Theoretische Informatik III (sotech)

Prof. Dr. Ulrich Hertrampf
Universität Stuttgart
Institut für Formale Methoden der Informatik

Sommer 2005

1 Strategien für den Algorithmenentwurf

1.1 Divide and Conquer

Die erste Entwurfsstrategie, die wir betrachten, ist die Divide and Conquer-Methode. Dieses Verfahren gliedert sich in drei Teile auf:

1. Zerlege das Problem in zwei möglichst gleichgroße Teilprobleme.
2. Löse die Teilprobleme einzeln.
3. Erzeuge aus den Teillösungen die Gesamtlösung.

Bemerkung:

- *Theoretisch reicht bei der Zerlegung des Problems oft ein Größenverhältnis* p *zu* $(1 - p)$*, mit festem* $0 < p < 1$*.*
- *Beim Lösen der Teilprobleme ist der Ansatz einer Rekursion häufig auch der Ansatz zur Parallelverarbeitung.*

Beispiel: Ein typisches Beispiel für das Divide and Conquer–Verfahren ist *Mergesort*. Dabei wird die Eingabesequenz in zwei Teilsequenzen aufgeteilt, die getrennt sortiert und dann wieder zu einer Sequenz gemischt werden.

Ein vergleichbarer Ansatz wird bei *Quicksort* angewendet: ein Feld wird durch ein Pivotelement in zwei Teilfelder aufgeteilt, die dann getrennt sortiert werden. Das Problem bei Quicksort ist die Wahl des Pivotelements, um möglichst gleichgroße Teilfelder zu erhalten.

Wie kann man die Rechenzeiten bei diesen Algorithmen ermitteln?

Wie verhält sich z.B. Quicksort, wenn in jedem Schritt jedes Teilfeld maximal 99 Prozent der vorherigen Feldgröße hat?

Wie ist es, wenn in jedem Schritt ein Teilfeld der Größe maximal 10 000 000 (also zehn Millionen) abgeteilt wird?

1.1.1 Multiplikation ganzer Zahlen

Hat man zwei Binärzahlen der Länge n, so braucht man bei der grundschulmäßigen Multiplikation der beiden Zahlen i.a. $\mathcal{O}(n^2)$ Operationen.
Seien nun die beiden Zahlen r, s wie folgt zusammengesetzt:

$$r = \boxed{\quad A \quad}\boxed{\quad B \quad}$$

$$s = \boxed{\quad C \quad}\boxed{\quad D \quad}$$

Dabei sind A die ersten k Bits von r und B die letzten k Bits von r. Analoges gilt für C, D und s. Wir können also r und s wie folgt schreiben :

$$r = A\,2^k + B; \qquad s = C\,2^k + D$$

Daraus folgt:

$$r\,s = A\,C\,2^{2k} + (A\,D + B\,C)\,2^k + B\,D$$

Statt diesen Ansatz zu verfolgen, berechnen wir rekursiv die drei Zahlen AC, $(A-B)(D-C)$ und BD. Damit können wir rs mit nur drei Multiplikationen von Zahlen mit höchstens k Bits berechnen:

$$rs = A\,C\,2^{2k} + (A-B)\,(D-C)\,2^{k} + (B\,D + A\,C)\,2^{k} + B\,D$$

Hieraus erhält man als Aufwand:

$$t_{\mathsf{mult}}(n) = 3 \cdot t_{\mathsf{mult}}(n/2) + \mathcal{O}(n) = \mathcal{O}(n^{\frac{\log 3}{\log 2}}) = \mathcal{O}(n^{1.58496\ldots}).$$

In dieser Formel steht n für die Bitlänge der Zahlen und der Ausdruck $\mathcal{O}(n)$ für den Aufwand der Additionen.

Wir haben also durch den Teile-und-Beherrsche Ansatz den Exponenten des naiven Ansatzes von 2 auf 1.58496... heruntergesetzt.

Ein weiteres Beispiel für den Divide-and-Conquer Ansatz ist das Verfahren von Strassen (1968) für schnelle Matrixmultiplikation.

1.1.2 Matrixmultiplikation nach Strassen

Die übliche Multiplikation zweier $n \times n$ Matrizen $(a_{i,j})\,(b_{i,j}) = (\sum_{k=1}^{n} a_{i,k} b_{k,j})$ erfordert $\mathcal{O}(n^3)$ skalare Multiplikationen. Wir versuchen, die Anzahl dieser Multiplikationen mit einem Divide-and-Conquer Ansatz zu reduzieren. Dabei werden die zwei Matrizen A, B jeweils in 4 etwa gleichgroße Untermatrizen unterteilt, wobei sich das Produkt $AB = C$ wie folgt darstellen lässt:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Dabei ergeben sich folgende Beziehungen:

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

Aus der Rekursionsgleichung der Laufzeit dieses Ansatzes

$$t(n) = 8 \cdot t(n/2) + \Theta(n^2) \in \Theta(n^3)$$

sieht man, dass wir damit keine Verbesserung erreicht haben. Das Verfahren von Strassen (1968) verwendet jedoch die Tatsache, dass man das Produkt zweier 2×2 Matrizen geschickter mit nur 7 Multiplikationen berechnen kann:

$$\begin{aligned}
M_1 &:= (A_{12} - A_{22})(B_{21} + B_{22}) \\
M_2 &:= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_3 &:= (A_{11} - A_{21})(B_{11} + B_{12}) \\
M_4 &:= (A_{11} + A_{12})B_{22} \\
M_5 &:= A_{11}(B_{12} - B_{22}) \\
M_6 &:= A_{22}(B_{21} - B_{11}) \\
M_7 &:= (A_{21} + A_{22})B_{11}
\end{aligned}$$

$$
\begin{aligned}
C_{11} &:= M_1 + M_2 - M_4 + M_6 \\
C_{12} &:= M_4 + M_5 \\
C_{21} &:= M_6 + M_7 \\
C_{22} &:= M_2 - M_3 + M_5 - M_7
\end{aligned}
$$

Bitte Nachrechnen!

Wir erhalten mit der Methode von Strassen folgende Rekursionsgleichung für die Laufzeit:

$$t(n) = 7t(n/2) + \Theta(n^2)$$

Damit kann man berechnen:

$$t(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2,81\ldots})$$

Es ergibt sich ein Gewinn im Vergleich zum direkten $\Theta(n^3)$-Ansatz. (Coppersmith und Winograd konnten die obere Schranke 1987 auf $\mathcal{O}(n^{2,376})$ verbessern.)

1.1.3 Transitive Hülle und Matrixmultiplikation

Sei $A = (a_{ij})$ die Adjazenzmatrix eines gerichteten Graphen mit n Knoten. Der Warshall-Algorithmus berechnet den reflexiven transitiven Abschluss A^* in $\mathcal{O}(n^3)$ Schritten. Hierbei ist

$$A^* = \sum_{k \geq 0} A^k \qquad \text{mit } A^0 = I_n, \wedge \text{ und } \vee \text{ als skalare Multiplikation bzw. Addition}$$

Mit Induktion ergibt sich leicht, dass $A^k(i,j) = 1$ genau dann gilt, wenn es von i nach j einen Weg der Länge k gibt. Klar ist auch $A^* = \sum_{k=0}^{n-1} A^k$.

Setze $B = I_n + A$. Dann gilt $A^* = B^m$ für alle $m \geq n - 1$. Also reicht es, eine Matrix $\lceil \log_2(n-1) \rceil$-mal zu quadrieren, um A^* zu berechnen.

Wir wollen nun den Aufwand für Matrixmultiplikation mit dem für transitive Hüllenbildung vergleichen:

Sei $M(n)$ der Aufwand, zwei boolesche $n \times n$-Matrizen zu multiplizieren, und sei $T(n)$ der Aufwand, die reflexive transitive Hülle zu berechnen. Dann gilt also:

$$T(n) \in \mathcal{O}(M(n) \cdot \log n).$$

Hieraus folgt für alle $\varepsilon > 0$ nach Strassen

$$T(n) \in \mathcal{O}(n^{\log_2(7)+\varepsilon}).$$

Die Beziehung $M(n) \in \mathcal{O}(T(n))$ ist offensichtlich (unter der plausiblen Annahme $M(3n) \in \mathcal{O}(M(n))$). Denn seien A und B beliebige Matrizen, dann gilt:

$$\begin{pmatrix} 0 & A & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{pmatrix}^* = \begin{pmatrix} I_n & A & AB \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}.$$

(Daraus erhält man zunächst $M(n) \in \mathcal{O}(T(3n))$, mit $M(3n) \in \mathcal{O}(M(n))$ dann auch $M(3n) \in \mathcal{O}(T(3n))$, und somit $M(n) \in \mathcal{O}(T(n))$.)

Unter den (ebenfalls plausiblen) Annahmen $M(n) \in \Omega(n^2)$ und $M(2n) \geq (2+\varepsilon)M(n)$ zeigen wir

$$T(n) \in \mathcal{O}(M(n)).$$

Dies bedeutet: die Berechnung der transitiven Hülle ist bis auf konstante Faktoren genauso aufwändig wie die Matrixmultiplikation.

