The Median Method

Emin Karayel

February 6, 2026

Abstract

The median method is an amplification result for randomized approximation algorithms described in [1]. Given an algorithm whose result is in a desired interval with a probability larger than $\frac{1}{2}$, it is possible to improve the success probability, by running the algorithm multiple times independently and using the median. In contrast to using the mean, the amplification of the success probability grows exponentially with the number of independent runs.

This entry contains a formalization of the underlying theorem: Given a sequence of n independent random variables, which are in a desired interval with a probability $\frac{1}{2} + \alpha$. Then their median will be in the desired interval with a probability of $1 - \exp(-2\alpha^2 n)$. In particular, the success probability approaches 1 exponentially with the number of variables.

In addition to that, this entry also contains a proof that order-statistics of Borel-measurable random variables are themselves measurable and that generalized intervals in linearly ordered Borel-spaces are measurable.

Contents

1 Intervals are Borel measurable **1**

2 Order statistics are Borel measurable **4**

3 The Median Method **8**

4 Some additional results about the median **19**

1 Intervals are Borel measurable

theory *Median*
imports
HOL−Probability.Probability
HOL−Library.Multiset
Universal-Hash-Families.Universal-Hash-Families-More-Independent-Families

begin

This section contains a proof that intervals are Borel measurable, where an interval is defined as a convex subset of linearly ordered space, more precisely, a set is an interval, if for each triple of points $x < y < z$: If x and z are in the set so is y. This includes ordinary intervals like $\{a..b\}$, $\{a<..<b\}$ but also for example $\{x::rat.\ x * x < (2::rat)\}$ which cannot be expressed in the standard notation.

In the *HOL−Analysis.Borel-Space* there are proofs for the measurability of each specific type of interval, but those unfortunately do not help if we want to express the result about the median bound for arbitrary types of intervals.

```
definition interval :: ('a :: linorder) set ⇒ bool where
  interval I = (∀ x y z. x ∈ I ⟶ z ∈ I ⟶ x ≤ y ⟶ y ≤ z ⟶ y ∈ I)

definition up-ray :: ('a :: linorder) set ⇒ bool where
  up-ray I = (∀ x y. x ∈ I ⟶ x ≤ y ⟶ y ∈ I)

lemma up-ray-borel:
  assumes up-ray (I :: (('a :: linorder-topology) set))
  shows I ∈ borel
proof (cases closed I)
  case True
  then show ?thesis using borel-closed by blast
next
  case False
  hence b:¬ closed I by blast

  have open I
  proof (rule Topological-Spaces.openI)
    fix x
    assume c:x ∈ I
    show ∃ T. open T ∧ x ∈ T ∧ T ⊆ I
    proof (cases ∃ y. y < x ∧ y ∈ I)
      case True
      then obtain y where a:y < x ∧ y ∈ I by blast
      have open {y<..} by simp
      moreover have x ∈ {y<..} using a by simp
      moreover have {y<..} ⊆ I
        using a assms(1) by (auto simp: up-ray-def)
      ultimately show ?thesis by blast
    next
      case False
      hence I ⊆ {x..} using linorder-not-less by auto
      moreover have {x..} ⊆ I
        using c assms(1) unfolding up-ray-def by blast
      ultimately have I = {x..}
        by (rule order-antisym)
      moreover have closed {x..} by simp
```

```
    ultimately have False using b by auto
    then show ?thesis by simp
   qed
  qed
  then show ?thesis by simp
qed

definition down-ray :: ('a :: linorder) set ⇒ bool where
  down-ray I = (∀ x y. y ∈ I ⟶ x ≤ y ⟶ x ∈ I)

lemma down-ray-borel:
  assumes down-ray (I :: (('a :: linorder-topology) set))
  shows I ∈ borel
proof −
  have up-ray (−I) using assms
    by (simp add: up-ray-def down-ray-def, blast)
  hence (−I) ∈ borel using up-ray-borel by blast
  thus I ∈ borel
    by (metis borel-comp double-complement)
qed
```

