The Median-Of-Medians Selection Algorithm

Manuel Eberl

March 17, 2025

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

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* ($\lambda x.$ 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 ($\bigwedge x.$ $x \in\#$ $A \Longrightarrow f\ x \subseteq\#\ g\ x$)
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 $A \subseteq\#\ B$ $\bigwedge x.$ $x \in\#\ A \Longrightarrow P\ x \Longrightarrow Q\ x$
shows *filter-mset* P A $\subseteq\#$ *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-mset* A :: *'a multiset*) = *sum-mset* (*image-mset size* A)
by (*induction* A) *auto*

lemma *sum-mset-mono*:
assumes $\bigwedge x.$ $x \in\#\ A \Longrightarrow f\ x \le (g\ x$ :: *'a* :: {*ordered-ab-semigroup-add*,*comm-monoid-add*})
shows ($\sum x{\in}\#A.\ f\ x$) $\le$ ($\sum x{\in}\#A.\ g\ x$)
using *assms* **by** (*induction* A) (*auto intro*!: *add-mono*)

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

lemma *sorted-filter-less-subset-take*:
assumes *sorted xs* i < *length xs*
shows {# $x \in\#$ *mset xs*. x < *xs* ! i #} $\subseteq\#$ *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 {#$y \in\#$ *mset* (x # *xs*). y < (x # *xs*) ! i#} $\subseteq\#$ *add-mset* x {#$y \in\#$ *mset xs*. y < *xs* ! i'#}
using *Suc Cons.prems* **by** (*auto*)
also have ... $\subseteq\#$ *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 ... = *mset* (*take* i (x # *xs*)) **by** (*simp add*: *Suc*)

```
    finally show ?thesis .
  qed
qed auto

lemma sorted-filter-greater-subset-drop:
  assumes sorted xs i < length xs
  shows   {# x ∈# mset xs. x > xs ! i #} ⊆# mset (drop (Suc 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 by (auto simp: sorted-append filter-mset-is-empty-iff)
  next
    case (Suc i')
    have {#y ∈# mset (x # xs). y > (x # xs) ! i#} ⊆# {#y ∈# mset xs. y >
xs ! i'#}
      using Suc Cons.prems by (auto simp: set-conv-nth)
    also have ... ⊆# mset (drop (Suc i') xs)
      using Cons.prems Suc by (intro Cons.IH) (auto)
    also have ... = mset (drop (Suc i) (x # xs)) by (simp add: Suc)
    finally show ?thesis .
  qed
qed auto
```

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.

```
typedef 'a dual-ord = UNIV :: 'a set morphisms of-dual-ord to-dual-ord
  by auto

setup-lifting type-definition-dual-ord

instantiation dual-ord :: (ord) ord
begin

lift-definition less-eq-dual-ord :: 'a dual-ord ⇒ 'a dual-ord ⇒ bool is
  λa b :: 'a. a ≥ b .

lift-definition less-dual-ord :: 'a dual-ord ⇒ 'a dual-ord ⇒ bool is
  λa b :: 'a. a > b .

instance ..
end

instance dual-ord :: (preorder) preorder
```

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*]

by *simp*

lemma *length-chop-part-le*: *ys* $\in$ *set* (*chop n xs*) $\Longrightarrow$ *length ys* $\leq$ *n*
by (*induction n xs rule*: *chop.induct*) *auto*

