The Median-Of-Medians Selection Algorithm

Manuel Eberl

December 14, 2021

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*)
⟨*proof*⟩

lemma *set-mset-sum-list* [*simp*]: *set-mset* (*sum-list xs*) = ($\bigcup$*x*∈*set xs*. *set-mset x*)
⟨*proof*⟩

lemma *filter-mset-image-mset*:

filter-mset P (*image-mset* f A) = *image-mset* f (*filter-mset* ($\lambda x.$ P (f x)) A)
⟨*proof*⟩

lemma *filter-mset-sum-list*: *filter-mset* P (*sum-list* *xs*) = *sum-list* (*map* (*filter-mset* P) *xs*)
⟨*proof*⟩

lemma *sum-mset-mset-mono*:
assumes ($\bigwedge x.$ $x \in\# A \Longrightarrow f\ x \subseteq\# g\ x$)
shows ($\sum x{\in}\#A.$ $f\ x$) $\subseteq\#$ ($\sum x{\in}\#A.$ $g\ x$)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *size-mset-sum-mset-distrib*: *size* (*sum-mset* A :: *'a multiset*) = *sum-mset* (*image-mset* *size* A)
⟨*proof*⟩

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 ($\sum x{\in}\#A.$ $f\ x$) $\le$ ($\sum x{\in}\#A.$ $g\ x$)
⟨*proof*⟩

lemma *filter-mset-is-empty-iff*: *filter-mset* P A = {#} $\longleftrightarrow$ ($\forall x.$ $x \in\# A \longrightarrow \neg P$ x)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *sorted-filter-greater-subset-drop*:
assumes *sorted* *xs* $i <$ *length* *xs*
shows {# $x \in\#$ *mset* *xs*. $x >$ *xs* ! i #} $\subseteq\#$ *mset* (*drop* (*Suc* i) *xs*)
⟨*proof*⟩

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*
⟨*proof*⟩

setup-lifting *type-definition-dual-ord*

instantiation *dual-ord* :: (*ord*) *ord*
begin

lift-definition *less-eq-dual-ord* :: $'a$ *dual-ord* $\Rightarrow$ $'a$ *dual-ord* $\Rightarrow$ *bool* **is**
$\lambda a\ b :: 'a.\ a \geq b$ ⟨*proof*⟩

lift-definition *less-dual-ord* :: $'a$ *dual-ord* $\Rightarrow$ $'a$ *dual-ord* $\Rightarrow$ *bool* **is**
$\lambda a\ b :: 'a.\ a > b$ ⟨*proof*⟩

instance ⟨*proof*⟩
end

instance *dual-ord* :: (*preorder*) *preorder*
⟨*proof*⟩

instance *dual-ord* :: (*linorder*) *linorder*
⟨*proof*⟩

1.3 Chopping a list into equal-sized sublists

function *chop* :: *nat* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *list list* **where**
chop n [] = []
| *chop* 0 *xs* = []
| $n > 0 \Longrightarrow xs \neq [] \Longrightarrow$ *chop* n *xs* = *take* n *xs* # *chop* n (*drop* n *xs*)
⟨*proof*⟩
termination ⟨*proof*⟩

context
includes *lifting-syntax*
begin

lemma *chop-transfer* [*transfer-rule*]:
((=) ===> *list-all2* R ===> *list-all2* (*list-all2* R)) *chop chop*
⟨*proof*⟩

end

lemma *chop-reduce*: *chop* n *xs* = (*if* $n = 0 \lor xs = []$ *then* [] *else take* n *xs* # *chop* n (*drop* n *xs*))
⟨*proof*⟩

lemma *concat-chop* [*simp*]: $n > 0 \Longrightarrow$ *concat* (*chop* n *xs*) = *xs*
⟨*proof*⟩

lemma *chop-elem-not-Nil* [*simp,dest*]: $ys \in$ *set* (*chop* n *xs*) $\Longrightarrow ys \neq []$
⟨*proof*⟩

lemma *chop-eq-Nil-iff* [*simp*]: *chop* n *xs* = [] $\longleftrightarrow n = 0 \lor xs = []$

⟨*proof*⟩

lemma *chop-ge-length-eq*: $n > 0 \Longrightarrow xs \neq [] \Longrightarrow n \geq length\ xs \Longrightarrow chop\ n\ xs = [xs]$
⟨*proof*⟩

lemma *length-chop-part-le*: $ys \in set\ (chop\ n\ xs) \Longrightarrow length\ ys \leq n$
⟨*proof*⟩

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*⟩

lemma *length-chop*:
assumes $n > 0$
shows $length\ (chop\ n\ xs) = nat\ \lceil length\ xs\ /\ n \rceil$
⟨*proof*⟩

lemma *sum-msets-chop*: $n > 0 \Longrightarrow (\sum ys \leftarrow chop\ n\ xs.\ mset\ ys) = mset\ xs$
⟨*proof*⟩

lemma *UN-sets-chop*: $n > 0 \Longrightarrow (\bigcup ys \in set\ (chop\ n\ xs).\ set\ ys) = set\ xs$
⟨*proof*⟩

lemma *in-set-chopD* [*dest*]:
assumes $x \in set\ ys\ ys \in set\ (chop\ d\ xs)$
shows $x \in set\ xs$
⟨*proof*⟩

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* :: $nat \Rightarrow 'a\ list \Rightarrow ('a :: linorder)$ **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 $select\ k\ xs \in set\ xs$
⟨*proof*⟩

lemma *median-in-set* [*intro*, *simp*]:
assumes $xs \neq []$
shows *median xs* $\in$ *set xs*
⟨*proof*⟩