Main result of this section:

```
lemma interval-borel:
  assumes interval (I :: (('a :: linorder-topology) set))
  shows I ∈ borel
proof (cases I = {})
  case True
  then show ?thesis by simp
next
  case False
  then obtain x where a:x ∈ I by blast
  have ⋀y z. y ∈ I ∪ {x..} ⟹ y ≤ z ⟹ z ∈ I ∪ {x..}
   by (metis assms a interval-def IntE UnE Un-Int-eq(1) Un-Int-eq(2) atLeast-iff
nle-le order.trans)
  hence up-ray (I ∪ {x..})
    using up-ray-def by blast
  hence b:I ∪ {x..} ∈ borel
    using up-ray-borel by blast

  have ⋀y z. y ∈ I ∪ {..x} ⟹ z ≤ y ⟹ z ∈ I ∪ {..x}
     by (metis assms a interval-def UnE UnI1 UnI2 atMost-iff dual-order.trans
linorder-le-cases)
  hence down-ray (I ∪ {..x})
    using down-ray-def by blast
  hence c:I ∪ {..x} ∈ borel
    using down-ray-borel by blast

  have I = (I ∪ {x..}) ∩ (I ∪ {..x})
    using a by fastforce
```

then show *?thesis* **using** *b c*
by (*metis sets.Int*)
qed

2 Order statistics are Borel measurable

This section contains a proof that order statistics of Borel measurable random variables are themselves Borel measurable.

The proof relies on the existence of branch-free comparison-sort algorithms. Given a sequence length these algorithms perform compare-swap operations on predefined pairs of positions. In particular the result of a comparison does not affect future operations. An example for a branch-free comparison sort algorithm is shell-sort and also bubble-sort without the early exit.

The advantage of using such a comparison-sort algorithm is that it can be lifted to work on random variables, where the result of a comparison-swap operation on two random variables X and Y can be represented as the expressions $\lambda\omega.\ min\ (X\ \omega)\ (Y\ \omega)$ and $\lambda\omega.\ max\ (X\ \omega)\ (Y\ \omega)$.

Because taking the point-wise minimum (resp. maximum) of two random variables is still Borel measurable, and because the entire sorting operation can be represented using such compare-swap operations, we can show that all order statistics are Borel measuable.

fun *sort-primitive* **where**
sort-primitive i j f k = (*if k* = *i then min* (*f i*) (*f j*) *else* (*if k* = *j then max* (*f i*) (*f j*) *else f k*))

fun *sort-map* **where**
sort-map f n = *fold id* [*sort-primitive j i. i* <− [*0*..<*n*], *j* <− [*0*..<*i*]] *f*

lemma *sort-map-ind*:
sort-map f (*Suc n*) = *fold id* [*sort-primitive j n. j* <− [*0*..<*n*]] (*sort-map f n*)
by *simp*