Wir müssen also nun zeigen, wie wir für eine boolesche $n \times n$-Matrix in $\mathcal{O}(M(n))$ Zeit die transitive Hülle ermitteln können:

Berechnung der transitiven Hülle:

Eingabe: $E \in \mathrm{Bool}(n \times n)$

1. Teile E in vier Teilmatrizen A, B, C, D so, dass A und D quadratisch sind und jede Matrix ungefähr die Größe $n/2 \times n/2$ hat:

$$E = \begin{pmatrix} A & B \\ C & D \end{pmatrix}.$$

2. Berechne rekursiv D^*: Aufwand $T(n/2)$.
3. Berechne $F = A + BD^*C$: Aufwand $\mathcal{O}(M(n/2))$, da $M(n) \in \Omega(n^2)$. (n^2 wird für Addition schon benötigt!)
4. Berechne rekursiv F^*: Aufwand $T(n/2)$.
5. Setze

$$E^* = \left(\begin{array}{c|c} F^* & F^*BD^* \\ \hline D^*CF^* & D^* + D^*CF^*BD^* \end{array}\right).$$

Damit erhalten wir die Rekursionsgleichung

$$T(n) \leq 2T(n/2) + c \cdot M(n) \qquad \text{für ein } c > 0.$$

Dies ergibt

$$\begin{aligned} T(n) &\leq c \cdot \left(\sum_{i \geq 0} 2^i \cdot M(n/2^i) \right) \\ &\leq c \cdot \sum_{i \geq 0} \left(\frac{2}{2+\varepsilon} \right)^i \cdot M(n) \qquad \text{(da } M(n/2^i) \leq \left(\frac{1}{2+\varepsilon} \right)^i M(n)\text{)} \\ &\in \mathcal{O}(M(n)). \end{aligned}$$

1.2 Greedy-Algorithmen

Greedy („gierig“) bezeichnet Lösungsstrategien, die auf der schrittweisen Berechnung von Teillösungen (lokalen Optima) basieren. Anders ausgedrückt nähert man sich dem Ziel dadurch, dass bei jedem Schritt mit Hilfe eines Optimum–Kriteriums der nächste Schritt berechnet wird. Dieses Verfahren eignet sich für Probleme, bei denen jede Teilfolge einer optimalen Folge auch optimal ist (*Optimalitätsprinzip*).

Beispiel: Ist in einem Graphen $u = u_0, u_1, u_2, \ldots, u_{n-1}, u_n = v$ ein kürzester Weg von Knoten u nach Knoten v, so ist immer auch $u_i, u_{i+1}, u_{i+2}, \ldots, u_j$ mit $0 \leq i < j \leq n$ ein kürzester Weg von u_i nach u_j.

Als klassisches Beispiel für die Greedy-Strategie betrachten wir im folgenden das Problem der Bestimmung kürzester Wege in einem kantengewichteten Graphen (von einer Quelle aus).

1.2.1 Kürzeste Wege in Graphen (Dijkstra-Algorithmus)

Sei ein Graph $G = (V, E, \gamma)$ gegeben, wobei V die Knotenmenge und $E \subseteq V \times V$ die Kanten sind. $\gamma : E \rightarrow \mathbb{N}_0$ ist die Gewichtsfunktion der Kanten.

Das Gewicht eines Pfades ist gleich der Summe der Kantengewichte. Sei nun $d(u, v)$ für $u, v \in V$ das Minimum der Gewichte aller Pfade von u nach v (mit der üblichen Konvention, dass $d(u, v) = \infty$ gilt, falls kein Pfad von u nach v existiert).

Die Aufgabe ist nun, bei gegebenen Graphen G und Knoten $u \in V$ für jedes $v \in V$ einen Pfad $u = u_0, u_1, u_2, \ldots, u_{n-1}, u_n = v$ mit minimalem Gewicht $\sum_{i=1}^{n} \gamma(u_{i-1}, u_i) = d(u, v)$ zu bestimmen.

Der Algorithmus von *Dijkstra* berechnet hierfür schrittweise Knotenmengen $B_i \subseteq V$ mit folgenden Eigenschaften:

1. $B_0 = \{u\}$.
2. Nach Schritt i ist B_i so berechnet, dass für jedes $w \in B_i$, $d(u, w)$ und ein zugehöriger kürzester Pfad von u nach w bekannt sind. Weiterhin gilt für alle $w' \in B_i$ und alle $z \in V \setminus B_i$: $d(u, w') \leq d(u, z)$.

In Schritt i sucht man eine Kante $(x, y) \in E$ mit den Eigenschaften

- $(x \in B_{i-1},\ y \notin B_{i-1})$, und
- $\forall (x', y') \in E, x' \in B_{i-1} \wedge y' \notin B_{i-1} : d(u, x) + \gamma(x, y) \leq d(u, x') + \gamma(x', y')$.

Anschließend wird $B_i := B_{i-1} \cup \{y\}$ und $d(u, y) = d(u, x) + \gamma(x, y)$ gesetzt.

Wir wollen den Dijkstra-Algorithmus konkret formulieren, wobei wir die genaue Datenstruktur offen lassen (Algorithmus 1). Die Idee des Algorithmus besteht darin, die Knotenmenge des Graphen in drei disjunkten Mengen zu verwalten: die Menge B (Baummenge, oder Menge der bekannten Knoten) bezeichnet diejenigen Knoten, für die ein kürzester Pfad (von u aus) bekannt ist; die Menge R (Randmenge) enthält die unmittelbaren Nachbarn der Knoten aus B, die selbst nicht zu B gehören. Für die Randknoten ist eine Abschätzung des Abstandes zu u berechnet. Schließlich ist U die Menge der „unbekannten“ Knoten. Wir gehen davon aus, dass eine Datenstruktur $v(y)$ definiert ist, die für jeden Knoten $y \in (B \cup R)$ den zugehörigen Vorgänger enthält. Weiterhin sei eine Datenstruktur $D(y)$ vorausgesetzt, so dass für jeden Knoten $y \in B$, $D(y) = d(u, y)$ und für jeden Randknoten $x \in R$, $D(x) = \min\{D(z) + \gamma(z, x) \mid z \in B\}$ gilt.

Algorithmus *Dijkstra-Algorithmus*

Eingabe : $(G = (V, E, \gamma), u)$ (* ger., kantengew. Graph G *)
(* u = Startknoten *)

var x, y : Knoten; α : **integer**;
D : array$[1 \ldots |V|]$ of **integer**;
v : array$[1 \ldots |V|]$ of Knoten;
B : Knotenmenge; (* Baumknoten *)
R : Knotenmenge; (* Randknoten *)
U : Knotenmenge; (* unbekannte Knoten *)
(* Für $v \in B$ ist $d(u, v)$ bekannt und es gilt: $D(v) = d(u, v)$. *)
(* Die Knoten in R sind von B aus direkt erreichbar, während *)
(* $U = V \setminus (B \cup R)$ die unbekannten Knoten umfasst. *)

begin
$B := \emptyset$;
$R := \{u\}$; $U := V \setminus \{u\}$; (* Initialisierung von B, R, U *)
$v(u) :=$ **nil**; $D(u) := 0$;

```
  while R ≠ ∅ do
    x := nil; α := ∞;                                   (* suche x ∈ R mit min. Abstand *)
    forall y ∈ R do
      if D(y) < α then x := y; α := D(y) endif
    endfor ;
    B := B ∪ {x};                                       (* verschiebe x von R nach B *)
    R := R\{x};
    forall (x, y) ∈ E do
      if y ∈ U then                                     (* Rand aktualisieren *)
        D(y) := D(x) + γ(x, y);
        v(y) := x; U := U\{y};
        R := R ∪ {y}
      elsif y ∈ R and D(x) + γ(x, y) < D(y) then
        D(y) := D(x) + γ(x, y);                         (* kürzerer Weg über x *)
        v(y) := x
      endif
    endfor
  endwhile
end
```

Terminierung
Nach dem i-ten Durchlauf ist $B = B_i$. Die Termination ist trivialerweise gewährleistet, denn jeder Schritt vergrößert B_i und enthält nur die von u aus erreichbaren Knoten.

Korrektheit
Wir werden die folgende Invariante beweisen:

> Für jedes $w \in B_i$ sind $d(u, w)$ und ein zugehöriger kürzester Pfad von u nach w bekannt. Weiterhin gilt für alle $w \in B_i$ und alle $z \notin B_i$: $d(u, w) \leq d(u, z)$.

Für $i = 0$ und $B_0 = \{u\}$ ist die Invariante erfüllt.

Sei $i > 0$. Wir nehmen an, dass B_{i-1} die Invariante erfüllt. Weiterhin sei $y \in R$ der im i-ten Durchlauf ausgewählte Randknoten und $x \in B_{i-1}$ sein Vorgänger im Baum, d.h. es gilt $x \in B_{i-1}$, $y \notin B_{i-1}$ und nach dem Algorithmus

$$\forall\, (x', y') \in E,\ x' \in B_{i-1},\ y' \notin B_{i-1} : d(u, x) + \gamma(x, y) \leq d(u, x') + \gamma(x', y').$$

Es reicht zu zeigen, dass

1. der Pfad von u nach x und dann direkt nach y ein kürzester Pfad von u nach y ist und
2. für alle $y' \in V \setminus B_i$ die Ungleichung $d(u, y) \leq d(u, y')$ erfüllt ist.

Sei $y' \in V \setminus B_{i-1}$ ein Knoten mit minimalem $d(u, y')$ und P' ein kürzester Pfad von u nach y', der unter allen kürzesten Pfaden von u nach y' die minimale Anzahl Zwischenknoten besitzt. Beachte, dass $y' = y$ sein könnte.

