Zweiter Schritt: Median der Mediane

Bisher haben wir bei $n/5$ Blöcken jeweils 6 Vergleiche verwendet, um die Blockmediane zu finden.
Der Gesamtaufwand ist hier also $\frac{6}{5}n$ Vergleiche.

Wenn wir aus jedem Fünferblock des Eingabe-Arrays jeweils einen Blockmedian ermittelt haben, können wir von diesen $\lceil \frac{n}{5} \rceil$ Elementen rekursiv den Median berechnen.

Welchen Aufwand brauchen wir hierfür?

Die Anzahl Vergleiche der Gesamtprozedur sei wie üblich durch $T(n)$ (bei Eingabe eines Arrays $a[1, \ldots, n]$) gegeben.

Der Aufwand dieser Rekursion ist also $T(\frac{n}{5})$.

Es sollte klar sein, dass dieser Median der Blockmediane nicht der wirkliche Median sein muss – aber er ist ein gutes Pivotelement.

Dritter Schritt: Feld aufteilen

Nun teilen wir mit dem ermittelten Median der Blockmediane als Pivotelement den gesamten Array auf in zwei Teile, von denen der erste, $a[1, \ldots, m]$, die Elemente enthält, die kleiner als das Pivotelement sind, der zweite, $a[m+1, \ldots, n]$, die übrigen.

Die folgende Feststellung ist entscheidend für diese Methode.

Es gilt: $\frac{3}{10}n \leq m \leq \frac{7}{10}n.$

Denn: Mit jedem Blockmedian, der kleiner als das Pivot-Element ist, sind insgesamt drei Elemente aus diesem Block im linken Teil. Das trifft bei der Hälfte der $n/5$ Blöcke, also bei $n/10$ Blöcken zu!

Auf der rechten Seite gilt das gleiche analog.

Damit ist klar, dass wir beim rekursiven Aufruf zum Ermitteln des gesuchten Elements nur mit einer Feldgröße bis $\frac{7}{10}n$ rechnen müssen.

Vierter Schritt: Rekursion

Nun wollen wir rekursiv das gesuchte Element in einem der beiden Teile finden. Wenn wir also das k-kleinste Element in $a[1, \ldots, n]$ suchen, dann unterscheiden wir so:

Ist $k \leq m$? Wenn ja: Suche das k-kleinste Element in $a[1, \ldots, m]$.
Wenn nein: Suche das $(k - m)$-kleinste Element in $a[m + 1, \ldots, n]$.

Dass damit das korrekte Element ermittelt wird, ist klar.
Aber welchen Gesamtaufwand haben wir nun?

Zunächst haben wir $\frac{6}{5}n$ Vergleiche für die Blockmediane verbraucht, dann $T(\frac{n}{5})$ für die erste Rekursion. Danach maximal n Vergleiche für den Quicksort-Schritt (tatsächlich genügen weniger), und zum Schluss noch einmal rekursiv $T(\frac{7}{10}n)$. Es folgt

$$T(n) \leq \tfrac{6}{5}n + T(\tfrac{n}{5}) + n + T(\tfrac{7}{10}n).$$

Mit dem Master-Theorem II folgt daraus lineare Anzahl Vergleiche.

Quickselect

Die Idee von Quickselect ist übernommen von Quicksort.
Man wählt ein zufälliges Pivot-Element und teilt das Feld in zwei Teile $a[1, \ldots, m]$ und $a[m+1, \ldots, n]$ auf.
Damit geht es rekursiv genau so weiter, wie eben bei der Medianberechnung in Linearzeit.

Im worst case kann es hier natürlich ebenso wie bei Quicksort passieren, dass man immer das kleinste oder das größte Element als Pivot bekommt – dann ist die Laufzeit auch hier quadratisch!
Wie sieht es im average case aus?

$Q(n)$ sei die Anzahl der Vergleiche, die man im Schnitt (bzgl. Gleichverteilung) bei der Durchführung des oben definierten Verfahrens machen muss. Dann gilt

$$Q(n) \leq 4n \quad \text{(Beweis an der Tafel)}$$

Zum Vergleich: Das vorher vorgestellte Linearzeitverfahren braucht $16n$ Vergleiche...

Der Dijkstra-Algorithmus

Wir wollen uns noch einmal dem Dijkstra-Algorithmus zur Lösung des single-source-all-targets shortest-paths Problems zuwenden. (Vergleiche Folien 5.5 und 5.6)

Wir geben zunächst den Algorithmus relativ detailliert an. Dabei sei $G = (V, E)$ ein gerichteter Graph ohne Schlingen oder Mehrfachkanten, und es sei eine Kantengewichtung gegeben durch $\gamma : E \to \mathbb{N} \setminus \{0\}$. $D[1, \ldots, n]$ mit $n = |V|$ sei ein Array, in dem wir anfangs Schätzwerte und später die genauen Abstandswerte der Knoten $x = x_i$ von u (dem gegebenen Startknoten) abspeichern wollen. Schließlich sei $v[1, \ldots, n]$ ein Array, in dem wir abspeichern wollen, von welchem Knoten $y = v[i]$ wir beim kürzesten Weg zum Knoten x_i gehen.

Der Algorithmus

```
B := {u}; R := ∅; v(u) := undef; D(u) := 0;     (* Init B, R und U *)
FORALL y ∈ V \ {u} mit (u,y) ∈ E DO
        D(y) := γ(u,y); v(y) := u; R := R ∪ {y};
ENDFOR;
U := V \ (R ∪ {u});
WHILE R ≠ ∅ DO
        x := undef; α := ∞;                  (* suche x mit D(x) minimal *)
        FORALL y ∈ R DO
                IF D(y) < α THEN x := y; α := D(y) ENDIF
        ENDFOR
        B := B ∪ {x}; R := R \ {x};                (* x von R nach B *)
        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);  v(y) := x;   (* kürzerer Weg *)
                                                             (*    über x    *)
                ENDIF
        ENDFOR
ENDWHILE
```

Terminierung

Um nachzuweisen, dass dieser Algorithmus tut, was wir uns von ihm erwarten, müssen wir nun zweierlei tun:

1. Zeigen, dass der Algorithmus immer terminiert.
2. Zeigen, dass die berechneten Werte immer korrekt sind.

Zunächst zum Terminieren:

Wir betrachten die Menge B, die immer eine Teilmenge von V ist, d.h. sie kann maximal n Elemente haben. Zu Beginn hat B ein Element. In jedem Durchgang der WHILE-Schleife kommt ein weiteres dazu, das aus R entfernt wird. Alle Elemente in R kommen aber aus U, wo wiederum jeder Knoten, der nach R geschoben wird, entfernt wird. Damit kann jeder Knoten höchstens einmal zu B hinzugefügt werden, d.h. maximal gibt es $n - 1$ Schleifendurchgänge.