. Hier geht es um ein Grundproblem: Das Suchen von Objekten anhand bestimmter Kriterien. Die Suchprobleme sind daher ganz unterschiedlich, und ebenso unterschiedlich sind die algorithmischen Ansätze.

Abschnitt 4.1 betrachtet ein sehr einfach aussehendes Problem: Gegeben ist eine endliche Grundmenge A mit einer Teilmenge B, die aber nur über Tests „ist x in B?“ zugänglich ist. Man soll ein Element von B finden. Der offensichtliche randomisierte Algorithmus, nämlich so lange zufällige Elemente von A zu wählen, bis man auf ein Element von B stößt, benötigt möglicherweise sehr viele Zufallszahlen. Wir finden einen Algorithmus, der fast ebenso schnell ist, aber im Wesentlichen nur konstant viele Elemente von A zufällig wählt.

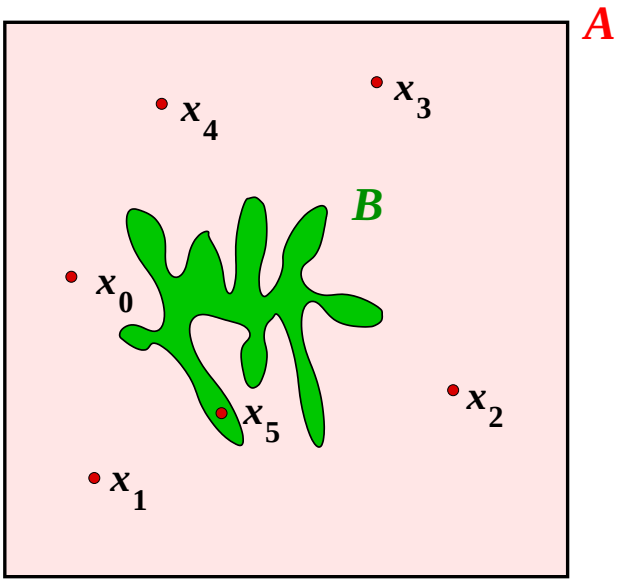
In Abschnitt 4.2 geht es um ein klassisches Problem, nämlich die Implementierung von Wörterbüchern mit linearen Listen. Normalerweise ist die Zeit für eine Suche, eine Einfügung, und eine Löschung $\Theta(n)$, wenn n Elemente gespeichert sind. Wir werden sehen, dass man in einer angeordneten Liste viel schneller suchen kann, wenn man Listenelemente zufällig auswählen kann.

In Abschnitt 4.3 betrachten wir den Algorithmus „Quickselect“, der schon aus der Vorlesung „Algorithmen und Datenstrukturen“ bekannt sein sollte. Wir betrachten hier eine alternative Analyse, die sich an der Analyse von Quicksort aus dem ersten Kapitel anlehnt. (Eine gute Gelegenheit, die Quicksort-Analyse nochmals anzusehen!)

In Abschnitt 4.4 schließlich wird ein anderer Algorithmus zur Ermittlung des Medians einer Folge präsentiert. Dieser verallgemeinert die Idee, ein Pivot zufällig zu wählen, darauf, mehrere zufällige Elemente zu wählen und sie zur sehr effizienten Reduzierung der Problemgröße zu benutzen. Das verbleibende Problem kann dann unter Einhaltung der linearen Zeitschranke einfach durch Sortieren gelöst werden. Das Ergebnis ist ein Algorithmus mit der bestmöglichen Vergleichsanzahl für das Medianproblem.

4.1 Suche mit wenigen Zufallsbits

Wir stellen uns folgendes Grundproblem vor: Gegeben ist eine endliche Menge A der Größe n sowie eine nichtleere Teilmenge $B \subseteq A$.



Dabei haben wir auf die Menge B keinen direkten Zugriff, wie etwa durch eine Auflistung, sondern wir haben nur die folgenden Operationen zur Verfügung:

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

Die Aufgabe ist, ein Element von B zu finden. Die *Kosten* hierfür ist die Anzahl der durchgeführten Tests „ist $x \in B$?“.

Algorithmus 4.1.1 ***Zufällige Suche***

INPUT: Endliche Menge A mit unbekannter nichtleerer Teilmenge B
METHODE:

```
1   repeat
2     x ← zufälliges Element von A
3   until x ∈ B;
4   return x.
```

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

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

Wie in Kapitel 5 im Detail besprochen wird, gibt es einen sehr effizienten (randomisierten) *Primzahltest*, der es erlaubt, Zahlen $x \in A$ darauf zu testen, ob sie in B sind. Ein Verfahren, effizient (d. h., in Rechenzeit $O(\ell^c)$ für eine Konstante c) eine Primzahl in A zu konstruieren, kennt man dagegen nicht.

Wenn wir die Elemente von A aufzählen und nacheinander testen, 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. Sogar in der angenehmen Situation, in der etwa jedes zweite Element von A zu B gehört, 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 (Algorithmus 4.1.1). Man wählt wiederholt ein Element x aus A zufällig; wenn und sobald dieses x in B liegt, stoppt man und gibt x aus. Dieses Verfahren würde in der Situation $|B| \geq \frac{1}{2}|A|$ im Schnitt nur höchstens zwei Versuche brauchen. Wir erkennen aber auch, 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 R die Anzahl der Schleifendurchläufe. Die Zufallsvariable R ist offensichtlich geometrisch verteilt mit Parameter ϱ

(siehe Abschnitt 2.5.1). Wir haben also:

$$\mathbf{E}(R) = \frac{1}{\varrho}. \tag{4.1.1}$$

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 (eine Zahl in $\{0, 1, \ldots, |A| - 1\}$) 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 erzeugten Zufallsbits ist $\lceil \log |A| \rceil / \varrho$.

Wenn die Dichte ϱ klein ist, kann die Zahl der Zufallsbits erheblich sein. Beispielsweise ist im Fall der Suche nach einer ℓ-ziffrigen Primzahl die Dichte $\Theta(1/\ell)$ und damit die erwartete Anzahl von Zufallsbits $\Theta(\ell^2)$.