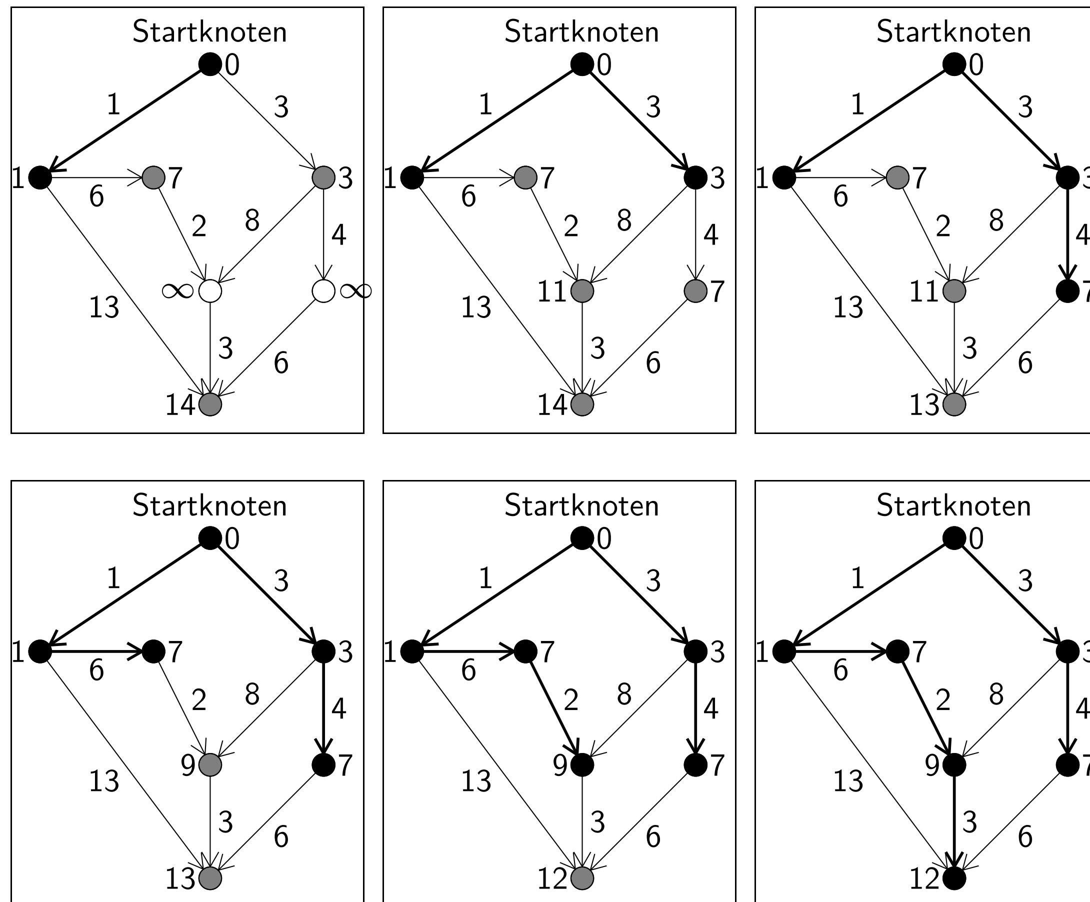
Sei x' der Vorgänger von y' auf P'. Dann ist $x' \in B_{i-1}$ (sonst wäre der Pfad von u nach x' entweder ein kürzerer Pfad oder ein Pfad mit weniger Zwischenknoten).

Damit gilt nach Wahl von y:

$$d(u, y) \leq d(u, x) + \gamma(x, y) \leq d(u, x') + \gamma(x', y') = d(u, y').$$

Nach Wahl von y' folgt daraus $d(u, y) = d(u, y') = d(u, x) + \gamma(x, y)$ und damit der erste Teil der Invariante. Außerdem haben wir den zweiten Teil der Invariante gezeigt: $d(u, y) \leq d(u, y')$ für alle $y' \in V \setminus B_i$.

Beispiel: Wir wollen uns anhand eines Beispiel-Graphen anschauen, wie der Dijkstra-Algorithmus die kürzesten Pfade bestimmt. Die Graphen zeigen, welche Pfade nacheinander „markiert“ werden. In jeder Sequenz ist B_i durch schwarze, R_i durch graue und U_i durch weiße Kreise symbolisiert, die besuchten Kanten sind fett, und die kleinen Zahlen neben den Knoten sind die D-Werte.



Satz: *Der Dijkstra–Algorithmus berechnet alle kürzesten Pfade von einer gegebenen Quelle aus im schlechtesten Fall in der Zeit $\mathcal{O}(n^2)$.*

Beweis: In der äußeren **while**–Schleife wird jeder Knoten $x \in V$ höchstens einmal betrachtet. Wir erhalten damit die folgende Formel für den Zeitaufwand, wobei $e = |E|$ und $n = |V|$:

$$t_{\text{Dijkstra}}(n) \in \mathcal{O}\left(\sum_{x \in V}\Big(|R| + \sum_{(x,y)\in E}\mathcal{O}(1)\Big)\right) \subseteq \mathcal{O}(n|R| + e) \subseteq \mathcal{O}(n^2 + e) = \mathcal{O}(n^2).$$

□

Die bisher durchgeführte Beschreibung und Analyse des Dijkstra–Algorithmus verwaltet den Rand R als Feld. Dies ist für sehr dichte Graphen optimal; in der Praxis sind jedoch viele Graphen dünn, z.B. ist die Zahl der Kanten e in planaren Graphen linear in der Knotenzahl n (genauer gilt die Eulerformel: $e \leq 3n - 6$ für $n \geq 3$).

Betrachtet man den Rand R als abstrakten Datentyp, so werden die folgenden Operationen benötigt:

insert	Füge ein neues Element in R ein.
decrease-key	Verringere den Schlüsselwert eines Elements von R (und erhalte die Eigenschaften des Datentyps R).
delete-min	Suche ein Element mit minimalem Schlüsselwert und entferne dieses aus R (und erhalte die Eigenschaften des Datentyps R).

In einer leicht abgewandelten Form lässt sich der Dijkstra–Algorithmus dann im Kern wie folgt beschreiben:

Algorithmus *Dijkstra–Algorithmus mit abstraktem Datentyp*

```
procedure dijkstra
begin
  B := ∅; R := {u}; U := V \ {u}; v(u) := nil; D(u) := 0;
  while (R ≠ ∅) do
    x := delete-min(R);
    B := B ∪ {x};
    forall (x, y) ∈ E do
      if y ∈ U then
        D(y) = D(x) + γ(x, y); v(y) := x; U := U \ {y};
        insert(R, y, D(y));                          (* y zum Rand hinzufügen *)
      elsif D(x) + γ(x, y) < D(y) then               (* die Bedingung impliziert y ∈ R *)
        D(y) := D(x) + γ(x, y); v(y) := x;
        decrease-key(R, y, D(y));                    (* Abstand von y verringern *)
      endif
    endfor
  endwhile
endprocedure
```

Eine einfache Analyse zeigt, dass die Operationen insert und delete-min höchstens n mal durchgeführt werden und decrease-key höchstens e mal aufgerufen wird.

Bemerkung:

1. *Der Dijkstra-Algorithmus liefert nicht notwendigerweise ein korrektes Ergebnis, falls für die Kantengewichte auch negative Zahlen erlaubt sind.*
2. *Für dünne Graphen ($\mathcal{O}(e) \subseteq o(n^2/\log n)$) ist es günstiger, den Rand R in einer Prioritätswarteschlange (Heap) zu verwalten. Der Aufwand des Algorithmus wird damit nach der obigen Herleitung $\mathcal{O}(e \log n + n \log n) \subseteq \mathcal{O}(e \log n)$.*
3. *Bei Verwendung der Datenstruktur der* Fibonacci-Heaps *ergibt sich ein Aufwand von $\mathcal{O}(e + n \log n)$.*

Im zweiten Teil dieses Abschnittes wollen wir ein weiteres Graphenproblem vorstellen, nämlich die Bestimmung von *minimal aufspannenden Bäumen*.

1.2.2 Minimale aufspannende Bäume (Prim-Algorithmus)

Definition: *Ein Graph* $G = (V, E)$ *heißt* zusammenhängend, *wenn je zwei Knoten durch einen Pfad verbunden sind. Ein* Baum *ist ein zusammenhängender, kreisfreier, ungerichteter Graph.*

Bemerkung: *Jeder Baum mit* n *Knoten besitzt genau* $n - 1$ *Kanten.*

Definition: *Ein* minimaler aufspannender Baum *(minimal spanning tree, MSB) zu einem gewichteten Graphen* $G = (V, E, \gamma)$ *ist ein Baum* $B = (V, F, \gamma|_F)$ *mit* $F \subseteq E$ *mit minimalem Gewicht* $\gamma(B) := \sum_{e \in F} \gamma(e)$*.*

Sei nun ein zusammenhängender gewichteter Graph $G = (V, E, \gamma)$ gegeben, wobei $\gamma : E \to \mathbb{N}$ die Kantengewichtsfunktion ist. Die Lösungsidee zur Bestimmung des MSB besteht darin, dass ein aufspannender Teilgraph $G' = (V, E', \gamma|_{E'})$ mit minimalem Gewicht $\gamma(G')$ ein gesuchter MSB ist, d.h. wir müssen G' nicht auf die Eigenschaft „Baum“ testen.

Algorithmus *Naiver Prim-Algorithmus für MSB's.*

---

function Prim ($G = (V, E, \gamma)$, G zusammenhängend, $|V| \geq 1$): MSB;
(* G ist ein ungerichteter, kantengewichteter Graph. *)
var
  B : Knotenmenge; (* Baumknoten *)
  T : Kantenmenge; (* Kantenmenge bezüglich B *)
  $x, u, v : V$;
begin
  wähle $x_0 \in V$ beliebig;
  $B := \{x_0\}$; $T := \emptyset$; (* Initialisierung *)
  while $B \neq V$ **do**
    wähle u, v mit $uv \in E$, $u \in B$ und $v \notin B$ so, dass $\gamma(uv)$ minimal ist;
    $B := B \cup \{v\}$;
    $T := T \cup \{uv\}$;
  endwhile
  return T;
endfunction

Der Zeitbedarf ist bei einer naiven Implementation $\mathcal{O}(|E| \cdot |V|) = \mathcal{O}(n^3)$ bei n Knoten, denn wir müssen für jeden Schleifendurchlauf die gesamte Kantenmenge E durchgehen.

Unter Beibehaltung der Grundidee können wir aber mit einem Dijkstra-ähnlichen Ansatz einen $\mathcal{O}(n^2)$ Algorithmus angeben.

Algorithmus *Prim-Algorithmus für minimal aufspannende Bäume*

---

function Prim($G = (V, E, \gamma)$, G zusammenhängend, $|V| \geq 1$): Kantenmenge;
var
  B, R, U : Knotenmenge; (* Baum-/Rand-/unbekannte Knoten *)
  T : Kantenmenge; (* Kantenmenge bezüglich B *)
  x, y : V;
  g : **integer**;
  v : **array** $[1..n]$ **of** V; (* $v(x)$ liefert Vorgänger von x *)
begin
  wähle $x_0 \in V$ beliebig;
  $B := \{x_0\}$; $T := \emptyset$; $R := \emptyset$; (* Initialisierung *)
  forall $x_0 y \in E$ **do**
    $R := R \cup \{y\}$; $v(y) := x_0$;
  endfor ;
  $U := (V \setminus \{x_0\}) \setminus R$;

```
x := x_0;
while R ≠ ∅ do
  forall xy ∈ E do
    if y ∈ U then                                   (* insert *)
      R := R ∪ {y}; v(y) := x; U := U \ {y};
    elsif y ∈ R and γ(xy) < γ(v(y)y) then            (* decrease-key *)
      v(y) := x;
    endif
  endfor ;
  x := nil ; g := ∞;                                (* begin delete-min *)
  forall y ∈ R do                                   (* suche minimale Kante von B nach R *)
    if γ(v(y)y) < g then
      x := y; g := γ(v(y)y);
    endif
  endfor ;
  R := R\{x}; B := B ∪ {x};                         (* end delete-min *)
  T := T ∪ {v(x)x}                                  (* erweitere den aufspannenden Baum *)
endwhile ;
return T
endfunction
```

Terminierung
Die Termination ist gesichert, denn in jedem Durchlauf wird B vergrößert. Nach Beendigung der Schleife gilt $B = V$.