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

lemma *length-chop*:
assumes *n* $>$ *0*
shows *length* (*chop n xs*) = *nat* $\lceil$*length xs* / *n*$\rceil$
using *assms*
proof (*induction n xs rule*: *chop.induct*)
case (*3 n xs*)
show *?case*
proof (*cases length xs* $\geq$ *n*)
case *False*
hence $\lceil$*real* (*length xs*) / *real n*$\rceil$ = *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 xs ! k ∈ set (sort xs) by (intro nth-mem) auto
```

also have *set* (*sort xs*) = *set xs* **by** *simp*
finally show *?thesis* **by** (*simp add*: *select-def*)
qed

lemma *median-in-set* [*intro*, *simp*]:
assumes $xs \neq []$
shows *median xs* ∈ *set xs*
proof −
from *assms* **have** *length xs* > *0* **by** *auto*
hence (*length xs* − *1*) *div 2* < *length xs* **by** *linarith*
thus *?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*: *mset xs* = *mset ys* ⟹ *sort xs* = *sort ys*
by (*rule properties-for-sort*) *simp-all*

lemma *select-cong*:
k = *k′* ⟹ *mset xs* = *mset xs′* ⟹ *select k xs* = *select k′ xs′*
by (*auto simp*: *select-def dest*: *sort-cong*)

lemma *median-cong*: *mset xs* = *mset xs′* ⟹ *median xs* = *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 ⋀*x y*. *x* ∈ *set xs* ⟹ *y* ∈ *set ys* ⟹ *x* ≤ *y*
shows *sort* (*xs* @ *ys*) = *sort xs* @ *sort ys*
using *assms* **by** (*intro properties-for-sort*) (*auto simp*: *sorted-append*)

lemma *select-append*:
assumes ⋀*y z*. *y* ∈ *set ys* ⟹ *z* ∈ *set zs* ⟹ *y* ≤ *z*
shows *k* < *length ys* ⟹ *select k* (*ys* @ *zs*) = *select k ys*
k ∈ {*length ys*..<*length ys* + *length zs*} ⟹
select k (*ys* @ *zs*) = *select* (*k* − *length ys*) *zs*
using *assms* **by** (*simp-all add*: *select-def sort-append nth-append*)

lemma *select-append′*:
assumes ⋀*y z*. *y* ∈ *set ys* ⟹ *z* ∈ *set zs* ⟹ *y* ≤ *z* *k* < *length ys* + *length zs*
shows *select k* (*ys* @ *zs*) = (*if k* < *length ys then select k ys else select* (*k* − *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*:
size {#*y* ∈# *mset xs*. *y* < *median xs*#} ≤ (*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*
    by (*intro size-mset-mono sorted-filter-less-subset-take*) *auto*
  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* (λ*y*. *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* (λ*y*. *y* < *x*) *xs*) ≤ *length xs div 2* ∧
                    length (*filter* (λ*y*. *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.

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* (*λx y. x* = *f y*) *xs ys* ⟷ *xs* = *map f ys*
proof
  assume *list-all2* (*λx y. x* = *f y*) *xs ys*
  thus *xs* = *map f ys* **by** *induction auto*
next
  assume *xs* = *map f ys*
  moreover have *list-all2* (*λx y. x* = *f y*) (*map f ys*) *ys*
    by (*induction ys*) *auto*
  ultimately show *list-all2* (*λx y. x* = *f y*) *xs ys* **by** *simp*
qed

lemma *transfer-is-median-dual-ord* [*transfer-rule*]:
  (*pcr-dual-ord* (=) ===> *list-all2* (*pcr-dual-ord* (=)) ===> (=)) *is-median 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 (*to-dual-ord x*) (*map to-dual-ord xs*) ⟷ *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 ⟷ *size* (*filter-mset* (*λy. y* < *x*) (*mset xs*)) ≤ *length xs div 2* ∧
          size (*filter-mset* (*λy. y* > *x*) (*mset xs*)) ≤ *length xs div 2*
proof −
  have ∗: *length* (*filter P xs*) = *size* (*filter-mset P* (*mset xs*)) **for** *P* **and** *xs* :: *'a list*