Wir überlegen im Folgenden, wie wir mit relativ wenigen Zufallsbits fast ebenso effizient ein x in B finden 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.[1] Die Idee ist, zwei Elemente r und s in $\mathbb{Z}_p$ zufällig zu wählen und dann die Folge

$$X_0 = r \cdot 0 + s, X_1 = r \cdot 1 + s, X_2 = r \cdot 2 + s, X_3 = r \cdot 3 + s, \ldots \tag{4.1.2}$$

[1] 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.

(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.

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 $r \leftarrow 1$; // erzwingt kompletten Durchlauf durch A
3 $\mathtt{x} \leftarrow (s - r) \bmod p$;
4 **repeat**
5 $\quad \mathtt{x} \leftarrow (\mathtt{x} + r) \bmod p$;
6 **until** $\mathtt{x} \in B$;
7 **return** x.

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 trifft. (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 zunächst vor, dass der Wert $r = 0$ einfach beibehalten wird (damit die Mathematik glatter funktioniert). Wenn dieser (mathematisch) ideale Algorithmus $r = 0$ und $s \notin B$ wählt, findet er niemals ein Element von B, wohingegen Algorithmus 4.1.3 nach höchstens p Versuchen erfolgreich ist. Wenn der ideale Algorithmus $r = 0$ und $s \in B$ wählt, finden beide Algorithmen in der ersten Runde ein $x \in B$.

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

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

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

$$\mathbf{Pr}(X_i = a \wedge X_j = b) = \mathbf{Pr}(r \cdot i + s = a \wedge r \cdot j + s = b) = \frac{\left|\{(r,s) \mid \binom{i\ 1}{j\ 1} \cdot \binom{r}{s} = \binom{a}{b}\}\right|}{p^2}.$$

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. Daher wird jeder Wert $\binom{a}{b}$ mit derselben Wahrscheinlichkeit p^{-2} angenommen, und wir erhalten:

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

was die Unabhängigkeit beweist. □

Korollar 4.1.5
Die Zufallsvariablen Y_i, $0 \leq i \leq p-1$, haben Erwartungswert $\mathbf{E}(Y_i) = \varrho$ 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 nach Fakt 2.5.5 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 2.5.7(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 $\mathbf{Var}(Z_k) = k\varrho(1-\varrho)$. □

Anwenden der Chebychev-Cantelli-Ungleichung[2] (Prop. 2.7.2) liefert:

$$\begin{aligned} \mathbf{Pr}(Z_k = 0) = \mathbf{Pr}(Z_k \le \mathbf{E}(Z_k) - \mathbf{E}(Z_k)) &\le \frac{\mathbf{Var}(Z_k)}{\mathbf{Var}(Z_k) + \mathbf{E}(Z_k)^2} \\ = \frac{k\varrho(1-\varrho)}{k\varrho(1-\varrho) + (k\varrho)^2} &= \frac{1-\varrho}{1-\varrho+k\varrho} \le \frac{1}{1+k\varrho}. \end{aligned} \tag{4.1.3}$$

Wir betrachten nun wieder den (realen) Algorithmus 4.1.3, mit $r \neq 0$, und bezeichnen mit R die Rundenzahl in diesem Algorithmus.

Lemma 4.1.7
Für Algorithmus 4.1.3 gilt:

$$\mathbf{Pr}(R \ge k+1) \le \frac{1-\varrho}{1-\varrho+k\varrho} \le \frac{1}{1+k\varrho},$$

für $0 \le k < p$, und $\mathbf{Pr}(R \ge p+1) = 0$.

Beweis: Die Modifikation vom idealen Algorithmus (der zur Zufallsvariablen Z_k führt) zum realen Algorithmus kann Rundenanzahlen nur verringern. Daher gilt $\mathbf{Pr}(R \ge k+1) \le \mathbf{Pr}(Z_k = 0)$. Wegen $r \neq 0$ ist $Z_p \ge 1$ garantiert. □

[2] In der Originalarbeit von Chor und Goldreich wurde die Chebychev-Ungleichung (Fakt 2.3.4) 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.

Für $k \geq \frac{1-\varrho}{\varrho} = \frac{1}{\varrho} - 1$ liefert die erste Ungleichung im Lemma die Schranke $\mathbf{Pr}(R \geq k+1) \leq \frac{1}{2}$. (Dies ist für $k \geq \lfloor 1/\varrho \rfloor$ sicher erfüllt.) Es ist bemerkenswert, dass selbst unter Aufwendung von beliebig vielen Zufallsbits wie in Algorithmus 4.1.1 $1/\varrho$ viele Tests nur zu einer konstanten Erfolgswahrscheinlichkeit führen: Für $k = \lceil 1/\varrho \rceil$ haben wir (s. Prop. A.1.2 im Anhang von Kap. 2):

$$\mathbf{Pr}(\text{die ersten } k \text{ Tests sind erfolglos}) \leq (1-\varrho)^{\lceil 1/\varrho \rceil} < e^{-1} \approx 0{,}368\,;$$

dabei gilt für $\varrho \leq \frac{1}{4}$ die Abschätzung $(1-\varrho)^{1/\varrho} \geq 0.31$ und natürlich $\lim_{\varrho \searrow 0}(1-\varrho)^{1/\varrho} = e^{-1}$. 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 nun Lemma 4.1.7 benutzen, um die *erwartete Rundenzahl* nach oben abzuschätzen. Dabei verwenden wir unser Wissen, dass es im realen Algorithmus (mit $r \neq 0$) mehr als p Runden auf keinen Fall geben kann. Mit dem Lemma und mit Fakt 2.2.9 sehen wir:

$$\mathbf{E}(R) = \sum_{k \geq 0} \mathbf{Pr}(R \geq k+1) \leq \sum_{0 \leq k < p} \frac{1}{1+k\varrho}. \tag{4.1.4}$$

Wir schätzen ab:

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

Mit (4.1.4) folgt:

$$\mathbf{E}(R) \leq 1 + \int_{0}^{p-1} \frac{dx}{1+x\varrho} < 1 + \int_{0}^{p} \frac{dx}{1+x\varrho}.$$

Da $\int \frac{dx}{1+x\varrho} = \ln(1+x\varrho)/\varrho + C$, folgt

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

(Zur allgemeinen Technik der Abschätzung von Summen durch Integrale siehe Abschnitt A im Anhang.)