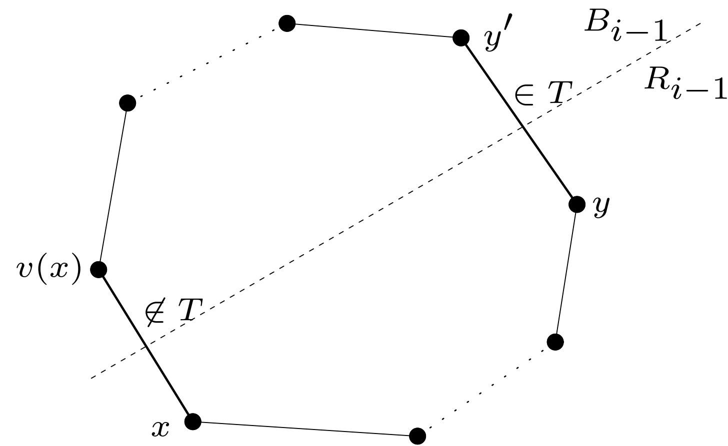
Korrektheit
Seien B_i, T_i, R_i die Mengen B, T, R nach dem i-ten Schleifendurchlauf. Dann lassen sich folgende Invarianten formulieren:

1. T_i lässt sich zu einem MSB von G ergänzen.
2. R_i enthält alle Knoten, die nicht zu B_i gehören, aber eine direkte Verbindung nach B_i haben.
3. Für alle $y \in R_i$ gilt: $\gamma(v(y)y) \leq \gamma(y'y)$, für alle $y' \in B_i$ mit $y'y \in E$.

Nach Voraussetzung ist $G = (V, E)$ zusammenhängend, daher genügt es, die Invarianten zu beweisen (denn aus (1) folgt, dass T_{n-1} ein MSB von G ist). Wir zeigen (1), (2), (3) mit Induktion.

Für $i = 0$ gilt $B_0 = \{x_0\}$, $T_0 = \emptyset$, $R_0 = \{y \in V \mid x_0 y \in E\}$ und die Invarianten sind trivialerweise erfüllt.

Sei $i \geq 1$ und die Invarianten für $i - 1$ erfüllt. Sei weiterhin $B_i = B_{i-1} \cup \{x\}$ und $T_i = T_{i-1} \cup \{v(x)x\}$. Sei $T = (V, F)$ ein MSB, der T_{i-1} ergänzt und die Kante $v(x)x$ nicht enthält (ansonsten ist (1) bereits erfüllt). Der Graph $T' = (V, F \cup \{v(x)x\})$ enthält nun einen Kreis und es existieren Knoten y, y' mit $y \in R_{i-1}$, $y' \in B_{i-1}$, so dass die Kante $yy' \in F$ zu diesem Kreis gehört:



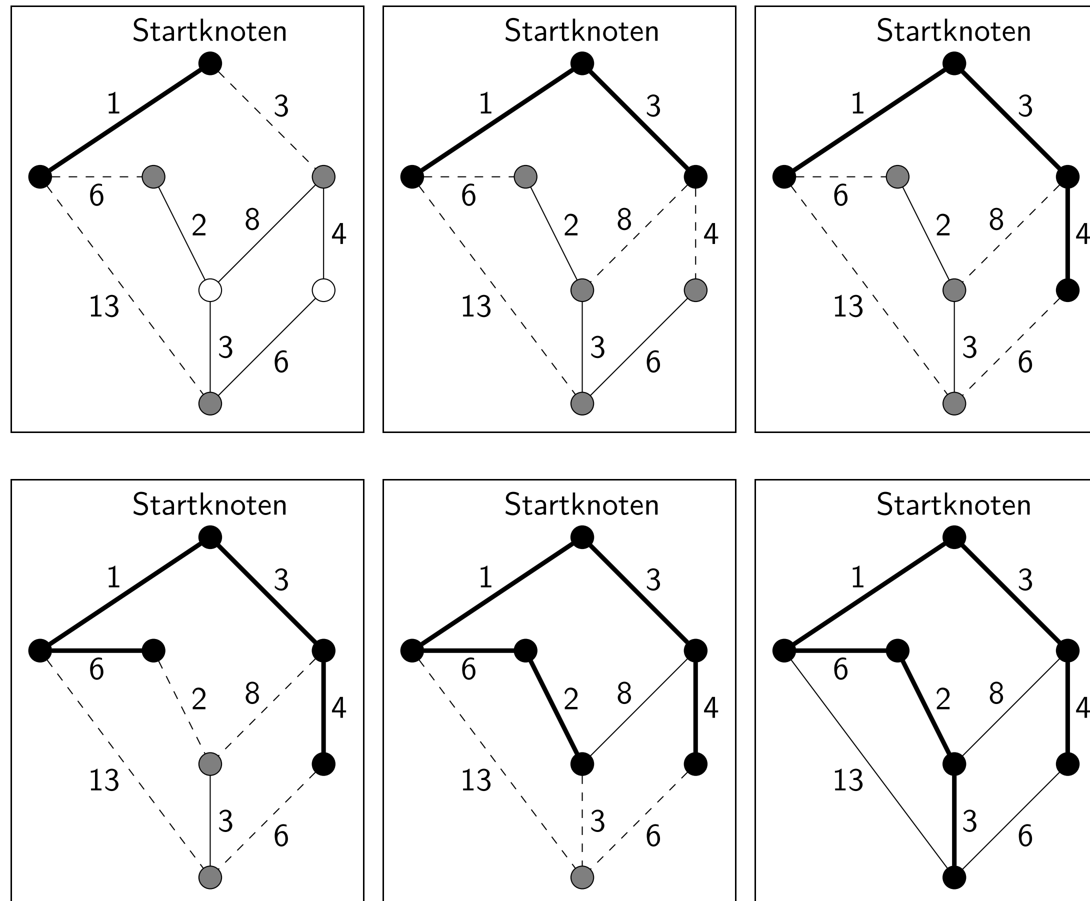
Weiterhin gilt nach Wahl von x bzw. wegen der letzten Bedingung der Invariante für $i-1$:

$$\gamma(v(x)x) \leq \gamma(v(y)y) \leq \gamma(y'y).$$

Tausche nun die Kanten $y'y$ und $v(x)x$ in T aus. Da dies nicht das Gesamtgewicht von T erhöht, erhalten wir erneut einen MSB für G und damit ist Bedingung (1) für i erfüllt. Die Bedingungen (2) und (3) ergeben sich direkt aus der Konstruktion.

Zur *Zeitanalyse* betrachten wir zunächst die erste forall-Schleife innerhalb der while-Schleife. Während des ganzen Algorithmus kann hier jede Kante xy maximal zweimal betrachtet werden, nämlich jeweils als ausgehende Kante ihrer zwei Knoten. Dies ergibt einen Gesamtaufwand von $\mathcal{O}(e) \subseteq \mathcal{O}(n^2)$, wobei $e = |E|$ und $n = |V|$.
Der Aufwand der zweiten forall-Schleife ist bei jedem Durchlauf durch $\mathcal{O}(n)$ begrenzt. Es gibt n Durchläufe durch die while-Schleife und somit erhalten wir einen Zeitbedarf von $\mathcal{O}(e + n^2) \subseteq \mathcal{O}(n^2)$.

Beispiel: Zur Veranschaulichung des Prim-Algorithmus verwenden wir erneut den Graph aus Beispiel 1. Die zu T_i gehörenden Kanten sind fett, die an T_i angrenzenden Kanten (zwischen B_i und R_i) gestrichelt dargestellt. Wie im Beispiel zum Dijkstra-Algorithmus sind die Knoten aus B_i schwarz, die aus R_i grau und die aus U_i weiß gefärbt.



Bemerkung:

1. *Der Prim Algorithmus läuft auch mit negativen Gewichten korrekt. Dies folgt aus der Tatsache, dass Gewichte um eine additive Konstante verschoben werden können, ohne dabei den MSB zu verändern: Sei* $G = (V, E, \gamma)$ *mit* $\gamma : E \rightarrow \mathbb{R}$ *ein gewichteter Graph und* $c \in \mathbb{R}$ *eine Konstante. Betrachte die neue Gewichtsfunktion* $\tilde{\gamma} : E \rightarrow \mathbb{R}$, $\tilde{\gamma}(e) := \gamma(e) + c$. *Nun gilt: Ist* T *ein MSB für* G, *so auch* T *für* $\tilde{G} = (V, E, \tilde{\gamma})$. *Diese Behauptung folgt leicht mit der Eigenschaft, dass jeder MSB für einen Graphen mit* n *Knoten* $n - 1$ *Kanten besitzt.*
2. *Für dünne Graphen (*$\mathcal{O}(e) \subseteq o(n^2/\log n)$*) ist es günstiger, die Menge* $\{v(y)y) \mid y \in R\}$ *in einer Prioritätswarteschlange (Heap) zu verwalten. Dies führt auf einen* $\mathcal{O}(e \log n)$ *Algorithmus.*
3. *Bei Verwendung der Datenstruktur der* Fibonacci-Heaps *ergibt sich ein Aufwand von* $\mathcal{O}(e + n \log n)$.

1.3 Dynamische Programmierung

Beim Verfahren der *dynamischen Programmierung* werden tabellarisch alle Teillösungen eines Problems bestimmt, bis schließlich die Gesamtlösung erreicht ist. Die Teillösungen werden dabei mit Hilfe der bereits existierenden Einträge berechnet.

