The Median-Of-Medians Selection Algorithm

Manuel Eberl
August 16, 2018

Abstract

This entry provides an executable functional implementation of the Median-of-Medians algorithm [1] for selecting the $k$-th smallest element of an unsorted list deterministically in linear time. The size bounds for the recursive call that lead to the linear upper bound on the run-time of the algorithm are also proven.

Contents

1 The Median-of-Medians Selection Algorithm 1
  1.1 Some facts about lists and multisets ....................... 1
  1.2 The dual order type ..................................... 3
  1.3 Chopping a list into equal-sized sublists ................. 4
  1.4 $k$-th order statistics and medians ...................... 6
  1.5 A more liberal notion of medians ......................... 8
  1.6 Properties of a median-of-medians ....................... 10
  1.7 The recursive step ...................................... 14
  1.8 Medians of lists of length at most 5 .................... 16
  1.9 Median-of-medians selection algorithm .................. 18

1 The Median-of-Medians Selection Algorithm

theory Median-Of-Medians-Selection
imports Complex-Main HOL-Library.Multiset
begin

1.1 Some facts about lists and multisets

lemma mset-concat: mset (concat xss) = sum-list (map mset xss)
  by (induction xss) simp-all

lemma set-mset-sum-list [simp]: set-mset (sum-list xs) = (⋃x∈set xs. set-mset x)
  by (induction xs) auto

lemma filter-mset-image-mset:
filter-mset P (image-mset f A) = image-mset f (filter-mset (λx. P (f x)) A)
by (induction A) auto

lemma filter-mset-sum-list: filter-mset P (sum-list xs) = sum-list (map (filter-mset P) xs)
by (induction xs) simp-all

lemma sum-mset-mset-mono:
assumes \( \forall x. x \in \# A \Rightarrow f x \subseteq \# g x \)
shows \( (\sum x \in \# A. f x) \subseteq (\sum x \in \# A. g x) \)
using assms by (induction A) (auto intro: subset-mset.add-mono)

lemma mset-filter-mono:
assumes \( A \subseteq \# B \:\&\:\\forall x. x \in \# A \Rightarrow P x \Rightarrow Q x \)
shows \( \text{filter-mset P A} \subseteq \# \text{filter-mset Q B} \)
by (rule mset-subset-eqI) (insert assms, auto simp: mset-subset-eq-count count-eq-zero-iff)

lemma size-mset-sum-mset-distrib:
size (\sum x \in \# A. f x) = \sum (\text{image-mset size A})
by (induction A) auto

lemma sum-mset-mono:
assumes \( \forall x. x \in \# A \Rightarrow f x \leq \# g x \)
shows \( \sum x \in \# A. f x \leq \sum x \in \# A. g x \)
using assms by (induction A) (auto intro: add-mono)

lemma filter-mset-is-empty-iff:
filter-mset P A = \{\#\} \iff \( \forall x. x \in \# A \Rightarrow \neg P \)
by (auto simp: mset-subset-eq-count count-eq-zero-iff)

lemma sorted-filter-less-subset-take:
assumes sorted xs i < length xs
shows \( \{\# y \in \# \text{mset xs}. y < (x \# xs) ! i \#\} \subseteq \# \text{mset (take i xs)} \)
using assms proof (induction xs arbitrary: i rule: list.induct)
case (Cons x xs i)
show ?case
proof (cases i)
case 0
thus ?thesis using Cons.prems by (auto simp: filter-mset-is-empty-iff)
next
case (Suc i)
have \( \{\# y \in \# \text{mset (x \# xs)}. y < (x \# xs) ! i \#\} \subseteq \# \text{add-mset x} \{\# y \in \# \text{mset xs}. y < (x \# xs) ! i'\#\} \)
using Suc Cons.prems by (auto)
also have \( \ldots \subseteq \# \text{add-mset x (mset (take i' xs))} \)
unfolding mset-subset-eq-add-mset-cancel using Cons.prems Suc
by (intro Cons.IH) (auto)
also have \( \ldots = \text{mset (take i (x \# xs))} \) by (simp add: Suc)
finally show \( \text{thesis} \).

\textbf{qed}

\textbf{qed auto}

\textbf{lemma} sorted-filter-greater-subset-drop:
\begin{itemize}
  \item \textbf{assumes} sorted \( xs \) \( i < \) length \( xs \)
  \item \textbf{shows} \( \{ \# x \in\# \text{mset} \, xs. \, x \succ xs ! \# i \}\subseteq \# \text{mset} \, (\text{drop} \, (\text{Suc} \, i) \, xs) \)
\end{itemize}
\textbf{using} \text{assms}

\textbf{proof} (induction \( xs \) arbitrary; \( i \) rule: list.induct)
\begin{itemize}
  \item \textbf{case} (\text{Cons} \, x \, xs \, i)
    \item \textbf{show} \( ?\text{case} \)
      \begin{itemize}
        \item \textbf{proof} (cases \( i \))
          \begin{itemize}
            \item \textbf{case} 0
              \item \textbf{thus} \( ?\text{thesis} \) \textbf{by} (auto simp: sorted-append filter-mset-is-empty-iff)
            \item \textbf{next}
              \item \textbf{case} (\text{Suc} \, i')
                \item \textbf{have} \( \{ \# y \in\# \text{mset} \, (x \# xs). \, y > (x \# xs)! i\# \} \subseteq \# \{ \# y \in\# \text{mset} \, xs. \, y > xs ! i'\# \} \)
                  \textbf{using} \text{Suc Cons.prems by} (auto simp: set-cone-nth)
                \item \textbf{also have} \( \ldots \subseteq \# \text{mset} \, (\text{drop} \, (\text{Suc} \, i') \, xs) \)
                  \textbf{using} \text{Cons.prems Suc by} (intro Cons.IH) (auto)
                \item \textbf{also have} \( \ldots = \text{mset} \, (\text{drop} \, (\text{Suc} \, i) \, (x \# xs)) \) \textbf{by} (simp add: Suc)
                \item \textbf{finally show} \( ?\text{thesis} \).
          \end{itemize}
        \end{itemize}
      \end{itemize}
\end{itemize}

\textbf{qed}

\textbf{qed auto}

\textbf{1.2 The dual order type}

The following type is a copy of a given ordered base type, but with the ordering reversed. This will be useful later because we can do some of our reasoning simply by symmetry.

\textbf{typedef} ‘a dual-ord = \text{UNIV} :: ‘a set \textbf{morphisms} of-dual-ord to-dual-ord
\textbf{by} \text{auto}

\textbf{setup-lifting} type-definition-dual-ord

\textbf{instantiation} dual-ord :: (ord) ord

\textbf{begin}

\textbf{lift-definition} less-eq-dual-ord :: ‘a dual-ord \Rightarrow ‘a dual-ord \Rightarrow \text{bool}
\textbf{is} \( \lambda a \, b :: ‘a. \, a \geq b \).

\textbf{lift-definition} less-dual-ord :: ‘a dual-ord \Rightarrow ‘a dual-ord \Rightarrow \text{bool}
\textbf{is} \( \lambda a \, b :: ‘a. \, a > b \).