Satz 4.1.8
Sei $\emptyset \neq B \subseteq A = \mathbb{Z}_p$. Die erwartete Anzahl von Tests „$x \in B$?“ bei der paarweise unabhängigen Suche nach einem Element von B ist $\leq 1 + \ln(1+|B|)/\varrho$, für $\varrho = |B| \,/\, |A|$. Es werden $2\lceil \log p \rceil$ Zufallsbits benötigt (entsprechend zwei Zufallselementen von A).

Das Ergebnis sollte man mit dem Wert $1/\varrho$ aus (4.1.1) 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. (Im Fall der Suche nach einer Primzahl in $[2^{\ell-1}, 2^\ell)$ weiß man nach dem Primzahlsatz (s. Kap. 5), dass $|B| = \Theta(2^\ell/\ell)$; damit erhält man als erwartete Rundenzahl $\Theta(\ell^2)$ gegenüber von $\Theta(\ell)$ für die vollständig zufällige Suche.)

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 4.1.9.

Algorithmus 4.1.9 ***Verdoppelte paarweise unabhängige Suche***

INPUT: Menge $A = \mathbb{Z}_p$ (Körper) mit unbekannter, nichtleerer Teilmenge B
METHODE:

```
1   Wähle r1, r2, s1, s2 aus Z_p zufällig;
2   Setze r1 ← max{1, r1} und r2 ← max{1, r2};
3   x1 ← (s1 − r1) mod p;
4   x2 ← (s2 − r2) mod p;
5   repeat
6       x1 ← (x1 + r1) mod p;
7       if x1 ∈ B then return x1;
8       x2 ← (x2 + r2) mod p;
9       if x2 ∈ B then return x2.
```

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.

Rechenzeit: Sei R die Anzahl der Runden bis zum Erfolg (mit jeweils zwei Tests „$x \in B$?“). Wir schätzen $\mathbf{E}(R)$ 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

$$Y_i^{(1)} := [X_i^{(1)} \in B] \quad \text{und} \quad Y_i^{(2)} := [X_i^{(2)} \in B],$$

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 wieder (4.1.3) und erhalten:

$$\begin{aligned}\mathbf{Pr}(\text{die ersten } k \text{ Runden sind erfolglos}) &= \mathbf{Pr}(Z_k^{(1)} = 0 \wedge Z_k^{(2)} = 0) \\ &= \mathbf{Pr}(Z_k^{(1)} = 0) \cdot \mathbf{Pr}(Z_k^{(2)} = 0) \le \frac{1}{(1+k\varrho)^2}\,. \qquad (4.1.5)\end{aligned}$$

Wie oben können wir mit Fakt 2.2.9 die erwartete Rundenzahl in der **repeat**-Schleife abschätzen (zur Abschätzung der Reihe durch das Integral vgl. wieder Abschnitt A):

$$\begin{aligned}\mathbf{E}(R) &= \sum_{k\ge 0} \mathbf{Pr}(R \ge k+1) = \sum_{k\ge 0} \mathbf{Pr}(\text{die ersten } k \text{ Runden sind erfolglos}) \\ &\overset{(4.1.5)}{\le} \sum_{k\ge 0} \frac{1}{(1+k\varrho)^2} = 1 + \sum_{k\ge 1} \frac{1}{(1+k\varrho)^2} \\ &< 1 + \int_0^\infty \frac{dx}{(1+x\varrho)^2} = 1 + \Big[-\frac{1/\varrho}{1+x\varrho}\Big]_0^\infty = 1 + \frac{1}{\varrho}.\end{aligned}$$

Satz 4.1.10
Sei $\emptyset \ne B \subseteq A = \mathbb{Z}_p$. Die erwartete Anzahl von Tests „$x \in B$?“ bei der **verdoppelten** zweifach unabhängigen Suche nach einem Element von B ist $\le 2(1 + 1/\varrho)$, für $\varrho = |B| \,/\, |A|$. Es werden $4\lceil \log p \rceil$ Zufallsbits benötigt.

Dieses Ergebnis macht sich neben $1/\varrho$ für die völlig zufällige Suche sehr gut!

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 $x_1 < x_2 < \cdots < x_n$

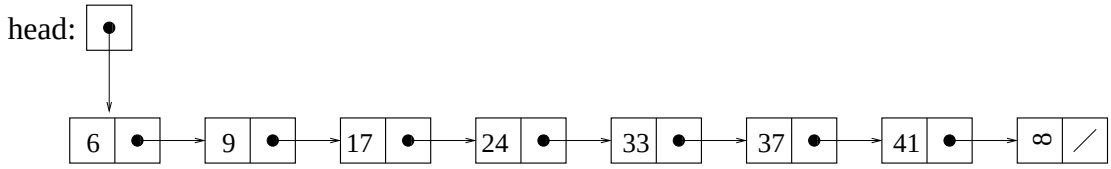


Abbildung 4.2.1: Geordnete lineare Liste, $n = 8$.

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 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 Listenimplementierungen, wie sie von den Programmiersprachen bereitgestellt werden, nicht der Fall. Allerdings ist es leicht, diese Voraussetzung herzustellen. Wir nennen drei Möglichkeiten.

- In Abbildung 4.2.2 ist die erste Möglichkeit dargestellt. Man könnte zur Implementierung der Liste vom Programm aus ein Array `A[1..`m`] of listelem` von Listenelementen kreieren und beim Einfügen in die Liste durch geeignete Verwaltung dafür sorgen, dass die *benutzten* Listenelemente immer einen vollen Anfangsabschnitt der Länge n 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 4.2.3 zeigt eine zweite Möglichkeit. Eine lineare Liste lässt sich auch „zu Fuß“ realisieren, mit Hilfe eines Arrays `value[1..`m`] of data`, das die Daten enthält (im Beispielbild sind dies auch Zahlen), und eines zweiten Arrays `next[1..`m`] of int`, das die Zeiger als explizite Zahlen in $\{1, \ldots, n\}$ enthält.