lemma *sort-map-strict-mono*:
fixes *f* :: *nat* ⇒ *'b* :: *linorder*
shows *j* < *n* ⟹ *i* < *j* ⟹ *sort-map f n i* ≤ *sort-map f n j*
proof (*induction n arbitrary*: *i j*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
define *g* **where** *g* = (*λk. fold id* [*sort-primitive j n. j* <− [*0*..<*k*]] (*sort-map f n*))
define *k* **where** *k* = *n*
have *a*:(∀ *i j. j* < *n* ⟶ *i* < *j* ⟶ *g k i* ≤ *g k j*) ∧ (∀ *l. l* < *k* ⟶ *g k l* ≤ *g k n*)
proof (*induction k*)

```
    case 0
    then show ?case using Suc by (simp add:g-def del:sort-map.simps)
  next
    case (Suc k)
    have g (Suc k) = sort-primitive k n (g k)
      by (simp add:g-def)
    then show ?case using Suc
      apply (cases g k k ≤ g k n)
       apply (simp add:min-def max-def)
       using less-antisym apply blast
      apply (cases g k n ≤ g k k)
       apply (simp add:min-def max-def)
       apply (metis less-antisym max.coboundedI2 max.orderE)
      by simp
  qed

  hence ⋀i j. j < Suc n ⟹ i < j ⟹ g n i ≤ g n j
    apply (simp add:k-def) using less-antisym by blast
  moreover have sort-map f (Suc n) = g n
    by (simp add:sort-map-ind g-def del:sort-map.simps)
  ultimately show ?case
    using Suc by (simp del:sort-map.simps)
qed

lemma sort-map-mono:
  fixes f :: nat ⇒ 'b :: linorder
  shows j < n ⟹ i ≤ j ⟹ sort-map f n i ≤ sort-map f n j
  by (metis sort-map-strict-mono eq-iff le-imp-less-or-eq)

lemma sort-map-perm:
  fixes f :: nat ⇒ 'b :: linorder
  shows image-mset (sort-map f n) (mset [0..<n]) = image-mset f (mset [0..<n])
proof −
  define is-swap where is-swap = (λ(ts :: ((nat ⇒ 'b) ⇒ nat ⇒ 'b)). ∃ i < n. ∃ j
< n. ts = sort-primitive i j)
  define t :: ((nat ⇒ 'b) ⇒ nat ⇒ 'b) list
    where t = [sort-primitive j i. i <− [0..<n], j <− [0..<i]]

  have a: ⋀x f. is-swap x ⟹ image-mset (x f) (mset-set {0..<n}) = image-mset
f (mset-set {0..<n})
  proof −
    fix x
    fix f :: nat ⇒ 'b :: linorder
    assume is-swap x
    then obtain i j where x-def: x = sort-primitive i j and i-bound: i < n and
j-bound:j < n
      using is-swap-def by blast
    define inv where inv = mset-set {k. k < n ∧ k ≠ i ∧ k ≠ j}
    have b:{0..<n} = {k. k < n ∧ k ≠ i ∧ k ≠ j} ∪ {i,j}
```

```
    apply (rule order-antisym, rule subsetI, simp, blast, rule subsetI, simp)
    using i-bound j-bound by meson
  have c:⋀k. k ∈# inv ⟹ (x f) k = f k
    by (simp add:x-def inv-def)
  have image-mset (x f) inv = image-mset f inv
    apply (rule multiset-eqI)
    using c multiset.map-cong0 by force
  moreover have image-mset (x f) (mset-set {i,j}) = image-mset f (mset-set
{i,j})
    apply (cases i = j)
    by (simp add:x-def max-def min-def)+
  moreover have mset-set {0..<n} = inv + mset-set {i,j}
    by (simp only:inv-def b, rule mset-set-Union, simp, simp, simp)
 ultimately show image-mset (x f) (mset-set {0..<n}) = image-mset f (mset-set
{0..<n})
    by simp
 qed

  have (∀ x ∈ set t. is-swap x) ⟹ image-mset (fold id t f) (mset [0..<n]) =
image-mset f (mset [0..<n])
  by (induction t arbitrary:f, simp, simp add:a)
 moreover have ⋀x. x ∈ set t ⟹ is-swap x
  apply (simp add:t-def is-swap-def)
  by (meson atLeastLessThan-iff imageE less-imp-le less-le-trans)
 ultimately have image-mset (fold id t f) (mset [0..<n]) = image-mset f (mset
[0..<n]) by blast
 then show ?thesis by (simp add:t-def)
qed

lemma list-eq-iff:
 assumes mset xs = mset ys
 assumes sorted xs
 assumes sorted ys
 shows xs = ys
 using assms properties-for-sort by blast

lemma sort-map-eq-sort:
 fixes f :: nat ⇒ ('b :: linorder)
 shows map (sort-map f n) [0..<n] = sort (map f [0..<n]) (is ?A = ?B)
proof −
 have mset ?A = mset ?B
  using sort-map-perm[where f=f and n=n]
  by (simp del:sort-map.simps)
 moreover have sorted ?B
  by simp
 moreover have sorted ?A
  apply (subst sorted-wrt-iff-nth-less)
  apply (simp del:sort-map.simps)
  by (metis sort-map-mono nat-less-le)
```

```
  ultimately show ?A = ?B
    using list-eq-iff by blast
qed

lemma order-statistics-measurable-aux:
  fixes X :: nat ⇒ 'a ⇒ ('b :: {linorder-topology, second-countable-topology})
  assumes n ≥ 1
  assumes j < n
  assumes ⋀i. i < n ⟹ X i ∈ measurable M borel
  shows (λx. (sort-map (λi. X i x) n) j) ∈ measurable M borel
proof −
  have n-ge-0:n > 0 using assms by simp
  define is-swap where is-swap = (λ(ts :: ((nat ⇒ 'b) ⇒ nat ⇒ 'b)). ∃ i < n. ∃ j
< n. ts = sort-primitive i j)
  define t :: ((nat ⇒ 'b) ⇒ nat ⇒ 'b) list
    where t = [sort-primitive j i. i <− [0..<n], j <− [0..<i]]

  define meas-ptw :: (nat ⇒ 'a ⇒ 'b) ⇒ bool
    where meas-ptw = (λf. (∀ k. k < n ⟶ f k ∈ borel-measurable M))

  have ind-step:
    ⋀x (g :: nat ⇒ 'a ⇒ 'b). meas-ptw g ⟹ is-swap x ⟹ meas-ptw (λk ω. x (λi.
g i ω) k)
  proof −
    fix x g
    assume meas-ptw g
    hence a:⋀k. k < n ⟹ g k ∈ borel-measurable M by (simp add:meas-ptw-def)
    assume is-swap x
    then obtain i j where x-def:x=sort-primitive i j and i-le:i < n and j-le:j <
n
      by (simp add:is-swap-def, blast)
    have ⋀k. k < n ⟹ (λω. x (λi. g i ω) k) ∈ borel-measurable M
    proof −
      fix k
      assume k < n
      thus  (λω. x (λi. g i ω) k) ∈ borel-measurable M
        apply (simp add:x-def)
        apply (cases k = i, simp)
        using a i-le j-le borel-measurable-min apply blast
        apply (cases k = j, simp)
        using a i-le j-le borel-measurable-max apply blast
        using a by simp
    qed
    thus meas-ptw (λk ω. x (λi. g i ω) k)
      by (simp add:meas-ptw-def)
  qed

  have (∀ x ∈ set t. is-swap x) ⟹ meas-ptw (λ k ω. (fold id t (λk. X k ω)) k)
  proof (induction t rule:rev-induct)
```

```
    case Nil
    then show ?case using assms by (simp add:meas-ptw-def)
  next
    case (snoc x xs)
    have a:meas-ptw (λk ω. fold (λa. a) xs (λk. X k ω) k) using snoc by simp
    have b:is-swap x using snoc by simp
    show ?case using ind-step[OF a b] by simp
  qed
  moreover have ⋀x. x ∈ set t ⟹ is-swap x
    apply (simp add:t-def is-swap-def)
    by (meson atLeastLessThan-iff imageE less-imp-le less-le-trans)
  ultimately show ?thesis using assms
    by (simp add:t-def[symmetric] meas-ptw-def)
qed
```