\textbf{instance} ..

\textbf{end}

\textbf{instance} dual-ord :: (preorder) preorder

3
by standard (transfer; force simp: less-le-not-le intro: order-trans)+

instance dual-ord :: (linorder) linorder
  by standard (transfer; force simp: not-le)+

1.3 Chopping a list into equal-sized sublists

function chop :: nat ⇒ 'a list ⇒ 'a list list where
  chop n [] = []
  | chop 0 xs = []
  | n > 0 ⇒ xs ≠ [] ⇒ chop n xs = take n xs # chop n (drop n xs)
  by force+
termination by lexicographic-order

context
  includes lifting-syntax

begin

lemma chop-transfer [transfer-rule]:
  ((=) ===> list-all2 R ===> list-all2 (list-all2 R)) chop chop
proof (intro rel-funI)
  fix m n :: nat and xs :: 'a list and ys :: 'b list
  assume m = n list-all2 R xs ys
  from this(2) have list-all2 (list-all2 R) (chop n xs) (chop n ys)
  proof (induction n xs arbitrary: ys rule: chop.induct)
    case (3 n xs ys)
    hence ys ≠ [] by auto
    with 3 show ?case by auto
  qed auto
  with ⟨m = n⟩ show list-all2 (list-all2 R) (chop m xs) (chop n ys) by simp
  qed

end

lemma chop-reduce: chop n xs = (if n = 0 ∨ xs = [] then [] else take n xs # chop n (drop n xs))
  by (cases n = 0; cases xs = []) auto

lemma concat-chop [simp]: n > 0 ==> concat (chop n xs) = xs
  by (induction n xs rule: chop.induct) auto

lemma chop-elem-not-Nil [simp,dest]: ys ∈ set (chop n xs) ==> ys ≠ []
  by (induction n xs rule: chop.induct) (auto simp: eq-commute[of []])

lemma chop-eq-Nil-iff [simp]: chop n xs = [] ⇔ n = 0 ∨ xs = []
  by (induction n xs rule: chop.induct) auto

lemma chop-ge-length-eq: n > 0 ==> xs ≠ [] ==> n ≥ length xs ==> chop n xs = [xs]
lemma length-chop-part-le: \( ys \in \text{set} \ (\text{chop} \ n \ xs) \implies \text{length} \ ys \leq n \)
by (induction \( n \) \( xs \) rule: chop.induct) auto

lemma length-nth-chop:
assumes \( i < \text{length} \ (\text{chop} \ n \ xs) \)
shows \( \text{length} \ (\text{chop} \ n \ xs ! i) = \)
\( \text{if} \ i = \text{length} \ (\text{chop} \ n \ xs) - 1 \land \neg n \text{ dvd length} \ xs \text{ then length} \ xs \mod n \)
else \( n \)
proof (cases \( n = 0 \))
case False
thus \(?\text{thesis}\)
using assms
proof (induction \( n \) \( xs \) arbitrary: \( i \) rule: chop.induct)
case (3 \( n \) \( xs \) \( i \))
show \(?\text{case}\)
proof (cases \( i \))
case 0
thus \(?\text{thesis using 3.prems}\)
by (cases \( \text{length} \ xs < n \)) (auto simp: le-Suc-eq dest: dvd-imp-le)
next
case [simp]: (Suc \( i \))
with 3.prems have [simp]: \( \text{xs} \neq [] \) by auto
with 3.prems have \(*\): \( \text{length} \ xs > n \) by (cases \( \text{length} \ xs \leq n \)) simp-all
with 3.prems have \( \text{chop} \ n \ xs ! i = \text{chop} \ n \ (\text{drop} \ n \ xs) ! i' \) by simp
also have \( \text{length} \ldots = (\text{if} \ i = \text{length} \ (\text{chop} \ n \ xs) - 1 \land \neg n \text{ dvd (length} \ xs - n) \)
\( \text{then (length} \ xs - n) \mod n \text{ else } n \) \)
by (subst 3.IH) (use Suc 3.prems in auto)
also have \( n \text{ dvd (length} \ xs - n) \iff n \text{ dvd length} \ xs \)
using \(*\) by (subst dvd-minus-self) auto
also have \( \text{(length} \ xs - n) \mod n = \text{length} \ xs \mod n \)
using \(*\) by (subst le-mod-geq [symmetric]) auto
finally show \(?\text{thesis} .\)
qed
qed auto
qed (insert assms, auto)

lemma length-chop:
assumes \( n > 0 \)
shows \( \text{length} \ (\text{chop} \ n \ xs) = \text{nat \ [length} \ xs / n] \)
using assms
proof (induction \( n \) \( xs \) rule: chop.induct)
case (3 \( n \) \( xs \))
show \(?case\)
proof (cases \( \text{length} \ xs \geq n \))
case False
hence \( [\text{real} \ (\text{length} \ xs) \ / \ \text{real} \ n] = 1 \) using 3.hyps
by (intro ceiling-unique) auto

with False show ?thesis using 3.prems 3.hyps
  by (auto simp: chop-ge-length-eq not-le)

next
  case True
  hence real (length xs) = real n + real (length (drop n xs))
  by simp
  also have ... / real n = real (length (drop n xs)) / real n + 1
    using (n > 0) by (simp add: divide-simps)
  also have ceiling ... = ceiling (real (length (drop n xs)) / real n + 1)
  by simp
  also have nat ... = nat (ceiling (real (length (drop n xs)) / real n)) + nat 1
  by (intro nat-add-distrib[OF order.trans[OF - ceiling-mono[of 0]]]) auto
  also have ... = length (chop n xs)
  using ⟨n > 0⟩ 3.hyps by (subst 3.IH [symmetric]) auto
  finally show ?thesis ..

qed

qed auto

lemma sum-msets-chop: n > 0 ⇒ (∑ys ← chop n xs. mset ys) = mset xs
  by (subst mset-concat [symmetric]) simp-all

lemma UN-sets-chop: n > 0 ⇒ (∪ys ∈ set (chop n xs). set ys) = set xs
  by (simp only: set-concat [symmetric] concat-chop)

lemma in-set-chopD [dest]:
  assumes x ∈ set ys ys ∈ set (chop d xs)
  shows x ∈ set xs
proof (cases d > 0)
  case True
  thus ?thesis by (subst UN-sets-chop [symmetric]) (use assms in auto)

qed (use assms in auto)

1.4 k-th order statistics and medians

This returns the k-th smallest element of a list. This is also known as the k-th order statistic.

definition select :: nat ⇒ 'a list ⇒ ('a :: linorder) where
  select k xs = sort xs ! k