Der Listenbeginn ist durch einen Wert `head` gegeben. (`free` enthält die Nummer der ersten freien Zelle, `maxlength` die Länge m des Arrays.) Um die Liste zu durchlaufen, beginnt man mit dem Wert j_1 in `head` – im Beispiel ist dies $j_1 = 4$ – und verfolgt iterativ die `next`-Werte, mt $j_{r+1} :=$ `next[`j_r`]` bis zum Listenende, das durch den Wert `next[`j_n`]`$= 0$ angezeigt wird.) Dabei müssen die benutzten Arrayzellen in beiden Arrays stets einen zusammenhängenden Anfangsabschnitt bilden. Die Implementierung von Einfüge- und Lösch-Operationen in einer solchen Struktur ist eine Übungsaufgabe für die ersten Semester. (Beim Löschen muss man zweimal suchen, um die kompakte Darstellung als Anfangsabschnitt aufrechtzuerhalten.)

- Eine dritte Möglichkeit ist es, zusätzlich zur Liste ein Array `p[1..`m`]` mitzuführen, das im Abschnitt `p[1..`n`]` einen Zeiger auf jedes Listenelement enthält. (Diese Lösung kostet natürlich zusätzlichen Platz. Sie hat aber den Vorteil, dass man beim Löschen in der linearen Liste nur einmal suchen muss.)

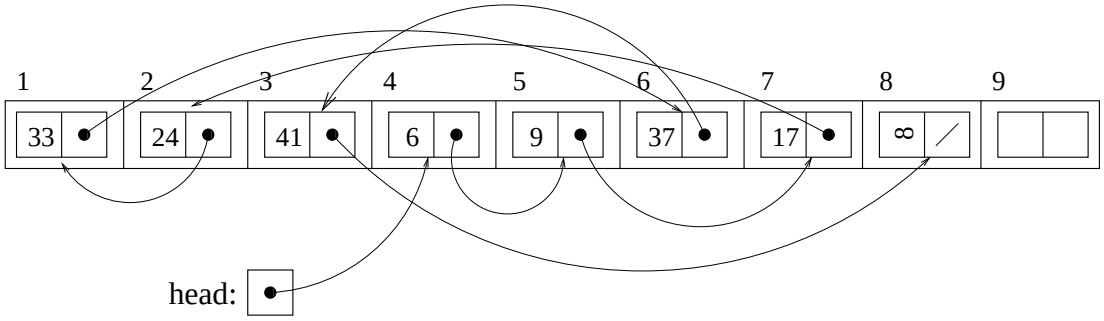


Abbildung 4.2.2: Geordnete lineare Liste, Listenelemente in Array, $m = 9$, $n = 8$.

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 sind erlaubt, man kümmert sich also nicht darum) und sucht unter diesen den größten Eintrag x_{i_0} heraus, der $< x$ ist. Mit dem Nachfolger dieses Listenelements startend sucht man das erste Element in der Liste mit einem Schlüssel, der $\geq x$ ist. Leichte Komplikationen werden dadurch verursacht, dass x kleiner als alle Einträge in der Liste sein könnte (dann ist das erste Listenelement die Ausgabe) oder $x > x_1$ gilt und alle zufällig gewählten Elemente mindestens so groß wie x sind (dann sucht man in der Liste

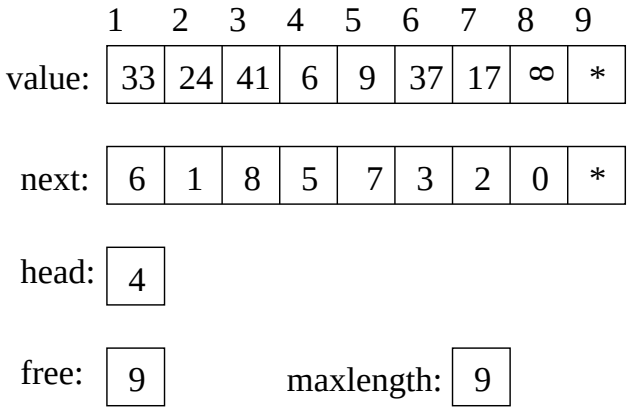


Abbildung 4.2.3: Geordnete lineare Liste in 2 Arrays, $m = 9$, $n = 8$.

startend mit dem zweiten Element x_2). Algorithmus 4.2.1 stellt dies systematisch dar. Die Zufallsauswahl ist Phase 1, die lineare Suche (inklusive Sonderfall) Phase 2.

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:

```
   // Trivialitätstest und Initialisierung:
1    if x ≤ head.key then return head else best ← head; bestkey ← head.key;
   // Phase 1: Wähle L Einträge zufällig, bestimme den größten, der < x ist:
2    repeat L times
3        rand ← (Zeiger auf) zufälliges Listenelement;
4        if rand.key < x and bestkey < rand.key
5        then best ← rand; bestkey ← rand.key;
   // Phase 2: Lineare Suche ab dem auf best folgenden Listenelement:
6    iter ← best;
7    repeat iter ← iter.next until iter.key ≥ x;  // endet wegen x_n = ∞
8    return iter.
```

Analyse: Korrektheit. Zeile 1 testet, ob das erste Listenelement die korrekte Ausgabe ist. Falls dies nicht so ist, ist der erste Listeneintrag kleiner als x, und x_k steht

weiter hinten in der Liste. *Phase 1*: In Zeilen 2–5 werden Listeneinträge zufällig gewählt. Unter ihnen wird der größte Eintrag ausgewählt, der $\leq x$ ist („`best`“) – wenn es unter den zufällig gewählten Einträgen keinen gibt, der kleiner als x ist, bleibt es beim ersten Listeneintrag x_1. Der so bestimmte Listeneintrag steht sicher *echt vor* x_k, dem gesuchten Eintrag. Es folgt *Phase 2*. Ab dem auf `best` folgenden Listeneintrag wird linear der erste gesucht, der einen Schlüssel $\geq x$ hat. Ein solcher existiert (weil $x_n = \infty$), und er ist das korrekte Ergebnis.