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

lemma *sort-cong*: *mset xs* = *mset ys* $\Longrightarrow$ *sort xs* = *sort ys*
⟨*proof*⟩

lemma *select-cong*:
$k = k' \Longrightarrow$ *mset xs* = *mset xs'* $\Longrightarrow$ *select k xs* = *select k' xs'*
⟨*proof*⟩

lemma *median-cong*: *mset xs* = *mset xs'* $\Longrightarrow$ *median xs* = *median xs'*
⟨*proof*⟩

Selection distributes over appending lists under certain conditions:

lemma *sort-append*:
assumes $\bigwedge x\ y.\ x \in$ *set xs* $\Longrightarrow y \in$ *set ys* $\Longrightarrow x \leq y$
shows *sort* (*xs* @ *ys*) = *sort xs* @ *sort ys*
⟨*proof*⟩

lemma *select-append*:
assumes $\bigwedge y\ z.\ y \in$ *set ys* $\Longrightarrow z \in$ *set zs* $\Longrightarrow y \leq z$
shows $k <$ *length ys* $\Longrightarrow$ *select k* (*ys* @ *zs*) = *select k ys*
$k \in$ {*length ys*..<*length ys* + *length zs*} $\Longrightarrow$
select k (*ys* @ *zs*) = *select* (*k* − *length ys*) *zs*
⟨*proof*⟩

lemma *select-append'*:
assumes $\bigwedge y\ z.\ y \in$ *set ys* $\Longrightarrow z \in$ *set zs* $\Longrightarrow y \leq z$ $k <$ *length ys* + *length zs*
shows *select k* (*ys* @ *zs*) = (*if k* < *length ys then select k ys else select* (*k* − *length ys*) *zs*)
⟨*proof*⟩

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*⟩

lemma *size-greater-than-median*:
size {#*y* ∈# *mset xs*. *y* > *median xs*#} ≤ *length xs div 2*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *list-all2-eq-fun-conv-map*: *list-all2* (λ*x y*. *x* = *f y*) *xs ys* ⟷ *xs* = *map f ys*
⟨*proof*⟩

lemma *transfer-is-median-dual-ord* [*transfer-rule*]:
(*pcr-dual-ord* (=) ===> *list-all2* (*pcr-dual-ord* (=)) ===> (=)) *is-median is-median*
⟨*proof*⟩
end

lemma *is-median-to-dual-ord-iff* [*simp*]:
is-median (*to-dual-ord x*) (*map to-dual-ord xs*) ⟷ *is-median x xs*
⟨*proof*⟩

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* (λ*y*. *y* < *x*) (*mset xs*)) ≤ *length xs div 2* ∧
size (*filter-mset* (λ*y*. *y* > *x*) (*mset xs*)) ≤ *length xs div 2*
⟨*proof*⟩

lemma *is-median-cong*:
assumes *x* = *y mset xs* = *mset ys*
shows *is-median x xs* ⟷ *is-median y ys*
⟨*proof*⟩

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*)
  ⟨*proof*⟩

And, of course, *the* median is a median.

lemma *is-median-median* [*simp*,*intro*]: *is-median* (*median xs*) *xs*
  ⟨*proof*⟩

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-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*⟩

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*: $\bigwedge xs.\ xs \neq [] \Longrightarrow length\ xs \leq Suc\ (2 * d) \Longrightarrow is\text{-}median\ (med\ xs)\ xs$
  assumes *median'*: *is-median x* (*map med* (*chop* (*Suc* (2∗*d*)) *xs*))
  defines $n \equiv length\ xs$
  defines $c \equiv (3 * real\ d + 1) / (2 * (2 * d + 1))$
  shows $size\ \{\#y \in\#\ mset\ xs.\ y < x\#\} \leq nat\ \lceil c * n \rceil + (5 * d)\ div\ 2 + 1$
⟨*proof*⟩

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*: $\bigwedge xs.\ xs \neq [] \Longrightarrow length\ xs \leq Suc\ (2 * d) \Longrightarrow is\text{-}median\ (med\ xs)\ xs$
  assumes *median'*: $is\text{-}median\ x\ (map\ med\ (chop\ (Suc\ (2{*}d))\ xs))$
  defines $n \equiv length\ xs$
  defines $c \equiv (3 * real\ d + 1)\ /\ (2 * (2 * d + 1))$
  shows $size\ \{\#y \in\#\ mset\ xs.\ y > x\#\} \leq nat\ \lceil c * n\rceil + (5 * d)\ div\ 2 + 1$
$\langle proof \rangle$
  include *lifting-syntax*
  $\langle proof \rangle$

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 $length\ (filter\ (\lambda y.\ y > x)\ xs) \leq nat\ \lceil 0.7 * length\ xs\rceil + 6$
  $\langle proof \rangle$

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*)
$\langle proof \rangle$

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 $threeway\text{-}partition :: 'a \Rightarrow 'a :: linorder\ list \Rightarrow 'a\ list \times 'a\ list \times 'a\ list$
where

threeway-partition x xs = (*filter* (λ*y*. *y* < *x*) *xs*, *filter* (λ*y*. *y* = *x*) *xs*, *filter* (λ*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*))
⟨*proof*⟩

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*⟩

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]
  ⟨proof⟩

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]
  ⟨proof⟩

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]
  ⟨proof⟩

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⟩

lemma median-le-5-in-set:
  assumes xs ≠ [] length xs ≤ 5
  shows   median-le-5 xs ∈ set xs
⟨proof⟩
```

lemma *median-le-5*:
  assumes $xs \neq []$ *length xs* ≤ 5
  shows *is-median* (*median-le-5 xs*) *xs*
⟨*proof*⟩

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 $\leq$ *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*
)
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*⟩

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

termination *fast-select* ⟨*proof*⟩

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

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
      )
  ⟨proof⟩

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⟩

end
```

References

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