    by (*simp flip*: *mset-filter*)
  show *?thesis* **by** (*simp only*: *is-median-def* ∗)
qed

lemma *is-median-cong*:
  assumes *x* = *y mset xs* = *mset ys*
  shows *is-median x xs* ⟷ *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 *is-median-median* [*simp*,*intro*]: *is-median* (*median xs*) *xs*
  using *size-less-than-median*[*of xs*] *size-greater-than-median*[*of xs*]
  unfolding *is-median-def size-mset* [*symmetric*] *mset-filter* **by** *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 *size-less-than-median-of-medians-strong*:
  fixes $xs :: {'a} :: linorder\ list$ **and** $d :: nat$
  assumes d: $d > 0$
  assumes *median*: $\bigwedge xs.\ xs \neq [] \Longrightarrow length\ xs \leq d \Longrightarrow is\text{-}median\ (med\ xs)\ xs$
  assumes *median'*: $is\text{-}median\ x\ (map\ med\ (chop\ d\ xs))$
  defines $m \equiv length\ (chop\ d\ xs)$
  shows $size\ \{\#y \in\#\ mset\ xs.\ y < x\#\} \leq m * (d\ div\ 2) + m\ div\ 2 * ((d + 1)\ div\ 2)$
proof −
  define n **where** [*simp*]: $n = length\ xs$
  — The medians of the sublists
  define M **where** $M = mset\ (map\ med\ (chop\ d\ xs))$
  define YS **where** $YS = mset\ (chop\ d\ xs)$
  — The sublists with a smaller median than the median-of-medians x and the rest.
  define $YS1$ **where** $YS1 = filter\text{-}mset\ (\lambda ys.\ med\ ys < x)\ (mset\ (chop\ d\ xs))$
  define $YS2$ **where** $YS2 = filter\text{-}mset\ (\lambda ys.\ \neg(med\ ys < x))\ (mset\ (chop\ d\ xs))$

  — At most roughly half of the lists have a median that is smaller than M
  have $size\ YS1 = size\ (image\text{-}mset\ med\ YS1)$ **by** *simp*
  also have $image\text{-}mset\ med\ YS1 = \{\#y \in\#\ mset\ (map\ med\ (chop\ d\ xs)).\ y < x\#\}$
    unfolding *YS1-def* **by** (*subst filter-mset-image-mset* [*symmetric*]) *simp-all*
  also have $size\ \ldots \leq (length\ (map\ med\ (chop\ d\ xs)))\ div\ 2$
    using *median'* **unfolding** *is-median-altdef* **by** *simp*
  also have $\ldots = m\ div\ 2$ **by** (*simp add*: *m-def*)
  finally have *size-YS1*: $size\ YS1 \leq m\ div\ 2$ **.**

  have $m = size\ (mset\ (chop\ d\ xs))$ **by** (*simp add*: *m-def*)
  also have $mset\ (chop\ d\ xs) = YS1 + YS2$ **unfolding** *YS1-def YS2-def*
    by (*rule multiset-partition*)
  finally have *m-eq*: $m = size\ YS1 + size\ YS2$ **by** *simp*

  — 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* ∈# *mset xs*. *y* < *x*#} = {# *y* ∈# (∑ *ys*←*chop d xs*. *mset ys*). *y* < *x*#} **using** *d*
  by (*subst sum-msets-chop*) *simp-all*
also have ... = (∑ *ys*←*chop d xs*. {#*y* ∈# *mset ys*. *y* < *x*#})
  by (*subst filter-mset-sum-list*) (*simp add*: *o-def*)
also have ... = (∑ *ys*∈#*YS*. {#*y* ∈# *mset ys*. *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*)
also have (∑ *ys*∈#.... {#*y* ∈# *mset ys*. *y* < *x*#}) =
    (∑ *ys*∈#*YS1*. {#*y* ∈# *mset ys*. *y* < *x*#}) + (∑ *ys*∈#*YS2*. {#*y* ∈# *mset ys*. *y* < *x*#})
  by *simp*
also have ... ⊆# (∑ *ys*∈#*YS1*. *mset ys*) + (∑ *ys*∈#*YS2*. {#*y* ∈# *mset ys*. *y* < *med ys*#})
  by (*intro subset-mset.add-mono sum-mset-mset-mono mset-filter-mono*) (*auto simp*: *YS2-def*)
finally have {# *y* ∈# *mset xs*. *y* < *x* #} ⊆# ... .
hence *size* {# *y* ∈# *mset xs*. *y* < *x* #} ≤ *size* ... **by** (*rule size-mset-mono*)

— We do some further straightforward estimations and arrive at our goal.