Analyse: Zeitbedarf. Die Initialisierung benötigt einen Schlüsselvergleich. Phase 1 benötigt höchstens $2L$ 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 4.2.4):

$$\begin{aligned} Y_2 \geq j &\Rightarrow \text{in Phase 2 werden } x_{k-j}, \ldots, x_{k-1} \text{ mit } x \text{ verglichen} \\ &\Rightarrow \text{in Phase 1 werden } x_{k-j}, \ldots, x_{k-1} \text{ nicht gewählt.} \end{aligned}$$

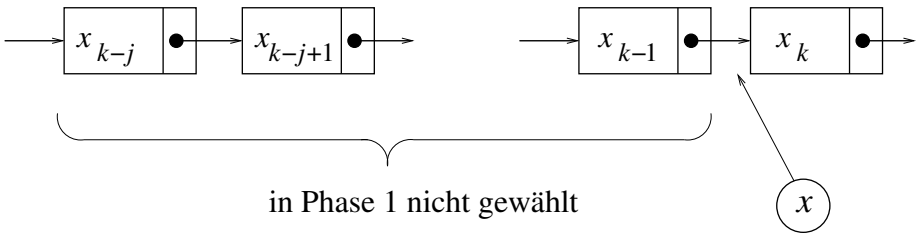


Abbildung 4.2.4: Listeneinträge, die nicht gewählt wurden, wenn $Y_2 \geq j$ ist.

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

$$\left(\frac{n-j}{n}\right)^L.$$

Damit (mit Fakt 2.2.9):

$$\mathbf{E}(Y_2) = \sum_{j \geq 1} \mathbf{Pr}(Y_2 \geq j) \leq \sum_{1 \leq j \leq n} \left(1 - \frac{j}{n}\right)^L.$$

Die Summe schätzen wir nach oben durch ein Integral ab, wie in Abschnitt A beschrieben. Die Funktion $f\colon t \mapsto (1-\frac{t}{n})^L$ ist im Intervall $[0,n]$ monoton fallend. Lemma A.0.1(b) mit $a=0$ und $b=n$ liefert

$$\sum_{1\le j\le n}\left(1-\frac{j}{n}\right)^L \le \int_0^n \left(1-\frac{t}{n}\right)^L dt = \Big[-\frac{n}{L+1}\Big(1-\frac{t}{n}\Big)^{L+1}\Big]_0^n = \frac{n}{L+1}.$$

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

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

Wir setzen nun L so fest, dass dieser Ausdruck möglichst klein wird. Standardmethoden (setze Ableitung von $2x + n/(x+1)$ gleich 0, was $x = \sqrt{n/2}-1$ liefert) ergeben als fast optimale Wahl $L = \lfloor\sqrt{n/2}\rfloor$. Damit wird

$$\mathbf{E}(Y) \le 2 + 2\Big\lfloor\sqrt{n/2}\Big\rfloor + \frac{n}{\Big\lfloor\sqrt{n/2}\Big\rfloor+1} \le 2 + \sqrt{2n} + \frac{n}{\sqrt{n/2}} = 2 + 2\sqrt{2n} = O(\sqrt{n}).$$

Satz 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 $\lceil \log n \rceil$ 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.

Eine durch die Zufallsauswahl von Listenelementen inspirierte Datenstruktur ist die *Skipliste* [William Pugh: Skip Lists: A Probabilistic Alternative to Balanced Trees. Commun. ACM 33(6): 668-676 (1990)]. Dabei wird diese Zufallsauswahl iteriert: $p \in (0,1]$ sei konstant gewählt, zum Beispiel $p = \frac{1}{2}$. Aus der Originalliste wird jedes Element mit Wahrscheinlichkeit p gewählt, und Zeiger auf die gewählten Elemente bilden die Liste der Stufe 1. Aus der Liste der Stufe i wird auf dieselbe Weise die Liste der Stufe $i+1$ gebildet, für $i = 1, 2, 3, \ldots$. Im Erwartungswert hat die Liste der Stufe i etwa $p^i n$ Einträge, und mit hoher Wahrscheinlichkeit gibt es nur $O(\log(1/p))$ viele Stufen. Suchen erfolgen erst in der Liste der höchsten Stufe, dann in der darunterliegenden, usw. Es ergeben sich erwartete Zeiten für Suchen, Einfügen und Löschen von $O(\log n)$. Details zur Datenstruktur *Skipliste* findet man zum Beispiel in dem Buch „Algorithmen und Datenstrukturen“ von Ottmann und Widmayer.

4.3 Quickselect

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

Definition 4.3.1
Wenn $S = (a_1, \ldots, a_n)$ eine Folge von n Elementen von U ist (Wiederholungen sind erlaubt!) und $1 \leq k \leq n$ ist, dann heißt $a \in \{a_1, \ldots, a_n\}$ **das Element von Rang k** in S, wenn

$$|\{i \mid 1 \leq i \leq n, a_i < a\}| < k \leq |\{i \mid 1 \leq i \leq n, a_i \leq a\}|.$$

Das Element a von Rang $\lceil n/2 \rceil$ heißt der **Median** von S.

Das Selektionsproblem besteht darin, zu einem gegebenen Array $A[1\,..\,n]$ mit Einträgen $a_1, \ldots, a_n$ den Eintrag von Rang k zu finden. Die „einfachste“ Möglichkeit hierfür ist, die Folge zu sortieren und das Element an Position k auszugeben. (Rechenzeit: $O(n \log n)$.)