1.3.1 Transitive Hülle und kürzeste Wege in Graphen

Als Beispiel betrachten wir den *Warshall-Algorithmus* zur Bestimmung der transitiven Hülle und den *Floyd-Algorithmus* zur Bestimmung der kürzesten Wege.

Algorithmus *Warshall-Algorithmus zur Berechnung der transitiven Hülle*

Eingabe : Graph als Adjazenzmatrix $(A[i,j]) \in \text{Bool}_{n\times n}$

```
procedure Warshall (var A : Adjazenzmatrix)
begin
  for k := 1 to n do
    for i := 1 to n do
      for j := 1 to n do
        if (A[i,k] = 1) and (A[k,j] = 1) then
          A[i,j] := 1
        endif
      endfor
    endfor
  endfor
end
```

Die Korrektheit des Warshall-Algorithmus folgt aus der Invariante:

1. Nach dem k-ten Durchlauf der ersten **for**-Schleife gilt $A[i, j] = 1$, falls ein Pfad von i nach j nur über Knoten mit Nummern $\leq k$ existiert (vgl. mit der Bestimmung rationaler Ausdrücke aus einem endlichen Automaten).
2. Gilt $A[i, j] = 1$, so existiert ein Pfad von i nach j.

Trägt man in die Adjazenz-Matrix Gewichte statt Boolesche Werte ein, so entsteht der Floyd-Algorithmus zur Berechnung kürzester Wege:

Algorithmus *Floyd-Algorithmus zur Bestimmung aller kürzesten Wege in einem Graphen*

Eingabe : Gewichteter Graph als Adjazenzmatrix $A[i, j] \in (\mathbb{N} \cup \infty)_{n \times n}$,
wobei $A[i, j] = \infty$ bedeutet, dass es keine Kante von i nach j gibt.
procedure Floyd (**var** A : Adjazenzmatrix)
begin
for $k := 1$ **to** n **do for** $i := 1$ **to** n **do for** $j := 1$ **to** n **do**
$A[i, j] := \min\{A[i, j], A[i, k] + A[k, j]\}$;
endfor endfor endfor
endprocedure

Bemerkung: *Der Floyd-Algorithmus liefert auch dann ein korrektes Ergebnis, wenn die Gewichte negativ sind, unter der Bedingung, dass keine negative Schleifen vorhanden sind.* Sowohl der Warshall- als auch der Floyd-Algorithmus haben einen Zeitaufwand von $\Theta(n^3)$. Eine „Verbesserung“ kann dadurch erzielt werden, dass vor der j-Schleife zuerst getestet wird, ob $A[i, k] = 1$ (bzw. ob $A[i, k] < \infty$) gilt. Damit erreicht man den Aufwand $\mathcal{O}(n^3)$.

1.3.2 Multiplikation einer Matrizenfolge

Ein weiteres Beispiel für dynamische Programmierung ist die optimale Klammerung bei der Multiplikation einer Matrizenfolge (mit verschiedenen Dimensionen).

Beispiel: Sei $A \cdot B \cdot C$ zu berechnen, wobei A eine 4×100, B eine 100×8 und C eine 8×2 Matrix ist. In welcher Reihenfolge multipliziert man diese Matrizen so, dass man möglichst wenige skalare Multiplikationen hat? Berechnet man zuerst $X := A \cdot B$ und dann $Y := X \cdot C$ so braucht man 3200 Multiplikationen um X zu berechnen und 64 für Y. Berechnet man aber zuerst $X' := B \cdot C$ und dann $Y' := A \cdot X'$, dann braucht man insgesamt nur $100 \cdot 8 \cdot 2 + 4 \cdot 100 \cdot 2 = 2400$ Multiplikationen.

Bemerkung: *$A_{(n,m)}$ ist die Bezeichnung für eine Matrix A mit n Zeilen und m Spalten. Für $A_{(n,m)} := B_{(n,q)} \cdot C_{(q,m)}$ benötigt man $n \cdot q \cdot m$ skalare Multiplikationen.*

Sei die Matrizenfolge $M^1_{(n_0,n_1)}, M^2_{(n_1,n_2)}, M^3_{(n_2,n_3)}, \ldots, M^N_{(n_{N-1},n_N)}$ gegeben. Weiterhin definieren wir die Funktion cost($M^1, \ldots, M^N$) als minimale Anzahl der skalaren Multiplikationen, die für das Produkt $\prod_{i=1}^{N} M^i$ benötigt werden.

Der Ansatz der dynamischen Programmierung ergibt sich nun mit der folgenden Beziehung:

$$\text{cost}(M^i, \ldots, M^j) =$$
$$\text{min}_k\{\text{cost}(M^i, \ldots, M^k) + \text{cost}(M^{k+1}, \ldots, M^j) + n_{i-1} \cdot n_k \cdot n_j\}$$

Für den folgenden Algorithmus wird in einer Tabelle cost$[i, j]$ berechnet, wobei $j - i$ die Werte $2, 3, 4, \ldots, n - 1$ annimmt. Die Hilfstabelle best$[i, j]$ wird jeweils einen Index $i \leq k < j$ beinhalten, der eine bzgl. der Anzahl der Multiplikationen optimale Aufteilung des Produktes $M^i \cdot \ldots \cdot M^j$ ergibt.

Mit diesem Algorithmus (sh. nächste Seite) haben wir einen Platzaufwand von $\Theta(N^2)$ und einen Zeitaufwand von $\Theta(N^3)$.

Eine wichtige Anwendung der dynamischen Programmierung findet man ebenfalls bei dem Algorithmus von *Cocke-Younger-Kasami* (*CYK-Algorithmus*) zur Lösung des Wortproblems für kontextfreie Sprachen (bekannt aus TI 1).

Algorithmus *Effiziente Multiplikation einer Matrizenfolge*

```
procedure mult-matrizenfolge()
begin
  for i := 1 to N do                                  (* Initialisierung der Kostentabelle *)
    cost[i, i] := 0;
    for j := i + 1 to N do
      cost[i, j] := ∞;
    endfor
  endfor ;
  (* Kostentabelle und best-Tabelle erzeugen *)
  for d := 1 to N − 1 do                              (* d ist die Differenz j − i *)
    for i := 1 to N − d do
      j := i + d;
      for k := i to j − 1 do
        t := cost[i, k]+ cost[k + 1, j] + n[i − 1] · n[k] · n[j];
        if t < cost[i, j] then
          cost[i, j] := t;
          best[i, j] := k;
        endif
      endfor
    endfor
  endfor ;
  return best;
endprocedure
```

1.4 Backtracking

Sucht man in einem Baum, so ist zum Beispiel Tiefensuche ein effizientes Verfahren, wenn sich frühzeitig entscheidet, ob in einem Teilbaum eine Lösung zu finden ist oder nicht. *Backtracking* (dt. „Zurückgehen“) beschreibt ein solches Verfahren, bei dem man im Berechnungsgraphen eines Problems solange vorwärts geht, bis ein Knoten erreicht wird, der eine Lösung darstellt oder bei dem sicher davon ausgegangen werden kann, dass von diesem Knoten aus keine Lösung mehr zu finden ist. Ist das letztere der Fall, dann geht man den Pfad einen Schritt zurück und entscheidet sich für einen anderen Folgepfad usw.

Backtracking findet oft Anwendung bei Problemen, für die nur exponentielle Algorithmen bekannt sind. Wir betrachten hier das sogenannte *Mautstraßenproblem* (engl. *Turnpike-Problem*).

1.4.1 Das Mautproblem

Beispiel: Fährt man in Frankreich über die Autobahn nach Paris, so muss Maut gezahlt werden. Man bekommt hierzu eine Karte, die eine Tabelle mit den Entfernungen und Gebühren zwischen den Ein- und Ausfahrten enthält.
Die Frage ist nun: lässt sich die Lage der Ausfahrten aus den Entfernungen rekonstruieren, wenn nur eine geordnete Liste der Entfernungen mit ihren Vielfachheiten vorliegt?

Wir suchen also eine Lösung für folgendes Problem:

Es seien $0 = x_0 < x_1 < \cdots < x_n$ positive Zahlenwerte und A eine $n \times n$-Matrix mit $A_{i,j} = |x_i - x_j|$. Weiterhin sei D ein Feld, das die Werte $A_{i,j}$, $1 \leq i < j \leq n$, in sortierter Reihenfolge enthält. Gesucht sind die Werte x_k für $1 \leq k \leq n$, wenn D gegeben ist.

Bemerkung:

1. *Sind die Werte x_k bekannt, dann kann A (bzw. D) in $\mathcal{O}(n^2)$ (bzw. in $\mathcal{O}(n^2 \log n)$) Schritten berechnet werden.*
2. *Sei D bekannt. Ist die Folge $x_0, \ldots, x_n$ eindeutig bestimmt, falls sie existiert? Wie komplex ist dann die Rekonstruktion?*

Die erste Frage kann mit Nein beantwortet werden. Selbst dann existiert keine eindeutige Lösung, wenn in D alle Einträge verschieden sind (bis jetzt wurde aber kein Beispiel für $n > 6$ gefunden).