The median of a list, where, for lists of even lengths, the smaller one is favoured:

definition median where median xs = select ((length xs - 1) div 2) xs

lemma select-in-set [intro,simp]:
  assumes k < length xs
  shows select k xs ∈ set xs
proof
  from assms have sort zs ! k ∈ set (sort zs) by (intro nth-mem) auto
also have \( \text{set } (\text{sort } xs) = \text{set } xs \) by simp
finally show \( ?\text{thesis} \) by (simp add: select-def)
qed

lemma median-in-set [intro, simp]:
  assumes \( xs \neq [] \)
  shows \( \text{median } xs \in \text{set } xs \)
proof –
  from assms have \( \text{length } xs > 0 \) by auto
  hence \( (\text{length } xs - 1) \div 2 < \text{length } xs \) by linarith
  thus \( ?\text{thesis} \) by (simp add: median-def)
qed

We show that selection and medians does not depend on the order of the elements:

lemma sort-cong: \( \text{mset } xs = \text{mset } ys \Rightarrow \text{sort } xs = \text{sort } ys \)
  by (rule properties-for-sort simp-all)

lemma select-cong:
  \( k = k' \Rightarrow \text{mset } xs = \text{mset } xs' \Rightarrow \text{select } k xs = \text{select } k' xs' \)
  by (auto simp dest: sort-cong)

lemma median-cong: \( \text{mset } xs = \text{mset } xs' \Rightarrow \text{median } xs = \text{median } xs' \)
  unfolding median-def by (intro select-cong) (auto dest: mset-eq-length)

Selection distributes over appending lists under certain conditions:

lemma sort-append:
  assumes \( \forall x \ y. x \in \text{set } xs \Rightarrow y \in \text{set } ys \Rightarrow x \leq y \)
  shows \( \text{sort } (xs @ ys) = \text{sort } xs @ \text{sort } ys \)
  using assms by (intro properties-for-sort) (auto simp: sorted-append)

lemma select-append:
  assumes \( \forall y \ z. y \in \text{set } ys \Rightarrow z \in \text{set } zs \Rightarrow y \leq z \)
  shows \( k < \text{length } ys \Rightarrow \text{select } k (ys @ zs) = \text{select } k ys \\
  k \in \{\text{length } ys..<\text{length } ys + \text{length } zs\} \Rightarrow \\
  \text{select } k (ys @ zs) = \text{select } (k - \text{length } ys) zs \)
  using assms by (simp-all add: select-def sort-append nth-append)

lemma select-append':
  assumes \( \forall y \ z. y \in \text{set } ys \Rightarrow z \in \text{set } zs \Rightarrow y \leq z \ k < \text{length } ys + \text{length } zs \)
  shows \( \text{select } k (ys @ zs) = (if k < \text{length } ys \text{ then select } k ys \text{ else select } (k - \text{length } ys) zs) \)
  using assms by (auto intro!: select-append)

We can find simple upper bounds for the number of elements that are strictly less than (resp. greater than) the median of a list.

lemma size-less-than-median:
  \( \text{size } \{\#y \in \# \text{mset } xs. y < \text{median } xs\} \leq (\text{length } xs - 1) \div 2 \)
proof (cases xs = [])
case False
  hence length xs > 0 by simp
  hence (length xs - 1) div 2 < length xs by linarith
  hence size {#y ∈# mset (sort xs). y < median xs#} ≤
    size (mset (take ((length xs - 1) div 2) (sort xs)))
  unfolding median-def select-def using False
  thus ?thesis using False by simp
qed auto

lemma size-greater-than-median:
  size {#y ∈# mset xs. y > median xs#} ≤ length xs div 2
proof (cases xs = [])
case False
  hence length xs > 0 by simp
  hence (length xs - 1) div 2 < length xs by linarith
  hence size {#y ∈# mset (sort xs). y > median xs#} ≤
    size (mset (drop (Suc ((length xs - 1) div 2)) (sort xs)))
  unfolding median-def select-def using False
  by (intro size-mset-mono sorted-filter-greater-subset-drop) auto
  hence size (filter-mset (λy. y > median xs) (mset xs)) ≤
    length xs - Suc ((length xs - 1) div 2) by simp
  also have ... = length xs div 2 by linarith
  finally show ?thesis .
qed auto

1.5 A more liberal notion of medians

We now define a more relaxed version of being “a median” as opposed to being “the median”. A value is a median if at most half the values in the list are strictly smaller than it and at most half are strictly greater. Note that, by this definition, the median does not even have to be in the list itself.

definition is-median :: 'a::linorder ⇒ 'a list ⇒ bool where
  is-median x xs ←→ length (filter (λy. y < x) xs) ≤ length xs div 2 ∧
    length (filter (λy. y > x) xs) ≤ length xs div 2

We set up some transfer rules for is-median. In particular, we have a rule that shows that something is a median for a list iff it is a median on that list w. r. t. the dual order, which will later allow us to argue by symmetry.

class context
  includes lifting-syntax
begin
lemma transfer-is-median [transfer-rule]:
  assumes [transfer-rule]: (r ===> r ===> (=)) (<) (<)
  shows (r ===> list-all2 r ===> (=)) is-median is-median
  unfolding is-median-def by transfer-prover
lemma list-all2-eq-fun-conv-map: list-all2 ($\lambda x. x = f y$) $xs$ $ys \iff xs = \text{map } f ys$

proof
  assume list-all2 ($\lambda x. x = f y$) $xs$ $ys$
  thus $xs = \text{map } f ys$ by induction auto

next
  assume $xs = \text{map } f ys$
  moreover have list-all2 ($\lambda x. x = f y$) ($\text{map } f ys$) $ys$
    by (induction $ys$) auto
  ultimately show list-all2 ($\lambda x. x = f y$) $xs$ $ys$ by simp

qed

lemma transfer-is-median-dual-ord [transfer-rule]:
  ($\text{pcr-dual-ord ($\lambda x. x = y$)}$ $\Longrightarrow$ list-all2 ($\text{pcr-dual-ord ($\lambda x. x = y$)}$) $\Longrightarrow$ ($\lambda x. x = y$) $\Longrightarrow$ ($\lambda x. x = y$)) $\iff$ is-median
  by (auto simp: pcr-dual-ord-def cr-dual-ord-def OO-def rel-fun-def is-median-def list-all2-eq-fun-conv-map o-def less-dual-ord.rep-eq)

end

lemma is-median-to-dual-ord-iff [simp]:
  is-median ($\text{to-dual-ord } x$) ($\text{map to-dual-ord } xs$) $\iff$ is-median $x$ $xs$
  unfolding is-median-def by transfer auto

The following is an obviously equivalent definition of is-median in terms of multisets that is occasionally nicer to use.

lemma is-median-altdef:
  is-median $x$ $xs \iff \text{size} (\text{filter-mset } (\lambda y. y < x) (\text{mset } xs)) \leq \text{length } xs \div 2 \land \\
  \text{size} (\text{filter-mset } (\lambda y. y > x) (\text{mset } xs)) \leq \text{length } xs \div 2$

proof –
  have $\ast$: length (filter $P$ $xs$) = size (filter-mset $P$ (mset $xs$)) for $P$ and $xs :: 'a list$
    by (simp add: mset-filter [symmetric])
  show $\text{thesis}$ by (simp only: is-median-def $\ast$)

qed

lemma is-median-cong:
  assumes $x = y$ $\text{mset } xs = \text{mset } ys$
  shows is-median $x$ $xs \iff$ is-median $y$ $ys$
  unfolding is-median-altdef by (simp only: assms mset-eq-length[OF assms(2)])

If an element is the median of a list of odd length, we can add any element to the list and the element is still a median. Conversely, if we want to compute a median of a list with even length $n$, we can simply drop one element and reduce the problem to a median of a list of size $n - 1$.

lemma is-median-Cons-odd:
  assumes is-median $x$ $xs$ and odd (length $xs$)
  shows is-median $x$ ($y \# xs$)
  using assms by (auto simp: is-median-def)
And, of course, the median is a median.