Beispiel: Sortieren von $S = (4, 1, 27, 4, 15, 11, 7, 30)$ liefert $(1, 4, 4, 7, 11, 15, 27, 30)$. Also ist 4 das Element vom Rang 2 (und vom Rang 3) und 7 ist der Median. Wenn man $S = (1, 30, 4, 11, 15, 27, 7)$ sortiert, ergibt sich $(1, 4, 7, 11, 15, 27, 30)$. Also ist 30 das Element vom Rang 7, und 11 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.

Eine weitere Möglichkeit zu demonstrieren, dass a das Element von Rang k ist, ist, A so umzusortieren, dass erst Einträge $\leq a$ kommen, dann a in Position k, dann Einträge $\geq a$. Im ersten Beispiel zeigt die Anordnung $(4, 1, 4, 11, 7, 27, 30, 15)$, dass 4 das Element von Rang 3 ist, die Anordnung $(4, 1, 4, 7, 27, 30, 11, 15)$, dass 7 der Median ist. Wir bemerken: Wenn eine solche Anordnung gegeben ist, genügen $n - 1$ Vergleiche, um die Behauptung zu verifizieren, dass an Stelle k der Eintrag von Rang k steht. Wir betrachten nun einen Algorithmus, der in (erwarteter) Linearzeit eine solche Anordnung herstellt. Für die Zwecke dieser theoretischen Diskussion nehmen wir an, dass alle Einträge verschieden sind.

Gegeben sei also ein Array `A[1..n]`, in dem die verschiedenen Einträge $a_1 < \cdots < a_n$ aus einem totalgeordneten Bereich $(U, <)$ in **irgendeiner Reihenfolge** 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[`$1 \ldots k-1$`]` kleiner als a_k und die in `A[`$k+1 \ldots n$`]` größer als a_k sind.

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

Algorithmus und Analyseprinzip finden sich separat auf Folien aus einer früheren Version der Vorlesung „Algorithmen und Datenstrukturen“ (separat bereitgestellt). Der Ansatz der Analyse ähnelt dem der Analyse von Quicksort aus Abschnitt 1.4. Das zentrale Ergebnis ist dabei Folgendes.

Satz 4.3.2
Der Algorithmus **QuickSelect** macht auf Eingaben der Länge n im erwarteten Fall weniger als $4n$ Vergleiche.

Auf den Folien wird die Summation der Terme der letzten Summe

$$S = \sum_{1 \le i < k < j \le n} \frac{1}{j - i + 1}$$

nur sehr summarisch durchgeführt, mit dem Ergebnis, dass diese Summe kleiner als $n - 2$ ist. Damit beträgt der Beitrag der Paare (i, j) mit $i < k < j$ zur erwarteten Vergleichsanzahl weniger als $2n$.

Hier wollen wir für Interessierte eine noch schärfere Schranke herleiten.

(Beginn nicht prüfungsrelevanter Abschnitt.)

Wir analysieren dazu die Summe genauer. Wie auf den Folien werden die Summanden in die nachfolgende $(k-1) \times (n-k)$-Matrix geschrieben (für $k \le n/2$):

$$\begin{pmatrix}
\frac{1}{k+1} & \frac{1}{k+2} & \cdots & \frac{1}{n-k} & \frac{1}{n-k+1} & \frac{1}{n-k+2} & \cdots & \cdots & \frac{1}{n-1} & \frac{1}{n} \\
\frac{1}{k} & \frac{1}{k+1} & \frac{1}{k+2} & \cdots & \frac{1}{n-k} & \frac{1}{n-k+1} & \frac{1}{n-k+2} & \cdots & \frac{1}{n-2} & \frac{1}{n-1} \\
\vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots & \vdots \\
\frac{1}{4} & \frac{1}{5} & \cdots & \frac{1}{k+1} & \frac{1}{k+2} & \cdots & \frac{1}{n-k} & \frac{1}{n-k+1} & \frac{1}{n-k+2} & \cdots \\
\frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{k} & \frac{1}{k+1} & \frac{1}{k+2} & \cdots & \frac{1}{n-k} & \frac{1}{n-k+1} & \frac{1}{n-k+2}
\end{pmatrix}$$

Auf den Folien wurde gesagt, dass die Summe aller Einträge kleiner als n sein muss, weil die (konstanten) Einträge in jeder Diagonalen sich höchstens zu 1 summieren und es exakt $n-2$ Diagonalen gibt.

Wir analysieren die Summe der Matrixeinträge nun genauer. Wir teilen sie dazu in drei Teile ein:

Teil I, Summe B_1: Einträge strikt unterhalb der Diagonalen mit den $\frac{1}{k+1}$-Einträgen,

Teil II, Summe B_2: Einträge ab der Diagonalen mit den $\frac{1}{k+1}$-Einträgen (startet in linker oberer Ecke) bis zur Diagonalen mit den $\frac{1}{n-k}$-Einträgen (endet zwei Positionen links von rechter unterer Ecke),

Teil III, Summe B_3: Einträge strikt oberhalb der Diagonalen mit den $\frac{1}{n-k}$-Einträgen.

Setze $\alpha := k/n$. Dann gilt $0 < \alpha \leq \frac{1}{2}$. Offenbar gilt:

$$B_1 = \frac{k-2}{k} + \frac{k-3}{k-1} + \frac{k-4}{k-2} + \cdots + \frac{1}{3} < k-2.$$

Nach Beispiel A.0.2(ii) im Anhang haben wir $\frac{1}{u+1} + \frac{1}{u+2} + \cdots + \frac{1}{v} \leq \ln\left(\frac{v}{u}\right)$. Damit:

$$\begin{aligned} B_3 &= \frac{1}{n} + \frac{2}{n-1} + \frac{3}{n-2} + \cdots + \frac{k-1}{n-k+2} + \frac{k-1}{n-k+1} \\ &\leq \left(\frac{n+1}{n} - 1\right) + \left(\frac{n+1}{n-1} - 1\right) + \cdots + \left(\frac{n+1}{n-k+1} - 1\right) \\ &= (n+1)\left(\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{n-k+1}\right) - k \\ &\leq (n+1)\ln\left(\frac{n}{n-k}\right) - k. \end{aligned}$$