also have ... = (∑ *ys*∈#*YS1*. *length ys*) + (∑ *x*∈#*YS2*. *size* {#*y* ∈# *mset x*. *y* < *med x*#})
  by (*simp add*: *size-mset-sum-mset-distrib multiset.map-comp o-def*)
also have (∑ *ys*∈#*YS1*. *length ys*) ≤ (∑ *ys*∈#*YS1*. *d*)
  by (*intro sum-mset-mono*) (*auto simp*: *YS1-def length-chop-part-le*)
also have ... = *size YS1* ∗ *d* **by** *simp*
also have *d*: *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 (∑ *ys*∈#*YS2*. *size* {#*y* ∈# *mset ys*. *y* < *med ys*#}) ≤
    (∑ *ys*∈#*YS2*. *length ys div 2*)
proof (*intro sum-mset-mono size-less-than-median*, *goal-cases*)
  case (*1 ys*)
  hence *ys* ≠ [] *length ys* ≤ *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 ... ≤ (∑ *ys*∈#*YS2*. *d div 2*)
  by (*intro sum-mset-mono div-le-mono diff-le-mono*) (*auto simp*: *YS2-def dest*: *length-chop-part-le*)
also have ... = *size YS2* ∗ (*d div 2*) **by** *simp*
also have *size YS1* ∗ (*d div 2*) + *size YS1* ∗ ((*d* + *1*) *div 2*) + ... =
    m ∗ (*d div 2*) + *size YS1* ∗ ((*d* + *1*) *div 2*) **by** (*simp add*: *m-eq algebra-simps*)
also have *size YS1* ∗ ((*d* + *1*) *div 2*) ≤ (*m div 2*) ∗ ((*d* + *1*) *div 2*)

by (*intro mult-right-mono size-YS1*) *auto*
finally show *size* {#*y* ∈# *mset xs*. *y* < *x*#} ≤
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* :: *linorder list* **and** *d* :: *nat*
assumes *median*: ⋀*xs*. *xs* ≠ [] ⟹ *length xs* ≤ *Suc* (*2* ∗ *d*) ⟹ *is-median* (*med xs*) *xs*
assumes *median*′: *is-median x* (*map med* (*chop* (*Suc* (*2*∗*d*)) *xs*))
defines *n* ≡ *length xs*
defines *c* ≡ (*3* ∗ *real d* + *1*) / (*2* ∗ (*2* ∗ *d* + *1*))
shows *size* {#*y* ∈# *mset xs*. *y* < *x*#} ≤ *nat* ⌈*c* ∗ *n*⌉ + (*5* ∗ *d*) *div 2* + *1*
proof (*cases xs* = [])
case *False*
define *m* **where** *m* = *length* (*chop* (*Suc* (*2*∗*d*)) *xs*)

have *real* (*m div 2*) ≤ *real* (*nat* ⌈*real n* / (*1* + *2* ∗ *real d*)⌉) / *2*
by (*simp add*: *m-def length-chop n-def flip*: *of-nat-int-ceiling*)
also have *real* (*nat* ⌈*real n* / (*1* + *2* ∗ *real d*)⌉) =
of-int ⌈*real n* / (*1* + *2* ∗ *real d*)⌉
by (*intro of-nat-nat*) (*auto simp*: *divide-simps*)
also have . . . / *2* ≤ (*real n* / (*1* + *2* ∗ *real d*) + *1*) / *2*
by (*intro divide-right-mono*) *linarith*+
also have . . . = *n* / (*2* ∗ (*2* ∗ *real d* + *1*)) + *1* / *2* **by** (*simp add*: *field-simps*)
finally have *m*: *real* (*m div 2*) ≤ . . . **.**

have *size* {#*y* ∈# *mset xs*. *y* < *x*#} ≤ *d* ∗ *m* + *Suc d* ∗ (*m div 2*)
using *size-less-than-median-of-medians-strong*[*of Suc* (*2* ∗ *d*) *med x xs*] *assms*
unfolding *m-def* **by** (*simp add*: *algebra-simps*)
also have . . . ≤ *d* ∗ (*2* ∗ (*m div 2*) + *1*) + *Suc d* ∗ (*m div 2*)
by (*intro add-mono mult-left-mono*) *linarith*+
also have . . . = (*3* ∗ *d* + *1*) ∗ (*m div 2*) + *d*
by (*simp add*: *algebra-simps*)
finally have *real* (*size* {#*y* ∈# *mset xs*. *y* < *x*#}) ≤ *real* . . .
by (*subst of-nat-le-iff*)
also have . . . ≤ (*3* ∗ *real d* + *1*) ∗ (*n* / (*2* ∗ (*2* ∗ *d* + *1*)) + *1*/*2*) + *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 . . . = *c* ∗ *real n* + (*5* ∗ *real d* + *1*) / *2*
by (*simp add*: *field-simps c-def*)
also have . . . ≤ *real* (*nat* ⌈*c* ∗ *n*⌉ + ((*5* ∗ *d*) *div 2* + *1*))
unfolding *of-nat-add* **by** (*intro add-mono*) (*linarith*, *simp add*: *field-simps*)