**Lemma** \(\text{is-median-median} \ [\text{simp}, \text{intro}]: \text{is-median} \ (\text{median} \ x) \ x\) using \(\text{size-less-than-median}[\text{of} \ x] \ \text{size-greater-than-median}[\text{of} \ x]\)

unfolding \(\text{is-median-def} \ \text{size-mset} \ [\text{symmetric}] \ \text{mset-filter} \ \text{by} \ \text{linarith}\)

### 1.6 Properties of a median-of-medians

We can now bound the number of list elements that can be strictly smaller than a median-of-medians of a chopped-up list (where each part has length \(d\) except for the last one, which can also be shorter).

The core argument is that at least roughly half of the medians of the sublists are greater or equal to the median-of-medians, and about \(\frac{d}{2}\) elements in each such sublist are greater than or equal to their median and thereby also than the median-of-medians.

**Lemma** \(\text{size-less-than-median-of-medians-strong}: \) fixes \(x::\text{linorder list} \ \text{and} \ d::\text{nat}\) assumes \(d::d>0\) assumes \(\text{median}::\bigwedge \ x. \ x \neq [] \Rightarrow \text{length} \ x \leq d \Rightarrow \text{is-median} \ (\text{med} \ x) \ x\)

defines \(m::\text{length} \ (\text{chop} \ d \ x)\)

shows \(\text{size} \ \{\#y \in\# \ \text{mset} \ x. \ y<x\}\leq m* (d \ \text{div} \ 2)+ m \ \text{div} \ 2 * ((d+1) \ \text{div} \ 2)\)

**Proof** —

\begin{enumerate}
\item Define \(n\) where [simp]: \(n = \text{length} \ x\) — The medians of the sublists
\item Define \(M\) where \(M = \text{mset} \ (\text{map} \ \text{med} \ (\text{chop} \ d \ x))\)
\item Define \(YS\) where \(YS = \text{mset} \ (\text{chop} \ d \ x)\) — The sublists with a smaller median than the median-of-medians \(x\) and the rest.
\item Define \(YS1\) where \(YS1 = \text{filter-mset} \ (\lambda \ \text{ys}. \ \text{med} \ \text{ys} < x) \ (\text{mset} \ (\text{chop} \ d \ x))\)
\item Define \(YS2\) where \(YS2 = \text{filter-mset} \ (\lambda \ \text{ys}. \ (\neg \text{med} \ \text{ys} < x)) \ (\text{mset} \ (\text{chop} \ d \ x))\)
\end{enumerate}

— At most roughly half of the lists have a median that is smaller than \(M\)

\begin{enumerate}
\item Have \(\text{size} \ YS1 = \text{size} \ (\text{image-mset} \ \text{med} \ YS1)\) by simp
\item Also have \(\text{image-mset} \ \text{med} \ YS1 = \{\#y \in\# \ \text{mset} \ (\text{map} \ \text{med} \ (\text{chop} \ d \ x)). \ y< x\}\) unfolding \(YS1\)-def by \((\text{subst filter-mset-image-mset} \ [\text{symmetric}]) \ \text{simp-all}\)
\item Also have \(\text{size} \ ... \leq (\text{length} \ (\text{map} \ \text{med} \ (\text{chop} \ d \ x)) ) \ \text{div} \ 2\) using median\’_unfolding is-median-altdef by simp
\item Also have \(\ldots = m \ \text{div} \ 2\) by \((\text{simp add: m-def})\)
\item Finally have \(\text{size}-YS1: \text{size} \ YS1 \leq m \ \text{div} \ 2\).
\end{enumerate}

\begin{enumerate}
\item Have \(m = \text{size} \ (\text{mset} \ (\text{chop} \ d \ x))\) by \((\text{simp add: m-def})\)
\item Also have \(\text{mset} \ (\text{chop} \ d \ x) = YS1 + YS2\) unfolding \(YS1\)-def \(YS2\)-def by \((\text{rule multiset-partition})\)
\item Finally have \(m\)-eq: \(m = \text{size} \ YS1 + \text{size} \ YS2\) by simp
\end{enumerate}

— We estimate the number of elements less than \(x\) by grouping them into elements...
coming from YS1 and elements coming from YS2. In the first case, we just note
that no more than $d$ elements can come from each sublist, whereas in the second
case, we make the analysis more precise and note that only elements that are less
than the median of their sublist can be less than $x$.

**have** \{ \# $y \in \# \text{mset } x, \; y < x \#\} = \{ \# y \in \# (\sum ys \leftarrow \text{chop } d \; x, \; \text{mset } y) \; y < x \#\}

**using** $d$

**by** (subst sum-msets-chop) simp-all

**also have** \ldots = (\sum ys \leftarrow \text{chop } d \; x. \; \{ \# y \in \# \text{mset } y, \; y < x \#\})

**by** (subst filter-mset-sum-list) (simp add: o-def)

**also have** \ldots = (\sum ys \in \# YS. \; \{ \# y \in \# \text{mset } y, \; y < x \#\}) unfolding YS-def

**by** (subst sum-mset-sum-list [symmetric]) simp-all

**also have** $YS = YS1 + YS2$

**by** (simp add: YS-def YS1-def YS2-def not-le multiset-partition [symmetric])

**also have** \( (\sum ys \in \# \ldots \{ \# y \in \# \text{mset } y, \; y < x \#\}) = \)

\( (\sum ys \in \# YS1. \; \{ \# y \in \# \text{mset } y, \; y < x \#\}) + (\sum ys \in \# YS2. \; \{ \# y \in \# \text{mset } y, \; y < x \#\}) \)

**by** simp

**also have** \ldots $\subseteq \ldots (\sum ys \in \# YS1. \; \text{mset } y) + (\sum ys \in \# YS2. \; \{ \# y \in \# \text{mset } y, \; y < \text{med } ys\#\})$

**by** (intro subset-mset.add-mono sum-mset-mset-mono mset-filter-mono) (auto simp: YS2-def)

**finally have** \{ \# y \in \# \text{mset } x, \; y < x \#\} $\subseteq \ldots$

**hence** \( \text{size } \{ \# y \in \# \text{mset } x. \; y < x \#\} \leq \ldots \) by (rule size-mset-mono)

— We do some further straightforward estimations and arrive at our goal.