Wir erhalten, weil $\ln(1/(1-\alpha)) \leq \ln 2 < 1$:

$$B_1 + B_3 < (n+1)\ln\left(\frac{n}{n-k}\right) - 2 = (n+1)\ln\left(\frac{1}{1-\alpha}\right) - 2 < -n\ln(1-\alpha).$$

(Der Logarithmus ist natürlich negativ!) Schließlich, nochmals mit Beispiel A.0.2(ii):

$$\begin{aligned} B_2 &= (k-1)\left(\frac{1}{k+1} + \frac{1}{k+2} + \frac{1}{k+3} + \cdots + \frac{1}{n-k}\right) \\ &\leq (k-1)\ln\left(\frac{n-k}{k}\right) < \alpha n \ln\left(\frac{n-k}{k}\right) \\ &= \alpha n \ln\left(\frac{1-\alpha}{\alpha}\right) = n \cdot (\alpha\ln(1-\alpha) - \alpha\ln(\alpha)). \end{aligned}$$

Wir fassen zusammen:

$$\begin{aligned} S &= B_1 + B_2 + B_3 \\ &\leq n \cdot (-\ln(1-\alpha) + \alpha\ln(1-\alpha) - \alpha\ln(\alpha)) \\ &= n \cdot (-(1-\alpha)\ln(1-\alpha) - \alpha\ln\alpha) \\ &= n \cdot H_e(\alpha), \end{aligned}$$

wobei $H_e(\alpha)$ die Entropie der Verteilung $(\alpha, 1-\alpha)$ zur Basis e ist. Bezüglich der in der Informatik üblichen binären Entropie $H_2(\alpha) = -(1-\alpha)\log_2(1-\alpha) - \alpha\log_2(\alpha)$ ergibt sich mit Hilfe der Beziehung $H_e(\alpha) = (\ln 2)H_2(\alpha)$:

$$S \leq n \cdot (\ln 2) H_2(\alpha).$$

Insgesamt, weil $\alpha \leq \frac{1}{2}$:

$$\mathbf{E}(C_k) \leq n \cdot (2 + (2 \ln 2) H_2(\alpha)) .$$

Der schlechteste Fall tritt für $\alpha = \frac{1}{2}$, also $k = n/2$, auf. Dies ist der Fall des Medians. Dann ist $H_2(\frac{1}{2}) = 1$, und $\mathbf{E}(C_{n/2}) \leq (2 + (2 \ln 2))n \approx 3.3863n$.

(Ende nicht prüfungsrelevanter Abschnitt.)

Satz 4.3.3
Quickselect mit $k = \alpha n$ hat erwartete Vergleichsanzahl $\leq (2 + (2 \ln 2) H_2(\alpha)) \cdot n$.

Der nächste Abschnitt 4.4 zeigt, dass (mit einem anderen, etwas komplizierteren Algorithmus) die Vergleichsanzahl noch auf etwa $\frac{3}{2}n$ verbessert werden kann.

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***.

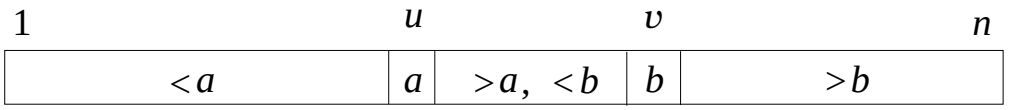
Zur Vereinfachung der Analyse nehmen wir an, dass $n/2$ und $n^{1/4}$ ganze Zahlen sind. (Andernfalls arbeitet man mit geeigneten gerundeten Werten.)

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 $v - u = o(n/\log n)$ gilt, 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 $b_1 \leq \cdots \leq b_L$ 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 (Algorithmus 4.4.1).

Algorithmus 4.4.1 ***Random Sample Median***
Input: Array `A[1..n]` (ungeordnet)
Methode:
(1) $L \leftarrow n^{3/4}$;
(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 ist $b_1 \leq b_2 \leq \cdots \leq b_L$.
(4) $a \leftarrow b_{L/2-\sqrt{n}}$;
(5) $b \leftarrow b_{L/2+\sqrt{n}}$;
(6) **partitioniere**(a, b, u, v), mit Ausgabewerten u und v, das bewirkt:
für geeignete u und v arrangiere `A[1..n]` so um, dass gilt:
Einträge $< a$ stehen in $\texttt{A}[1\,..\,u-1]$;
Einträge $> b$ stehen in $\texttt{A}[v+1\,..\,n]$;
a steht in $\texttt{A}[u]$, b steht in $\texttt{A}[v]$;
Einträge zwischen a und b stehen in $\texttt{A}[u+1\,..\,v-1]$;
(7) **Fall 1:** $u > n/2$ oder $v < n/2$: **return** „?“;
(8) **Fall 2:** $v - u > 2D \cdot n^{3/4}$: **return** „?“; // D: eine noch festzulegende Konstante
(9) **Fall 3:** Sonst. Sortiere $\texttt{A}[u+1\,..\,v-1]$, z.B. mit Mergesort; **return** $\texttt{A}[n/2]$.

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 in die drei Abteilungen „$< a$“, „zwischen a und b (einschließlich)“ und „$> b$“ werden im schlechtesten Fall $2n$ Vergleiche benötigt. Wenn man für jeden Eintrag x *zufällig* entscheidet, ob man x zuerst mit a oder mit b vergleicht, ist mit hoher Wahrscheinlichkeit die Zahl von Vergleichen für Zeile (6) 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. Wir können uns o. B. d. A. vorstellen, dass $a_1 < \cdots < a_n$ gilt. (Diese Anordnung wird vom Algorithmus natürlich nicht hergestellt. Da über die zufälligen Indizes $j_1, \ldots, j_L$ Einträge zufällig gewählt werden, ist die Anordnung im Array `A` 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 häufig als $(L/2 - \sqrt{n})$-mal. Andererseits erwarten wir genau $L/2$ Treffer. Diese Unterschreitung des Erwartungswertes ist sehr unwahrscheinlich, wie man durch Anwendung der Hoeffding-Schranke (Abschnitt 2.6) sieht. Wir definieren:

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

Dann haben wir, nach einer Form der Hoeffding-Schranke (Korollar 2.6.4, Formel (2.6.18) in Abschnitt 2.6):

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

Da man $\mathbf{Pr}(A_2)$ analog abschätzen kann, ergibt sich als Schranke für die Wahrscheinlichkeit, dass Fall 1 eintritt, $2e^{-n^{1/4}}$, eine mit wachsendem n sehr rasch verschwindende Zahl.