finally show *?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:
  fixes xs :: 'a :: linorder list and d :: nat
  assumes median: ⋀xs. xs ≠ [] ⟹ length xs ≤ Suc (2 ∗ d) ⟹ is-median (med xs) xs
  assumes median': is-median x (map med (chop (Suc (2∗d)) xs))
  defines n ≡ length xs
  defines c ≡ (3 ∗ real d + 1) / (2 ∗ (2 ∗ d + 1))
  shows   size {#y ∈# mset xs. y > x#} ≤ nat ⌈c ∗ n⌉ + (5 ∗ d) div 2 + 1
proof −
  include lifting-syntax
  define med' where med' = (λxs. to-dual-ord (med (map of-dual-ord xs)))
  have xs = map of-dual-ord ys if list-all2 cr-dual-ord xs ys for xs :: 'a list and ys
    using that by induction (auto simp: cr-dual-ord-def)
  hence [transfer-rule]: (list-all2 (pcr-dual-ord (=)) ===> pcr-dual-ord (=)) med med'
    by (auto simp: rel-fun-def pcr-dual-ord-def OO-def med'-def cr-dual-ord-def
              dual-ord.to-dual-ord-inverse)

  have size {#y ∈# mset xs. y > x#} = length (filter (λy. y > x) xs)
    by (subst size-mset [symmetric]) (simp only: mset-filter)
  also have ... = length (map to-dual-ord (filter (λy. y > x) xs)) by simp
  also have (λy. y > x) = (λy. to-dual-ord y < to-dual-ord x)
    by transfer simp-all
  hence length (map to-dual-ord (filter (λy. y > x) xs)) = length (map to-dual-ord (filter ... xs))
    by simp
  also have ... = length (filter (λy. y < to-dual-ord x) (map to-dual-ord xs))
    unfolding filter-map o-def by simp
  also have ... = size {#y ∈# mset (map to-dual-ord xs). y < to-dual-ord x#}
    by (subst size-mset [symmetric]) (simp only: mset-filter)
  also have ... ≤ nat ⌈(3 ∗ real d + 1) / real (2 ∗ (2 ∗ d + 1)) ∗ length (map to-dual-ord xs)⌉
                + 5 ∗ d div 2 + 1
  proof (intro size-less-than-median-of-medians)
    fix xs :: 'a dual-ord list assume xs: xs ≠ [] length xs ≤ Suc (2 ∗ d)
    from xs show is-median (med' xs) xs by (transfer fixing: d) (rule median)
  next
   show is-median (to-dual-ord x) (map med' (chop (Suc (2 ∗ d)) (map to-dual-ord xs)))
      by (transfer fixing: d x xs) (use median' in simp-all)
  qed
  finally show ?thesis by (simp add: n-def c-def)
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 $\bigwedge xs.\ xs \neq [] \Longrightarrow length\ xs \leq 5 \Longrightarrow is\text{-}median\ (med\ xs)\ xs$
  assumes $is\text{-}median\ x\ (map\ med\ (chop\ 5\ xs))$
  shows $length\ (filter\ (\lambda y.\ y < x)\ xs) \leq nat\ \lceil 0.7 * length\ xs \rceil + 6$
    and $length\ (filter\ (\lambda y.\ y > x)\ xs) \leq nat\ \lceil 0.7 * length\ xs \rceil + 6$
  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 $d > 0\ k < length\ xs$
  shows $select\ k\ xs = ($
          $let\ (ys, zs) = partition\ (\lambda y.\ y \leq x)\ xs$
          $in\ \ if\ k < length\ ys\ then\ select\ k\ ys\ else\ select\ (k - length\ ys)\ zs$
) (**is** - = *?rhs*)
proof −
  define *ys zs* **where** $ys = filter\ (\lambda y.\ y \leq x)\ xs$ **and** $zs = filter\ (\lambda y.\ \neg(y \leq x))\ xs$
  have $select\ k\ xs = select\ k\ (ys\ @\ zs)$
    by (*intro select-cong*) (*simp-all add*: *ys-def zs-def*)
  also have $\ldots = (if\ k < length\ ys\ then\ select\ k\ ys\ else\ select\ (k - length\ ys)\ zs)$
   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 $\frac{3d'+1}{2(2d'+1)}$ 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* :: $'a \Rightarrow 'a$ :: *linorder list* $\Rightarrow 'a\ list \times 'a\ list \times 'a\ list$
where