**also have** \ldots = (\sum ys \in \# YS1. \; \text{length } ys) + (\sum x \in \# YS2. \; \text{size } \{ \# y \in \# \text{mset } x, \; y < \text{med } x\#\})

**by** (simp add: size-mset-sum-mset-distrib multiset.map-comp o-def)

**also have** \( (\sum ys \in \# YS1. \; \text{length } ys) \leq (\sum ys \in \# YS1. \; d) \)

**by** (intro sum-mset-mono) (auto simp: YS1-def length-chop-part-le)

**also have** \ldots $=$ size YS1 * $d$ by simp

**also have** \( d = (d \div 2) + ((d + 1) \div 2) \) using $d$ by linarith

**have** size YS1 * $d = size YS1 * (d \div 2) + size YS1 * ((d + 1) \div 2)$

**by** (subst $d$) (simp add: algebra-simps)

**also have** \( (\sum ys \in \# YS2. \; \text{size } \{ \# y \in \# \text{mset } y, \; y < \text{med } ys\#\}) \leq \)

\( (\sum ys \in \# YS2. \; \text{length } ys \div 2) \)

**proof** (intro sum-mset-mono size-less-than-median, goal-cases)

**case** (1 ys)

**hence** ys $\neq [] \text{length } ys \leq d$ by (auto simp: YS2-def length-chop-part-le)

**from** median[OF this] **show** ?case by (auto simp: is-median-altdef)

**qed**

**also have** \ldots $\leq (\sum ys \in \# YS2. \; d \div 2)$

**by** (intro sum-mset-mono div-le-mono divf-le-mono) (auto simp: YS2-def dest: length-chop-part-le)

**also have** \ldots $=$ size YS2 * $d \div 2$ by simp

**also have** size YS1 * $d \div 2$ + size YS1 * $((d + 1) \div 2)$ + \ldots $=$

\( m \times (d \div 2) + size YS1 * ((d + 1) \div 2) \) by (simp add: m-eq algebra-simps)

**also have** size YS1 * $((d + 1) \div 2)$ $\leq (m \div 2) \times ((d + 1) \div 2)$

11
by (intro mult-right-mono size-YS1) auto
finally show \( \text{size} \{ \#y \in \# \text{mset} \; xs. \; y < x\# \} \leq m * (d \div 2) + m \div 2 * ((d + 1) \div 2) \) by simp-all
qed

We now focus on the case of an odd chopping size and make some further estimations to simplify the above result a little bit.

**Theorem** size-less-than-median-of-medians:
fixes \( xs :: 'a :: \text{linorder list} \) and \( d :: \text{nat} \)
assumes median: \( \forall xs. \; xs \neq [] \implies \text{length} \; xs \leq \text{Suc} \; (2 * d) \implies \text{is-median} \; (\text{med} \; xs) \; xs \)
assumes median’: \( \text{is-median} \; x \; (\text{map} \; \text{med} \; (\text{chop} \; (\text{Suc} \; (2*d)) \; xs)) \)
defines \( n \equiv \text{length} \; xs \)
defines \( c \equiv (3 * \text{real} \; d + 1) / (2 * (2 * d + 1)) \)
shows \( \text{size} \{ \#y \in \# \; \text{mset} \; xs. \; y < x\# \} \leq \text{nat} \; [c * n] + (5 * d) \div 2 + 1 \)
proof (cases \( xs = [] \))
case False
define \( m \) where \( m = \text{length} \; (\text{chop} \; (\text{Suc} \; (2*d)) \; xs) \)

have \( \text{real} \; (m \div 2) \leq \text{real} \; (\text{nat} \; [\text{real} \; n / (1 + 2 * \text{real} \; d)]) / 2 \)
  by (simp add: m-def length-chop n-def)
also have \( \text{real} \; (\text{nat} \; [\text{real} \; n / (1 + 2 * \text{real} \; d)]) = \text{of-int} \; [\text{real} \; n / (1 + 2 * \text{real} \; d)] \)
  by (intro of-nat-nat) (auto simp: divide-simps)
also have \( \ldots / 2 \leq (\text{real} \; n / (1 + 2 * \text{real} \; d) + 1) / 2 \)
  by (intro divide-right-mono) linarith+
also have \( \ldots = n / (2 * (2 * \text{real} \; d + 1)) + 1 / 2 \)
  by (simp add: field-simps)
finally have \( m : \text{real} \; (m \div 2) \leq \ldots \).

have \( \text{size} \{ \#y \in \# \; \text{mset} \; xs. \; y < x\# \} \leq d * m + \text{Suc} \; d * (m \div 2) \)
  using size-less-than-median-of-medians-strong[of Suc (2 * d) med xs] assms
unfolding m-def by (simp add: algebra-simps)
also have \( \ldots \leq d * (2 * (m \div 2) + 1) + \text{Suc} \; d * (m \div 2) \)
  by (intro add-mono mult-left-mono) linarith+
also have \( \ldots = (3 * d + 1) * (m \div 2) + d \)
  by (simp add: algebra-simps)
finally have \( \text{real} \; (\text{size} \; \{ \#y \in \# \; \text{mset} \; xs. \; y < x\# \}) \leq \text{real} \ldots \)
  by (subst of-nat-le-iff)
also have \( \ldots \leq (3 * \text{real} \; d + 1) * (n / (2 * (2 * d + 1)) + 1/2) + \text{real} \; d \)
unfolding of-nat-add of-nat-mult of-nat-1 of-nat-numeral
  by (intro add-mono mult-mono order_refl m) (auto simp: m-def length-chop n-def add-ac)
also have \( \ldots = c * \text{real} \; n + (5 * \text{real} \; d + 1) / 2 \)
  by (simp add: field-simps c-def)
also have \( \ldots \leq \text{real} \; [c * n] + ((5 * d) \div 2 + 1) \)
unfolding of-nat-add by (intro add-mono) (linarith, simp add: field-simps)
finally show \( ?\text{thesis} \) by (subst (asm) of-nat-le-iff) (simp-all add: add-ac)
qed auto
We get the analogous result for the number of elements that are greater than a median-of-medians by looking at the dual order and using the transfer method.

**Theorem size-greater-than-median-of-medians:**

\[\text{fixes } xs :: 'a :: linorder list and } d :: \text{nat} \]
\[\text{assumes median: } \forall xs. \; xs \neq [] \Rightarrow \text{length } xs \leq \text{Suc } (2 \ast d) \Rightarrow \text{is-median } (\text{med } xs) \; xs\]
\[\text{assumes median': is-median } x \; (\text{map med } (\text{chop } \text{Suc } (2 \ast d)) \; xs)\]
\[\text{defines } n \equiv \text{length } xs\]
\[\text{defines } c \equiv (3 \ast \text{real } d + 1) / (2 \ast (2 \ast d + 1))\]
\[\text{shows } \text{size } \{ \# y \in \#. \; \text{mset } y. \; y > x\# \} \leq \text{nat } \left[ c \ast n \right] + (5 \ast d) \; \text{div } 2 + 1\]