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 $v - n/2$ und $n/2 - u$ beide $\leq D \cdot n^{3/4}$ wären, hätten wir $v - u \leq 2D \cdot n^{3/4}$.)

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 $\texttt{A}[j_1], \ldots, \texttt{A}[j_L]$ weniger als $L/2+\sqrt{n}$ oft die Menge $S_{\text{unten}} := \{a_1, \ldots, a_{n/2+D\cdot n^{3/4}}\}$ 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 $\leq L/2 + \sqrt{n}$ für nicht zu kleine D extrem unwahrscheinlich ist.

Definiere:

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

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

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

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 durchgeführten 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.

A Ergänzung: Abschätzung von Summen durch Integrale

Im Abschnitt über zweifach unabhängige Suche (4.1) haben wir zweimal eine unendliche Reihe durch ein Integral abgeschätzt. Auch bei der genauen Analyse von Quickselect (Abschnitt 4.3) ergab sich die Notwendigkeit für eine solche Abschätzung. 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 A.0.1
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 $a < b$ ganze Zahlen sind, gilt*:*

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

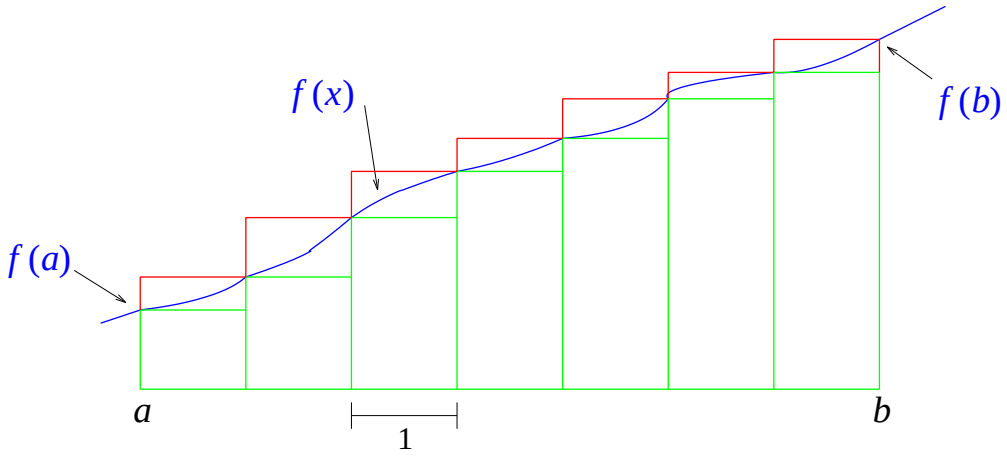
(b) Wenn f monoton fallend ist und $a < b$ ganze Zahlen sind, gilt*:*

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

(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:

$$\int_{i-1}^{i} f(x)\,dx \le f(i), \text{ für } a < i \le b, \text{ und } f(i) \le \int_{i}^{i+1} f(x)\,dx, \text{ für } a \le i < b. \quad \text{(A.0.6)}$$

Wir summieren die erste Ungleichung in (A.0.6) über $a < i \le b$ und erhalten $\int_a^b f(x)\,dx \le \sum_{a<i\le b} f(i)$. 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:

$$\int_{i}^{i+1} f(x)\,dx \le f(i), \text{ für } a \le i < b, \text{ und } f(i) \le \int_{i-1}^{i} f(x)\,dx, \text{ für } a < i \le b. \quad \text{(A.0.7)}$$

Wir summieren die erste Ungleichung in (A.0.7) über $a \le i < b$ und erhalten $\int_a^b f(x)\,dx \le \sum_{a\le i<b} f(i)$. Die zweite Ungleichung summieren wir über $a < i \le b$ und erhalten $\sum_{a<i\le b} f(i) \le \int_a^b f(x)\,dx$.

(c) Wenn $b = \infty$, summieren wir die erste Ungleichung in (A.0.7) über alle $i \ge a$ und die zweite über alle $i > a$, um die behaupteten Ungleichungen zu erhalten. □

Beispiele A.0.2
(i) Mit $f(x) = 1/x$ und $a = 1$ und $b = n + 1$ erhalten wir mit Lemma A.0.1(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 $f(x) = 1/x$ und $0 \le a < b$ beliebig gilt, nach Lemma A.0.1(b):

$$\sum_{a<i\le b} \frac{1}{i} \le \int_a^b \frac{dx}{x} = \int_1^b \frac{dx}{x} - \int_1^a \frac{dx}{x} = \ln b - \ln a = \ln\left(\frac{b}{a}\right).$$

(iii) Mit $f(x) = 1/x^2$ und $b = \infty$ erhalten wir mit Lemma A.0.1(c):

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

(iv) Mit $f(x) = \ln(x)$ und $a = 1$ und $b = n$ können wir Lemma A.0.1(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:

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

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 A.0.1(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 $n!$.

Eine viel schärfere Abschätzung bietet die *Stirling'sche Formel*, grob $n! \sim \sqrt{2\pi n}(n/e)^n$, die zeigt, dass der wahre Wert von $n!$ (geometrisch) zwischen $e(n/e)^n$ und $(en)(n/e)^n$ liegt, und dass die wirkliche Konstante $\sqrt{2\pi}$ ist. Abschätzungen, die auf etwa $1 + \frac{1}{144n^2}$ genau sind, bieten die folgenden Ungleichungen, die für alle $n \geq 1$ gelten:

$$\boxed{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\frac{1}{12n+1}} \;\leq\; n! \;\leq\; \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\frac{1}{12n}}.}$$