  $threeway\text{-}partition\ x\ xs = (filter\ (\lambda y.\ y < x)\ xs, filter\ (\lambda y.\ y = x)\ xs, filter\ (\lambda y.\ y > x)\ xs)$

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 simp del: filter-True)

  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 nl-ne-def using assms False
      by (intro select-append') (auto simp: ls-def es-def gs-def length-eq)
    also have ... = (if k − nl < ne then x else select (k − nl − ne) gs)
      by simp
    also from False have ... = ?rhs' by auto
    finally show ?thesis .
  qed simp-all
  also have ... = ?rhs
    by (simp add: threeway-partition-def Let-def ls-def es-def gs-def)
  finally show ?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 median-3 :: 'a :: linorder ⇒ - where
  median-3 a b c =
    (if a ≤ b then
       if b ≤ c then b else max a c
     else
       if c ≤ b then b else min a c)

lemma median-3: median-3 a b c = median [a, b, c]
  by (auto simp: median-3-def median-def select-def min-def 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* **by** (*rule median-in-set*) *auto*
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)
auto
 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
    define nl ne where nl = length ls and ne = length es
    note defs = nl-def ne-def x-def ls-def es-def gs-def
    have tw: (ls, es, gs) = threeway-partition (fast-select (((length xs + 4) div 5 −
1) div 2)
                           (map median-le-5 (chop 5 xs))) xs
      unfolding threeway-partition-def defs One-nat-def ..
    have tw': (ls, es, gs) = threeway-partition x xs
      by (simp add: tw x-def)

    have 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
     by (subst fast-select.psimps) (simp-all add: threeway-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 ls = select k ls
        by (rule 1; (rule refl tw)?)
           (insert ∗, auto simp: False threeway-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 simp del: filter-True)
      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 gs-def es-def x-def threeway-partition-def,
simp-all)
    finally show ?thesis .
  qed
qed
```

Termination of the algorithm is reasonably obvious because the lists that are recursed into never contain the pivot (the median-of-medians), while the original list clearly does. The proof is still somewhat technical though.

lemma *fast-select-termination*: *All fast-select-dom*
proof (*relation measure* (*length* ∘ *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* (λ*y*. *y* < *x*) *xs gs* = *filter* (λ*y*. *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* ∘ *snd*)
      and ((*k* − *length ls* − *length es*, *gs*), *k*, *xs*) ∈ *measure* (*length* ∘ *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]:
  fast-select k xs = (
    if length xs ≤ 20 then
      fold insort 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;
            nl = length ls; ne = nl + length es
        in
          if k < nl then fast-select k ls
          else if k < ne then x
          else fast-select (k − ne) gs
      )
  by (subst fast-select.simps) (simp-all only: Let-def algebra-simps sort-conv-fold)

lemma select-code [code]:
  select k xs = (if k < length xs then fast-select k xs
                 else Code.abort (STR ''Selection index out of bounds.'') (λ-. select
k xs))
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

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.