**Proof** –

\[\text{include lifting-syntax}\]
\[\text{define med' where med' } = (\lambda xs. \text{to-dual-ord } (\text{med } \text{of-dual-ord } xs))\]
\[\text{have } xs = \text{map of-dual-ord } ys \; \text{if} \; \text{list-all2 } \text{cr-dual-ord } xs \; ys \; \text{for } xs :: 'a \; \text{list} \; \text{and} \; ys\]
\[\text{using that by induction } (\text{auto simp: cr-dual-ord-def})\]
\[\text{hence } [\text{transfer-rule}]: (\text{list-all2 } (\text{pcr-dual-ord } (=))) \Rightarrow pcr-dual-ord (=) \text{ med med'}\]
\[\text{by } (\text{auto simp: rel-fun-def pcr-dual-ord-def OO-def med'-def cr-dual-ord-def dual-ord-to-dual-inverse})\]

\[\text{have } \text{size } \{ \# y \in \#. \; \text{mset } y. \; y > x\# \} = \text{length } (\text{filter } (\lambda y. \; y > x) \; xs)\]
\[\text{by } (\text{subst size-mset [symmetric]})(\text{simp only: mset-filter})\]
\[\text{also have } \ldots = \text{length } (\text{map to-dual-ord } (\text{filter } (\lambda y. \; y > x) \; xs)) \text{ by simp}\]
\[\text{also have } (\lambda y. \; y > x) = (\lambda y. \; \text{to-dual-ord } y < \text{to-dual-ord } x)\]
\[\text{by } \text{transfer simp-all}\]
\[\text{hence } \text{length } (\text{map to-dual-ord } (\text{filter } (\lambda y. \; y > x) \; xs)) = \text{length } (\text{map to-dual-ord } (\text{filter } \ldots \; xs))\]
\[\text{by simp}\]
\[\text{also have } \ldots = \text{length } (\text{filter } (\lambda y. \; y < \text{to-dual-ord } x) \; (\text{map to-dual-ord } xs))\]
\[\text{unfolding filter-map o-def by simp}\]
\[\text{also have } \ldots = \text{size } \{ \# y \in \#. \; \text{mset } \text{map to-dual-ord } xs. \; y < \text{to-dual-ord } x\#\}
\[\text{by } (\text{subst size-mset [symmetric]})(\text{simp only: mset-filter})\]
\[\text{also have } \ldots \leq \text{nat } (3 \ast \text{real } d + 1) / (2 \ast (2 \ast d + 1)) \ast \text{length } (\text{map to-dual-ord } xs)\]
\[+ 5 \ast d \; \text{div } 2 + 1\]

**Proof** (intro size-less-than-median-of-medians)
\[\text{fix } xs :: 'a \; \text{dual-ord list assume } xs : xs \neq [] \Rightarrow \text{length } xs \leq \text{Suc } (2 \ast d)\]
\[\text{from } xs \text{ show is-median } (\text{med' } xs) \; xs \text{ by } (\text{transfer fixing: } d) \; (\text{rule median})\]
\[\text{next}\]
\[\text{show is-median } (\text{to-dual-ord } x) \; (\text{map med'} (\text{chop } \text{Suc } (2 \ast d)) \; (\text{map to-dual-ord } xs)))\]
\[\text{by } (\text{transfer fixing: } d \; xs \; (\text{use median' in simp-all}))\]
\[\text{qed}\]
\[\text{finally show } ?\text{thesis by } (\text{simp add: n-def c-def})\]
\[\text{qed}\]

The most important case is that of chopping size 5, since that is the most
practical one for the median-of-medians selection algorithm. For it, we obtain the following nice and simple bounds:

**corollary** size-less-greater-median-of-medians-5:
fixes $xs :: 'a :: linorder list$
assumes $\forall xs. xs \neq [] \implies \text{length } xs \leq 5 \implies \text{is-median } (\text{med } xs) \; xs$
assumes $\text{is-median } x$ ($\text{map med } (\text{chop 5 } xs)$)
shows $\text{length } (\text{filter } (\lambda y. y < x) \; xs) \leq \text{nat } [0.7 \times \text{length } xs] + 6$
and $\text{length } (\text{filter } (\lambda y. y > x) \; xs) \leq \text{nat } [0.7 \times \text{length } xs] + 6$
using size-less-than-median-of-medians[of 2 med $x$ $xs$]
size-greater-than-median-of-medians[of 2 med $x$ $xs$] assms
by (simp-all add: size-mset [symmetric] mset-filter mult-ac add-ac del: size-mset)

1.7 The recursive step

We now turn to the actual selection algorithm itself. The following simple reduction lemma illustrates the idea of the algorithm quite well already, but it has the disadvantage that, if one were to use it as a recursive algorithm, it would only work for lists with distinct elements. If the list contains repeated elements, this may not even terminate.

The basic idea is that we choose some pivot element, partition the list into elements that are bigger than the pivot and those that are not, and then recurse into one of these (hopefully smaller) lists.

**theorem** select-rec-partition:
assumes $d > 0 \; k < \text{length } xs$
shows select $k \; xs = (\text{let } (ys, zs) = \text{partition } (\lambda y. y \leq x) \; xs$
in $\text{if } k < \text{length } ys \text{ then select } k \; ys \text{ else select } (k - \text{length } ys) \; zs$
) (is = ?rhs)

**proof**
- define $ys \; zs$ where $ys = \text{filter } (\lambda y. y \leq x) \; xs$ and $zs = \text{filter } (\lambda y. \neg(y \leq x)) \; xs$
have select $k \; xs = \text{select } k \; (ys @ zs)$
  by (intro select-cong) (simp-all add: ys-def zs-def)
also have $\ldots = (\text{if } k < \text{length } ys \text{ then select } k \; ys \text{ else select } (k - \text{length } ys) \; zs)$
using assms(2) by (intro select-append') (auto simp: ys-def zs-def sum-length-filter-compl)
finally show $?thesis$ by (simp add: ys-def zs-def Let-def o-def)
qed

The following variant uses a three-way partitioning function instead. This way, the size of the list in the final recursive call decreases by a factor of at least $\frac{3d' + 1}{2d' + 1}$ by the previous estimates, given that the chopping size is $d = 2d' + 1$. For a chopping size of 5, we get a factor of 0.7.