Algorithmus *Mautproblem*

Die Operationen *member*, *delete* und *insert* werden auf der *Multimenge* D ausgeführt. Zur Verdeutlichung verwenden wir die Zeichen $+$ für insert und $-$ für delete.

type XFeld = **array** $[0..n]$ **of real**;
  DFeld = geordnete Liste $[0..n(n+1) \operatorname{div} 2]$ **of real**;
procedure maut(**var** X: XFeld; **var** D: DFeld; n: **integer**; **var** $found$: **boolean**);
(* Ist eine Platzierung möglich, so steht das Ergebnis in X und $found$ ist true. *)
begin
  (* Initialisierung, linker und rechter Rand wird festgelegt – $\max(D)$ sei vordefiniert als Maximumsfunktion *)
  $found :=$ **false**;
  $X_0 := 0;$
  $X_n := \max(D);$
  $D := D - \max(D);$ (* D aktualisieren *)
  (* X_1 kann wegen Symmetrie so gewählt werden *)
  $X_1 := X_n - \max(D);$
  if $(X_1 - X_0) \in D$ **then**
    $D := D - \{|X_1 - X_0|, |X_n - X_1|\};$
    place$(X, D, n, 1, n, found)$
  endif
endprocedure

Algorithmus *Mautproblem Teil 2 (Prozedur* place())

```
procedure place(var X: XFeld; var D: DFeld; n, l, r: integer; var found: boolean);
(* X_0, ..., X_l und X_r, ..., X_n sind beim Aufruf von Place schon versuchsweise festgelegt *)
var d: real;
    D': DFeld;
begin
  if D = ∅ then found := true
  else
    d := max(D);
    (* probiere X_{l+1} := X_n − d *)
    D' := {|X_j − (X_n − d)| | j ∈ {0, ..., l} ∪ {r, ..., n}};
    if D' ⊆ D then
      X_{l+1} := X_n − d;
      D := D − D';                              (* D aktualisieren *)
      place(X, D, n, l + 1, r, found);          (* rekursiv Lösung suchen *)
      if found = false then                     (* Falls Misserfolg: Backtracking *)
        D := D + D'                             (* D wiederherstellen *)
      endif
    endif
```

```
      (* probiere X_{r-1} := d *)
      D' := {|X_j - d| | j ∈ {0, ..., l} ∪ {r, ..., n}};
      if (found = false) and D' ⊆ D then
         X_{r-1} := d;
         D := D - D';                                     (* D aktualisieren *)
         place(X, D, n, l, r - 1, found);                 (* rekursiv Lösung suchen *)
         if found = false then                            (* Falls Misserfolg: Backtracking *)
            D := D + D'                                   (* D wiederherstellen *)
         endif
      endif
   endif
endprocedure
```

Zeitanalyse zum maut-Algorithmus: Die Datenstruktur für D sollte die Operationen $\max(D)$, member, delete, insert effizient erlauben d.h. in $\mathcal{O}(\log(n))$ $(= \mathcal{O}(\log(n^2)))$ Schritten. Dies wird etwa durch eine geordnete Liste ermöglicht, bei der die delete-Funktion nur die Plätze markiert, die insert-Funktion (wir fügen nur Werte ein, die wir vorher gelöscht haben) hebt die Markierung von delete wieder auf. Member ist einfach eine binäre Suche, und $\max(D)$ ist der Wert des Feldes mit dem höchsten nicht-markierten Index.

Der worst-case ist exponentiell und tritt dann ein, wenn sehr viel zurückgesetzt werden muss. Beispiele hierfür sind bekannt es zeigt sich aber (empirisch), dass dieser Fall selten auftritt.

1.5 Zusammenfassung

Als erstes haben wir die *Divide and Conquer*-Methode betrachtet. Typische Vertreter dieser Algorithmengruppe sind Mergesort und Quicksort. Anschließend haben wir den Dijkstra- und den Prim-Algorithmus als Beispiele für die *Greedy*-Strategie kennengelernt. Die Methode der *dynamischen Programmierung* wurde anhand des Warshall-Algorithmus und des Algorithmus zur Matrizenmultiplikation eingeführt. Schließlich haben wir noch *Backtracking* betrachtet, und als Beispiel das Mautproblem algorithmisch gelöst. Nicht betrachtet haben wir die Gruppe der *randomisierten* Algorithmen.

2 Sortieren und Medianberechnung

Wir befassen uns jetzt mit der Analyse von Sortieralgorithmen. Dabei werden die grundsätzlichen Sortierverfahren als bekannt vorausgesetzt. Wir werden hier nur einige Teilaspekte, wie die Herleitung des „Average case"-Aufwands von Quicksort und das *Bottom-Up-Heapsort* behandeln.

2.1 Quicksort

Der *Quicksort*-Algorithmus wurde 1962 von Hoare vorgestellt. Das grundlegende Prinzip besteht darin, ein Pivotelement zu bestimmen und das Feld in zwei Teilfelder zu zerlegen, wobei das Pivotelement als Trennungsmarke der beiden Felder dient. Die Elemente der Felder werden dann so vertauscht, dass im ersten Feld nur die Elemente enthalten sind, die kleiner (oder gleich) als das Pivotelement sind, während das zweite Feld die übrigen Elemente enthält. Schließlich wendet man auf beide Teilfelder rekursiv den gleichen Algorithmus an.

Für die Aufwandsabschätzung benötigt man zusätzliche Informationen, wobei der wichtigste Aspekt die Wahl des Pivots ist. Die beste Laufzeit ergibt sich natürlich, falls das Pivot gleich dem mittleren Element des Feldes (Median) ist. In der Praxis hat sich eine einfachere Methode bewährt, die *Median-aus-Drei*-Methode. Diese benutzt den Median des ersten, letzten und mittleren Elementes als Pivot. Diese Version ist im folgenden dargestellt.

Wir geben zuerst die Prozedur zum Partitionieren eines Feldes A (im Bereich ℓ bis r) bzgl. eines Pivot-Elements $P = A[p]$ an, wobei $\ell < r$ und $p \in \{\ell, \ldots, r\}$ sei. Ergebnis ist ein Index $m \in \{\ell, \ldots, r-1\}$ mit folgenden Eigenschaften:

$$A[i] \leq P \text{ für alle } \ell \leq i \leq m \qquad \text{und} \qquad A[i] \geq P \text{ für alle } m+1 \leq i \leq r.$$

Man beachte, dass das rechte Teilfeld nach dem Partitionieren nicht leer sein kann.

Man vergewissere sich, dass die geforderten Eigenschaften von dieser Implementierung wirklich erfüllt werden, und dass es außerdem nicht vorkommen kann, dass der linke bzw. rechte „Zeiger“ (x bzw. y) über die Feldgrenze ℓ bzw. r hinauswandert.

Die vorgestellte Implementierung führt in der Regel $n + 2$ Vergleiche auf einem Feld mit n Elementen durch. Bei der Durchschnittsanalyse von Quicksort (und von Quickselect in Abschnitt 2.4) werden wir jedoch von $n - 1$ Vergleichen ausgehen — dies lässt sich durch eine trickreichere Implementierung erreichen (jedes Element außer $A[p]$ muss genau einmal mit P verglichen werden).

Algorithmus *Partitionieren bzgl. eines Pivot-Elements*

---

function partitioniere($A[\ell \dots r]$: **array of integer**, p : **integer**) : **integer**
(* Partitioniere $A[\ell \dots r]$ bzgl. $A[p]$; Rückgabewert = Index m *)
begin
  $P := A[p]$; (* Pivot-Element merken *)
  swap($A[\ell]$, $A[p]$); (* Pivot an erste Pos. stellen *)
  $x := \ell - 1$;
  $y := r + 1$;
  while $x < y$ **do**
    repeat $x := x + 1$ **until** $A[x] \geq P$;
    repeat $y := y - 1$ **until** $A[y] \leq P$;
    if $x < y$ **then**
      swap($A[x]$, $A[y]$);
    endif
  endwhile
  return y
endfunction

Mit Hilfe der Partitionier-Funktion lässt sich Quicksort nun leicht aufschreiben.

Algorithmus *Quicksort*

---

procedure quicksort($A[\ell \ldots r]$: **array of integer**)
begin
  if $\ell < r$ **then**
    p := Index des Median von $A[\ell]$, $A[(\ell + r)$ div $2]$, $A[r]$;
    m := partitioniere($A[\ell \ldots r]$, p); (* Feld bzgl. $A[p]$ partitionieren *)
    quicksort($A[\ell \ldots m]$); (* linkes Teilfeld sortieren *)
    quicksort($A[m + 1 \ldots r]$); (* rechtes Teilfeld sortieren *)
  endif
endprocedure

Der ungünstigste Fall des Quicksort-Algorithmus ist quadratisch und tritt dann ein, wenn in jedem Schritt eines der beiden Teilfelder genau ein Element enthält.

Wir wollen hier den average-case für den Quicksort-Algorithmus untersuchen. Dabei bezieht sich unsere Durchschnittsanalyse auf eine zufällige Auswahl des Pivotelements.

Sei $Q(n)$ die mittlere Anzahl der benötigten Vergleiche bei n Elementen und $H(n) := \sum_{k=1}^{n} \frac{1}{k}$ die $n-$te harmonische Zahl.

Satz: *Die mittlere Anzahl der benötigten Vergleiche von Quicksort ist:*

$$Q(n) = 2(n+1)H(n) - 4n.$$

Beweis: Für $n = 1$ gilt offensichtlich $Q(1) = 0 = 2 \cdot 2 \cdot 1 - 4 \cdot 1$. Für $n \geq 2$ gilt:

$$\begin{aligned} Q(n) &= (n-1) + \frac{1}{n}\sum_{i=1}^{n}[Q(i-1) + Q(n-i)] \\ &= (n-1) + \frac{2}{n}\sum_{i=1}^{n} Q(i-1) \end{aligned}$$

Dabei ist $(n-1)$ die Zahl der Vergleiche beim Pivotieren und $[Q(i-1)+Q(n-i)]$ die mittlere Zahl der Vergleiche für das rekursive Sortieren der beiden Teilhälften; dabei sind alle Positionen für das Pivotelement gleich wahrscheinlich (deswegen der Faktor $1/n$).

Es gilt:

$$\begin{aligned}
nQ(n) - (n-1)Q(n-1) &= n(n-1) + 2\sum_{i=1}^{n} Q(i-1) \\
&\quad -(n-1)(n-2) - 2\sum_{i=1}^{n-1} Q(i-1) \\
&= n(n-1) - (n-1)(n-2) + 2Q(n-1) \\
&= 2(n-1) + 2Q(n-1)
\end{aligned}$$

und damit

$$\begin{aligned}
nQ(n) &= 2(n-1) + 2Q(n-1) + (n-1)Q(n-1) \\
&= 2(n-1) + (n+1)Q(n-1)
\end{aligned}$$

Weiter gilt:

$$\begin{aligned}
\frac{Q(n)}{n+1} &= \frac{2(n-1)}{n(n+1)} + \frac{Q(n-1)}{n} = \frac{2(n-1)}{n(n+1)} + \frac{2(n-2)}{(n-1)n} + \frac{Q(n-2)}{n-1} \\
&= \sum_{k=2}^{n} \frac{2(k-1)}{k(k+1)} \\
&= 2\sum_{k=2}^{n} \left[\frac{2}{k+1} - \frac{1}{k}\right] \\
&= 2\left[2\left(H(n) + \frac{1}{n+1} - 1 - \frac{1}{2}\right) - (H(n) - 1)\right] \\
&= 2H(n) + \frac{4}{n+1} - 4.
\end{aligned}$$

Schließlich erhält man für $Q(n)$:

$$Q(n) = 2(n+1)H(n) + 4 - 4(n+1) = 2(n+1)H(n) - 4n.$$

□

Für große n ist $H(n) - \ln n \approx 0.57\ldots$. Damit ergibt sich:

$$Q(n) \quad \approx \quad 1.38n \log n - 2.8n.$$

Man beachte dabei, dass die theoretische Grenze bei $\log(n!) = n \log n - 1.44n$ Vergleichen liegt und somit der Quicksort-Algorithmus im Mittel um 38% schlechter ist.

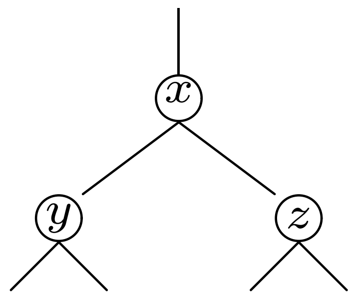
Die Durchschnittsanalyse der Median-aus-Drei Methode ist aufwändig und wird hier nicht durchgeführt. Der Durchschnittswert liegt dann jedoch bei $1.18n \log n - 2.2n$. Damit ist dann der Spielraum für Verbesserungen sehr eng geworden.

2.2 Bottom-Up-Heapsort

Definition: *Ein (Min-)*Heap *ist ein Feld* $a[1 \ldots n]$ *mit* $a[i] \leq \min\{a[2i], a[2i+1]\}$ *für alle* $i \leq n/2$. *(Wenn* n *gerade ist, muss für* $i = n/2$ *natürlich nur* $a[n/2] \leq a[n]$ *gelten, da* $a[n+1]$ *nicht definiert ist.)*

Der übliche Heapsort-Algorithmus besteht aus zwei Teilen:

1. Der Heapaufbau, der in $\mathcal{O}(n)$ Schritten erfolgen kann. Der Heap wird dabei von rechts nach links aufgebaut, wodurch viele kleine und nur wenige große Heaps betrachtet werden.
2. $a[1]$ ist das kleinste (bzw. größte) Element. Vertausche nun $a[1]$ und $a[n]$. Die Heapbedingung in $a[1 \ldots n-1]$ ist jetzt in $a[1]$ eventuell verletzt. Lasse das zu schwere Element einsinken:



Man benötigt zwei Vergleiche, um das Minimum von $\{x, y, z\}$ zu bestimmen.
Ist y (bzw. z) das Minimum, dann vertausche x und y (bzw. x und z). Ist x das Minimum, dann stoppt der Einsinkprozess.

Der Kern des Algorithmus ist folgende Prozedur *reheap*, wobei der in der Definition beschriebene Sonderfall für gerade n und $i = n/2$ vernachlässigt wird. In einer realen Implementierung müsste darauf natürlich geachtet werden:

Algorithmus *Reheap*

---

procedure reheap(i, n: **integer**)
(* $i \leq n$ ist die Wurzel des betrachteten Teilbaums *)
var m: integer;
begin
  if $i \leq n/2$ **then**
    $m := \min\{a[i], a[2i], a[2i+1]\}$;
    if $(m \neq a[i]) \wedge (m = a[2i])$ **then**
      swap($i, 2i$); (* vertausche x, y *)
      reheap($2i, n$)
    elsif $(m \neq a[i]) \wedge (m = a[2i+1])$ **then**
      swap($i, 2i+1$); (* vertausche x, z *)
      reheap($2i+1, n$)
    endif
  endif
endprocedure

Der *Heap-Aufbau* erfolgt dann mit

$$\textbf{for } i := \left\lfloor \frac{n}{2} \right\rfloor \textbf{ downto } 1 \textbf{ do } \text{reheap}(i, n) \textbf{ endfor}$$

Die Invariante hierfür ist: $a[i \ldots n]$ erfüllt bereits die obige Heap-Bedingung (d.h. vor dem Aufruf reheap$(i-1, n)$).

Für $i = \left\lfloor \frac{n}{2} \right\rfloor + 1$ ist dies richtig.

Setze i um 1 herab, dann ist die Heapbedingung in $a[i]$ eventuell verletzt. Einsinken kostet im schlechtesten Fall 2 · (Höhe des Teilbaums unter $a[i]$) Vergleiche. Wir führen die Analyse für $n = 2^k - 1$ durch, d.h. die maximale Höhe des Heaps ist $k - 1$. Allgemein gibt es

- 2^0 Bäume der Höhe $k - 1$,
- 2^i Bäume der Höhe $k - 1 - i$,
- 2^{k-1} Bäume der Höhe 0.

Daher sind zum Heapaufbau maximal

$$2 \cdot \sum_{i=0}^{k-1} 2^i (k - 1 - i)$$

Vergleiche nötig.

Sei $V(k) = \sum_{i=0}^{k-1} 2^i(k-1-i) = \sum_{i=1}^{k} 2^{i-1}(k-i)$. Wir zeigen, dass $V(k) = 2^k - k - 1$ gilt.

Für $k = 1$ ist $V(1) = 0$ korrekt. Für $k > 1$ gilt:

$$\begin{aligned} V(k) &= \sum_{i=1}^{k} 2^{i-1}(k-i) = \sum_{i=1}^{k-1} 2^{i-1} + \sum_{i=1}^{k-1} 2^{i-1}((k-1)-i) \\ &= 2^{k-1} - 1 + V(k-1) \\ &\underset{\text{induktiv}}{=} 2^{k-1} - 1 + 2^{k-1} - (k-1) - 1 = 2^k - k - 1. \end{aligned}$$

Dies ergibt den folgenden Satz:

Satz: *Heapaufbau ist in linearer Zeit möglich.*

Beweis: Für $n = 2^k - 1$ gilt: $2V(k) = 2(n - \log(n+1)) \in \mathcal{O}(n)$. □

Das eigentliche *Sortieren* findet dann statt mit:

Algorithmus *Heap-Sort*

---

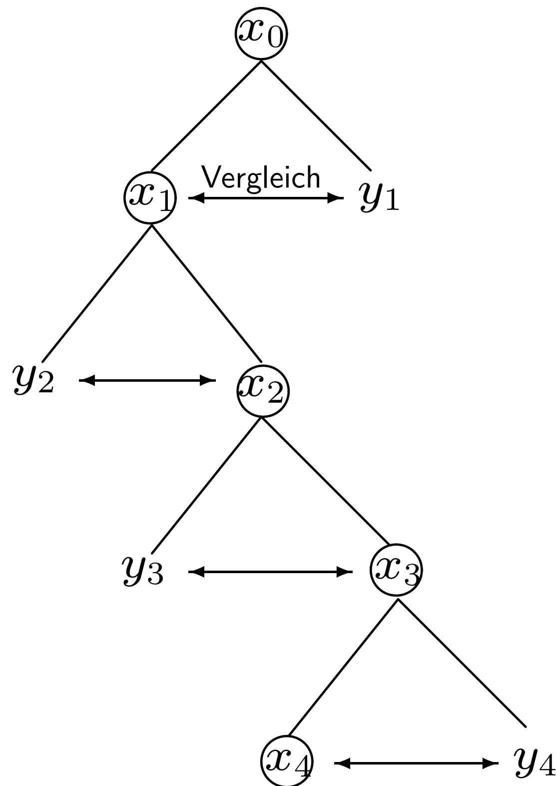
procedure heapsort(n: **integer**)
begin
  for $i := n$ **downto** 2 **do**
    swap($1, i$);
    reheap($1, i - 1$)
  endfor
endprocedure

Satz: *Standard Heapsort erfordert höchstens* $2n \log n + \mathcal{O}(n)$ *Vergleiche.*

Beweis: Der Aufbau des Heaps erfordert $\mathcal{O}(n)$ und der Abbau durch Einsinken $2n \log n$. □

Bemerkung: *Eine genaue Analyse der Durchschnittskomplexität von Heapsort ist unter Verwendung von Methoden aus der sogenannten Kolmogorov-Komplexität möglich. Es ergibt sich ein mittlerer Aufwand von* $2n \log n$ *Vergleichen. Damit ist Standard-Heapsort zu Quicksort nicht konkurrenzfähig.*

Bottom-Up-Heapsort ist eine Variante, mit der die Konstante vor dem $n \log(n)$ zu 1 verbessert werden soll. Dabei geht man folgendermaßen vor: nach dem Entfernen der Wurzel wird zuerst der *potentielle* Einsinkpfad des Elementes bestimmt (siehe Abbildung unten), das die Wurzel ersetzen soll. Dies geschieht, indem man von der Wurzel aus den Weg verfolgt, der immer zum kleineren der beiden Nachfolger führt (Kosten insgesamt: $n \log n$ Vergleiche). Da erwartungsgemäß dieses Element tief einsinken wird (es war vorher ein Blatt), bietet sich anschließend eine bottom-up Bestimmung der tatsächlichen Position auf dem Einsinkpfad an (in der Hoffnung, insgesamt mit $\mathcal{O}(n)$ Vergleichen auskommen zu können).



Es gilt $x_i \leq y_i$ für $1 \leq i \leq$ Höhe.

Das Einsinken geschieht längs dieses Pfades,
der mit $\log n$ Vergleichen bestimmt werden kann.

Ein tiefes Einsinken des Blattes, dessen potentieller Einsinkpfad bestimmt wurde, ist auch aufgrund folgender Eigenschaft des Heaps plausibel: im Heap ist die Hälfte der Knoten Blätter, $\frac{3}{4}$ haben höchstens Höhe 1, $\frac{7}{8}$ höchstens Höhe 2, usw..

Die erwartete Höhe eines Knotens in einem Binärbaum mit zufälliger Knotenverteilung ist daher:

$$0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} + 2 \cdot \frac{1}{8} + \ldots = \frac{1}{2} \sum_{n=0}^{\infty} \frac{n}{2^n} = 1$$

Hierbei entspricht der Höhe 0 die Ebene der Blätter. Im Heap sind die Knoten nicht zufällig verteilt, die mittlere Einsinkhöhe sollte jedoch nicht größer sein, da ja $a[n]$ schwer ist.