**definition** threeway-partition :: $'a => 'a :: linorder list => 'a list \times 'a list \times 'a list$
where
threeway-partition $x \; xs = (\text{filter } (\lambda y. y < x) \; xs, \; \text{filter } (\lambda y. y = x) \; xs, \; \text{filter } (\lambda y. y > x) \; xs)$
lemma threeway-partition-code [code]:

threeway-partition x [] = ([] , [], [])
threeway-partition x (y # ys) =
  (case threeway-partition x ys of (ls, es, gs) ⇒
    if y < x then (y # ls, es, gs) else if x = y then (ls, y # es, gs) else (ls, es, y # gs))
by (auto simp: threeway-partition-def)

theorem select-rec-threeway-partition:
assumes d > 0 k < length xs
shows select k xs = (let (ls, es, gs) = threeway-partition x xs;
  nl = length ls; ne = length es
in
  if k < nl then select k ls
  else if k < nl + ne then x
  else select (k - nl - ne) gs)
(is - = ?rhs)
proof
  define ls es gs where ls = filter (λy. y < x) xs and es = filter (λy. y = x) xs
  and gs = filter (λy. y > x) xs
  define nl ne where simp: nl = length ls ne = length es
  have mset-eq: mset xs = mset ls + mset es + mset gs unfolding ls-def es-def gs-def
   by (induction xs) auto
  have length-eq: length xs = length ls + length es + length gs unfolding ls-def es-def gs-def
   by (induction xs) auto
  have [simp]: select i es = x if i < length es for i
  proof
    have select i es ∈ set (sort es) unfolding select-def
      using that by (intro nth-mem) auto
    hence select i es ∈ set es using that by (auto simp: select-def)
    also have set es ⊆ {x} unfolding es-def by (induction es) auto
    finally show ?thesis by simp
  qed

  have select k xs = select k (ls @ (es @ gs))
   by (intro select-cong) (simp-all add: mset-eq)
  also have . . . = (if k < nl then select k ls else select (k - nl) (es @ gs))
unfolding nl-ne-def using assms
   by (intro select-append') (auto simp: ls-def es-def gs-def length-eq)
  also have . . . = (if k < nl then select k ls else if k < nl + ne then x
 else select (k - nl - ne) gs) (is ?lhs' = ?rhs')
proof (cases k < nl)
  case False
  hence ?lhs' = select (k - nl) (es @ gs) by simp
  also have . . . = (if k - nl < ne then select (k - nl) es else select (k - nl -
  ne) gs)
unfolding \textit{nl-ne-def} using \textit{assms False}

by (intro select-append') (auto simp: ls-def es-def gs-def length-eq)
also have \ldots = (if \( k - nl < ne \) then \( x \) else select \( (k - nl - ne) \) gs)
by simp
also from \textit{False} have \ldots = \textit{rhs'} by auto
finally show \textit{?thesis}.
qed simp-all
also have \ldots = \textit{rhs}
by (simp add: threeway-partition-def Let-def ls-def es-def gs-def)
finally show \textit{?thesis}.
qed

By the above results, it can be seen quite easily that, in each recursive step, the algorithm takes a list of length \( n \), does \( O(n) \) work for the chopping, computing the medians of the sublists, and partitioning, and it calls itself recursively with lists of size at most \( \lceil 0.2n \rceil \) and \( \lceil 0.7n \rceil + 6 \), respectively. This means that the runtime of the algorithm is bounded above by the Akra–Bazzi-style recurrence

\[ T(n) = T(\lceil 0.2n \rceil) + T(\lceil 0.7n \rceil + 6) + O(n) \]

which, by the Akra–Bazzi theorem, can be shown to fulfil \( T \in \Theta(n) \).

However, a proper analysis of this would require an actual execution model and some way of measuring the runtime of the algorithm, which is not what we aim to do here. Additionally, the entire algorithm can be performed in-place in an imperative way, but this because quite tedious.

Instead of this, we will now focus on developing the above recursion into an executable functional algorithm.

### 1.8 Medians of lists of length at most 5

We now show some basic results about how to efficiently find a median of a list of size at most 5. For length 1 or 2, this is trivial, since we can just pick any element. For length 3 and 4, we need at most three comparisons. For length 5, we need at most six comparisons.

This allows us to save some comparisons compared with the naive method of performing insertion sort and then returning the element in the middle.

definition \textit{median-3} :: \('a :: linorder ⇒ -\ where
\textit{median-3} \ a \ b \ c = 
(if \ a \leq \ b \ then
  (if \ b \leq \ c \ then \ b \ else \ max \ a \ c)
else
  (if \ c \leq \ b \ then \ b \ else \ min \ a \ c))

lemma \textit{median-3}: \textit{median-3} \ a \ b \ c = \textit{median} \ [a, b, c]
by (auto simp: \textit{median-3-def} \textit{median-def} \textit{select-def} \textit{min-def} \textit{max-def})
definition median-5-aux :: ′a :: linorder ⇒ - where
median-5-aux x1 x2 x3 x4 x5 = (  
if x2 ≤ x3 then if x2 ≤ x4 then min x3 x4 else min x2 x5  
else if x4 ≤ x3 then min x3 x5 else min x2 x4  )

lemma median-5-aux:  
assumes x1 ≤ x2 x4 ≤ x5 x1 ≤ x4  
shows median-5-aux x1 x2 x3 x4 x5 = median [x1,x2,x3,x4,x5]  
using assms by (auto simp: median-5-aux-def median-def select-def min-def)

definition median-5 :: ′a :: linorder ⇒ - where
median-5 a b c d e = (  
let (x1 , x2) = (if a ≤ b then (a, b) else (b, a));  
(x4 , x5) = (if d ≤ e then (d, e) else (e, d))  
in if x1 ≤ x4 then median-5-aux x1 x2 c x4 x5 else median-5-aux x4 x5 c x1 x2 )

lemma median-5: median-5 a b c d e = median [a, b, c, d, e]  
by (auto simp: median-5-def Let-def median-5-aux intro: median-cong)

fun median-le-5 where
median-le-5 [a] = a  
| median-le-5 [a,b] = a  
| median-le-5 [a,b,c] = median-3 a b c  
| median-le-5 [a,b,c,d] = median-3 a b c  
| median-le-5 [a,b,c,d,e] = median-5 a b c d e  
| median-le-5 - = undefined

lemma median-5-in-set: median-5 a b c d e ∈ {a, b, c, d, e}  
proof –  
  have median-5 a b c d e ∈ set [a, b, c, d, e]  
  unfolding median-5 in set [a, b, c, d, e]  
  thus ?thesis by simp
qed

lemma median-le-5-in-set:  
assumes xs ≠ [] length xs ≤ 5  
shows median-le-5 xs ∈ set xs  
proof (cases xs rule: median-le-5.cases)  
  case (5 a b c d e)  
  with median-5-in-set[of a b c d e] show ?thesis by simp
qed (insert assms, auto simp: median-3-def min-def max-def)

lemma median-le-5:  
assumes xs ≠ [] length xs ≤ 5  
shows is-median (median-le-5 xs) xs  
proof (cases xs rule: median-le-5.cases)
case (3 a b c)
have is-median (median xs) xs by simp
also have median xs = median-3 a b c by (simp add: median-3 3)
finally show ?thesis using 3 by simp
next
case (4 a b c d)
have is-median (median [a,b,c]) [a,b,c] by simp
also have median [a,b,c] = median-3 a b c by (simp add: median-3 4)
finally have is-median (median-3 a b c) (d ≠ [a,b,c]) by (rule is-median-Cons-odd)
also have ?this ←→ is-median (median-3 a b c) [a,b,c,d] by (intro is-median-cong)
auto
finally show ?thesis using 4 by simp
next
case (5 a b c d e)
have is-median (median xs) xs by simp
also have median xs = median-5 a b c d e by (simp add: median-5 5)
finally show ?thesis using 5 by simp
qed (insert assms, auto simp: is-median-def)