Wir bestimmen jetzt vom Blatt aus (also bottom-up) die tatsächliche Position auf dem Einsinkpfad. Sei $x_0, x_1, \ldots, x_k$ der berechnete Pfad. Die Prozedur

$i := k + 1;$
repeat
    $i := i - 1$
until $x_0 \geq x_i$

berechnet die Position x_i, auf die x_0 einsinkt. Ist der Index i gefunden, so werden die Elemente $x_0, \ldots, x_i$ zyklisch vertauscht (x_0 geht an die Stelle von x_i, und $x_1, \ldots, x_i$ rutschen hoch).

Es kann gezeigt werden, dass im schlechtesten Fall höchstens $1.5n \log n + o(n \log n)$ Vergleiche benötigt werden.

Wenn man auf dem Pfad eine binäre Suche anwendet, so kommt man auf einen Aufwand von höchstens $n \log n + \mathcal{O}(n \log \log n)$. Eine binäre Suche ist aber in der Praxis zu aufwändig und außerdem steigt man in den Pfad i.a. zu hoch ein.

Durch eine leichte Abwandlung der reinen bottom-up Positionsbestimmung können wir die Zahl der wesentlichen Vergleiche auf $n \log n + o(n \log n)$ bringen.

Ebenfalls möglich ist eine Schrittweitenverdoppelung bei der bottom-up Suche: man geht beim Aufsteigen im Pfad solange in Schrittweiten $1, 2, 4, 8, \ldots$ hoch, bis ein Knoten erreicht ist, der zu hoch liegt. Man geht dann zu dem vorher betrachteten Knoten zurück und fängt wieder mit Schrittweite 1 zu suchen an.

Die Zahl der wesentlichen Vergleiche ist hier $n \log n + \mathcal{O}(n \log^2(\log n))$.

Beispiel: In der folgenden Tabelle ist die Schrittanzahl in Abhängigkeit vom Level für einen Heap mit 15 Levels dargestellt (dabei gehören die Blätter dem Level 0 an).

Levelnummer	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Schritte zum Ziel	1	3	2	4	6	5	3	5	7	6	8	10	9	7	4
Zeile 2 - Zeile 1	0	1	-1	0	1	-1	-4	-3	-2	-4	-3	-2	-4	-7	-11

2.3 Medianberechnung in linearer Zeit

Gegeben sei ein Feld $a[1 \ldots n]$ von Zahlen. Gesucht ist für ein $1 \leq k \leq n$ das k-kleinste Element m, d.h. die Zahl $m \in \{a[i] \mid 1 \leq i \leq n\}$ so, dass

$$|\{i \mid a[i] < m\}| < k \quad \text{und} \quad |\{i \mid a[i] > m\}| \leq n - k$$

Die folgende Prozedur berechnet rekursiv den Wert m in linearer Zeit. Sei $n \geq 54$.

1. *Bestimme ein Pivotelement als Median der Mediane aus 5:*
   Wir teilen das Feld in Fünferblöcken auf. In jedem Block wird der Median bestimmt (mit 6 Vergleichen möglich). Wir bestimmen rekursiv den Median p dieses Feldes (mit dem gesamten Algorithmus). Der Wert p wird als Pivotelement im folgenden verwendet.
   Kosten: $T(\frac{n}{5})$.
2. *Quicksortschritt:*
   Das gesamte Feld wird nun mit dem Pivot p zerlegt, so dass für gewisse $m_1 < m_2$ gilt:

$$\begin{array}{rclll} a[i] & < & p & \text{für } 1 \leq i \leq m_1 \\ a[i] & = & p & \text{für } m_1 < i \leq m_2 \\ a[i] & > & p & \text{für } m_2 < i \leq n \end{array}$$

   Kosten: maximal n Schritte.

3. *Fallunterscheidung:*
   (a) $k \leq m_1$: Suche das k-te Element rekursiv in $a[1], \ldots, a[m_1]$.
   (b) $m_1 < k \leq m_2$: Das Ergebnis ist p.
   (c) $k > m_2$: Suche das $(k - m_2)$-te Element in $a[m_2 + 1], \ldots, a[n]$.

   Wir werden zeigen, dass die Wahl des Pivots als Median-aus-Fünf folgende Ungleichungen für m_1, m_2 ergibt:

$$\frac{3}{10}n \leq m_2 \qquad \text{und} \qquad m_1 \leq \frac{7}{10}n$$

   Damit ergeben sich die Kosten für den Rekursionsschritt als $T(\frac{7n}{10})$.

Wir zeigen nur den ersten Teil der obigen Behauptung, d.h. $\frac{3}{10}n \leq m_2$:

Die Hälfte der Fünferblöcke hat einen Median kleiner oder gleich p. In jedem Block sind 3 Elemente kleiner oder gleich dem Blockmedian. Dies ergibt insgesamt $\frac{1}{2} \cdot \frac{n}{5} \cdot 3 = \frac{3n}{10}$ Elemente, die kleiner oder gleich p sind. Mit dieser Bemerkung sieht man auch, dass für den 2. Schritt des Algorithmus (Zerlegungsschritt) das Pivot p lediglich noch mit $\frac{2n}{5}$ Elementen verglichen werden muss, um m_1, m_2 bestimmen zu können.

Die Aussage für m_1 wird analog bewiesen.

Zeitanalyse: Sei $T(n)$ die Gesamtzahl der Vergleiche. Wir erhalten folgende Rekursionsgleichung für $T(n)$:

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + \mathcal{O}(n)$$

Damit erhält man $T(n) \in \mathcal{O}(n)$. (Man kann das direkt beweisen, es folgt aber auch aus dem sogenannten *Mastertheorem II*).

Eine genauere Analyse ergibt:

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + \frac{6n}{5} + \frac{2n}{5}$$

wobei $\frac{6n}{5}$ der Aufwand für die Bestimmung der Blockmediane und $\frac{2n}{5}$ der Aufwand für den Zerlegungsschritt ist. Damit kann nun gezeigt werden, dass $T(n) \leq 16n$ gilt: mit $\frac{1}{5} + \frac{7}{10} = \frac{9}{10}$ erhalten wir $T(n) \leq T(\frac{9n}{10}) + \frac{8n}{5}$ und damit $T(n) \leq 10 \cdot \frac{8n}{5} = 16n$.

2.4 Quickselect

Die im vorigen Abschnitt beschriebene Medianberechnung in linearer Zeit ist in der Praxis nicht immer sinnvoll, da die Ablaufstruktur in der Rekursion sehr kompliziert und die Konstante 16 groß ist. Der hohe Aufwand zur Bestimmung des Pivot-Elements lässt sich durch eine zufällige Auswahl dieses Elements vermeiden. Wie wir in diesem Abschnitt analysieren werden, erhalten wir damit einen Algorithmus, der Quicksort sehr ähnlich ist und der in erwarteter Linearzeit abläuft.

Der folgende Algorithmus verwendet die Funktion random(a, b), die eine (ganzzahlige) Zufallszahl aus dem Intervall $[a, \ldots, b]$ liefert. Die Prozedur **partitioniere** ist die selbe wie für Quicksort.

Algorithmus *Quickselect*

```
function quickselect(A[ℓ...r] : array of integer, k : integer) : integer
(* Bestimme das k-kleinste Element in A[ℓ...r] *)
begin
  if ℓ = r then
    return A[ℓ]
  else
    p := random(ℓ, r);                  (* Index des Pivot-El. zufällig best. *)
    m := partitioniere(A[ℓ...r], p);    (* Feld bzgl. A[p] partitionieren *)
    k' := (m − ℓ + 1);                  (* # Elemente in linkem Teilfeld *)
    if k ≤ k' then
      return quickselect(A[ℓ...m], k)
    else
      return quickselect(A[m + 1...r], k − k')
    endif
  endif
endfunction
```

Analyse von Quickselect. Ähnlich wie bei Quicksort können wir folgende Rekursionsgleichung aufstellen:

$$Q(n) = (n-1) + \frac{1}{n} \sum_{i=1}^{n-1} Q(\max\{i, n-i\})$$

Hierbei ist $(n-1)$ wiederum die Anzahl der Vergleiche für das Pivotieren und $Q(\max\{i, n-i\})$ die mittlere Anzahl der Vergleiche für den rekursiven Aufruf auf *einem* der beiden Teilfelder.

Es gilt:

$$\begin{aligned} Q(n) &= (n-1) + \frac{1}{n} \sum_{i=1}^{n-1} Q(\max\{i, n-i\}) \\ &= (n-1) + \frac{1}{n} \left(\sum_{i=1}^{\lceil \frac{n}{2} \rceil - 1} Q(n-i) + \sum_{i=\lceil \frac{n}{2} \rceil}^{n-1} Q(i) \right) \\ &\leq (n-1) + \frac{2}{n} \sum_{i=\lceil \frac{n}{2} \rceil}^{n-1} Q(i) \end{aligned}$$

Wir nehmen nun an, dass $Q(n) \leq c \cdot n$ für eine geeignete Konstante $c > 0$ ist.

Folgende Rechnung zeigt die Korrektheit dieser Annahme:

$$\begin{aligned}
Q(n) &\leq (n-1) + \frac{2}{n} \sum_{i=\lceil \frac{n}{2} \rceil}^{n-1} Q(i) \\
&\leq (n-1) + \frac{2c}{n} \sum_{i=\lceil \frac{n}{2} \rceil}^{n-1} i \\
&= (n-1) + \frac{2c}{n} \left(\sum_{i=1}^{n-1} i - \sum_{i=1}^{\lceil \frac{n}{2} \rceil - 1} i \right) \\
&= (n-1) + \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{\left(\lceil \frac{n}{2} \rceil - 1\right) \lceil \frac{n}{2} \rceil}{2} \right) \\
&\leq (n-1) + c(n-1) - c \cdot \frac{\frac{n}{2} - 1}{2} \\
&= (n-1) + c\left(\frac{3}{4}n - \frac{1}{2}\right) \\
&\leq c \cdot n
\end{aligned}$$

Die letzte Ungleichung gilt für alle $c \geq 4$.