1.9 Median-of-medians selection algorithm

The fast selection function now simply computes the median-of-medians of
the chopped-up list as a pivot, partitions the list into with respect to that
pivot, and recurses into one of the resulting sublists.

function fast-select where
fast-select k xs = (if length xs ≤ 20 then sort xs ! k
else let x = fast-select (((length xs + 4) div 5 - 1) div 2) (map median-le-5
(chop 5 xs));
(ls, es, gs) = threeway-partition x xs
in
if k < length ls then fast-select k ls
else if k < length ls + length es then x
else fast-select (k - length ls - length es) gs
)
by auto

The correctness of this is obvious from the above theorems, but the proof
is still somewhat complicated by the fact that termination depends on the
correctness of the function.

lemma fast-select-correct-aux:
assumes fast-select-dom (k, xs) k < length xs
shows fast-select k xs = select k xs
using assms
proof induction
case (1 k xs)
show ?case
proof (cases length xs ≤ 20)
case True
thus ?thesis using 1.prems 1.hyps
  by (subst fast-select.psimps) (auto simp: select-def)
next
case False
define x where
  x = fast-select (((length xs + 4) div 5 - Suc 0) div 2) (map median-le-5 (chop 5 xs))
define ls where ls = filter (λy. y < x) xs
define es where es = filter (λy. y = x) xs
define gs where gs = filter (λy. y > x) xs
notedefs = nl-def ne-def x-def ls-def es-def gs-def
have tw': (ls, es, gs) = thrreeway-partition (fast-select (((length xs + 4) div 5 - 1) div 2)
  (map median-le-5 (chop 5 xs))) xs
  unfolding thrreeway-partition-def defs One-nat-def ..
have tw: (ls, es, gs) = thrreeway-partition x xs
  by (simp add: tw x-def)
also have ... = (if k < nl then select k ls else if k < nl + ne then x
    else fast-select (k - nl - ne) gs)
    using 1.hyps False
    by (subst fast-select.psimps) (simp-all add: thrreeway-partition-def defs [symmetric])
also have ... = (if k < nl then select k ls else if k < nl + ne then x
    else select (k - nl - ne) gs)
proof (intro if-cong refl)
assume *: k < nl
show fast-select k xs = (if k < nl then fast-select k ls else if k < nl + ne then x
  else fast-select (k - nl - ne) gs)
  using 1.hyps False
also have ... = (if k < nl then select k ls else if k < nl + ne then x
    else select (k - nl - ne) gs)
by (rule 1; (rule refl tw))? (insert *, auto simp: False thrreeway-partition-def ls-def x-def nl-def)+
next
assume *: ¬k < nl ¬¬k < nl + ne
have **: length xs = length ls + length es + length gs
  unfolding ls-def es-def gs-def by (induction xs) auto
show fast-select (k - nl - ne) gs = select (k - nl - ne) gs
  unfolding nl-def ne-def
  by (rule 1; (rule refl tw))? (insert False * ** (k < length xs), auto simp: nl-def ne-def)
qed
also have ... = select k xs using (k < length xs)
  by (subst (3) select-rec-threeway-partition[of 5::nat - - x])
  (unfold Let-def nl-def ne-def ls-def es-def x-def thrreeway-partition-def, simp-all)
finally show ?thesis .
qed
qed
qed
Termination of the algorithm is reasonably obvious because the lists that are recursed into never contain the pivot (the median-of-median), while the original list clearly does. The proof is still somewhat technical though.

**lemmas**: fast-select-termination: All fast-select-dom

**proof**

(relation measure (length o snd); (safe)?, goal-cases)

**case** (1 k xs)

thus ?case

by (auto simp: length-chop nat-less-iff ceiling-less-iff)

**next**

fix k :: nat and xs ls es gs :: 'a list

define x where x = fast-select (((length xs + 4) div 5 - 1) div 2) (map median-le-5 (chop 5 xs))

assume A: - length xs ≤ 20

(ls, es, gs) = threeway-partition x xs

fast-select-dom (((length xs + 4) div 5 - 1) div 2, map median-le-5 (chop 5 xs))

from A have eq: ls = filter (λy. y < x) xs gs = filter (λy. y > x) xs

by (simp-all add: x-def threeway-partition-def)

have len: (length xs + 4) div 5 = nat [(length xs / 5)] by linarith

have less: (nat [real (length xs) / 5] - Suc 0) div 2 < nat [real (length xs) / 5]

using A(1) by linarith

have x = select (((length xs + 4) div 5 - 1) div 2) (map median-le-5 (chop 5 xs))

using less unfolding x-def by (intro fast-select-correct-aux A) (auto simp: length-chop len)

also have ... = median (map median-le-5 (chop 5 xs)) by (simp add: median-def len length-chop)

finally have x: x = ...

moreover {

have x ∈ set (map median-le-5 (chop 5 xs))

using A(1) unfolding x by (intro median-in-set) auto

also have ... ⊆ (∪ys∈set (chop 5 xs). {median-le-5 ys}) by auto

also have ... ⊆ (∪ys∈set (chop 5 xs). set ys) using A(1)

by (intro UN-mono) (auto simp: median-le-5-in-set length-chop-part-le)

also have ... = set xs by (subst UN-sets-chop) auto

finally have x ∈ set xs .
}

ultimately show ((k, ls), k, xs) ∈ measure (length o snd)

and ((k - length ls - length es, gs), k, xs) ∈ measure (length o snd)

using A(1) by (auto simp: eq intro!: length-filter-less[of x])

qed

We now have all the ingredients to show that fast-select terminates and does, indeed, compute the k-th order statistic.

**termination**: fast-select by (rule fast-select-termination)

**theorem**: fast-select-correct: k < length xs ⟹ fast-select k xs = select k xs

using fast-select termination by (intro fast-select-correct-aux) auto
The following version is then suitable for code export.

**lemma** fast-select-code [code]:

\[
\text{fast-select } k \text{ xs } = \begin{cases} 
\text{fold insert xs [] ! k} \\
\text{let } x = \text{fast-select } ((\lfloor (\text{length xs } + 4) \div 5 - 1 \rfloor \div 2) \ (\text{map median-le-5} \ (\text{chop } 5 \text{ xs})));
\text{(ls, es, gs) } = \text{threeway-partition } x \text{ xs};
\text{nl } = \text{length ls}; \text{ne } = \text{nl } + \text{length es}
\text{in }
\text{if } k < \text{nl } \text{then fast-select } k \text{ ls}
\text{else if } k < \text{ne } \text{then } x
\text{else fast-select } (k - \text{ne}) \text{ gs}
\end{cases}
\]

by (subst fast-select.simps) (simp-all only: Let-def algebra-simps sort-conv-fold)

**lemma** select-code [code]:

\[
\text{select } k \text{ xs } = \begin{cases} 
\text{if } k < \text{length xs then fast-select } k \text{ xs} \\
\text{else Code.abort } (\text{STR "Selection index out of bounds."})\end{cases}
\]

proof (cases k < length xs)

  case True
  thus ?thesis by (simp only: if-True fast-select-correct)

  qed (simp-all only: Code.abort-def if-False)

end

**References**