Main results of this section:

```
lemma order-statistics-measurable:
  fixes X :: nat ⇒ 'a ⇒ ('b :: {linorder-topology, second-countable-topology})
  assumes n ≥ 1
  assumes j < n
  assumes ⋀i. i < n ⟹ X i ∈ measurable M borel
  shows (λx. (sort (map (λi. X i x) [0..<n])) ! j) ∈ measurable M borel
  apply (subst sort-map-eq-sort[symmetric])
  using assms by (simp add:order-statistics-measurable-aux del:sort-map.simps)

definition median :: nat ⇒ (nat ⇒ ('a :: linorder)) ⇒ 'a where
  median n f = sort (map f [0..<n]) ! (n div 2)

lemma median-measurable:
  fixes X :: nat ⇒ 'a ⇒ ('b :: {linorder-topology, second-countable-topology})
  assumes n ≥ 1
  assumes ⋀i. i < n ⟹ X i ∈ measurable M borel
  shows (λx. median n (λi. X i x)) ∈ measurable M borel
  apply (simp add:median-def)
  apply (rule order-statistics-measurable[OF assms(1) - assms(2)])
  using assms(1) by force+
```

3 The Median Method

This section contains the proof for the probability that the median of independent random variables will be in an interval with high probability if the individual variables are in the same interval with probability larger than $\frac{1}{2}$. The proof starts with the elementary observation that the median of a seqeuence with n elements is in an interval I if at least half of them are in I. This works because after sorting the sequence the elements that will be in the interval must necessarily form a consecutive subsequence, if its length is larger than $\frac{n}{2}$ the median must be in it.

The remainder follows the proof in [1, §2.1] using the Hoeffding inequality to estimate the probability that at least half of the sequence elements will be in the interval I.

```
lemma interval-rule:
  assumes interval I
  assumes a ≤ x x ≤ b
  assumes a ∈ I
  assumes b ∈ I
  shows x ∈ I
  using assms(1) apply (simp add:interval-def)
  using assms by blast

lemma sorted-int:
  assumes interval I
  assumes sorted xs
  assumes k < length xs i ≤ j j ≤ k
  assumes xs ! i ∈ I xs ! k ∈ I
  shows xs ! j ∈ I
  apply (rule interval-rule[where a=xs ! i and b=xs ! k])
  using assms by (simp add: sorted-nth-mono)+

lemma mid-in-interval:
  assumes 2*length (filter (λx. x ∈ I) xs) > length xs
  assumes interval I
  assumes sorted xs
  shows xs ! (length xs div 2) ∈ I
proof −
  have length (filter (λx. x ∈ I) xs) > 0  using assms(1) by linarith
  then obtain v where v-1: v < length xs and v-2: xs ! v ∈ I
    by (metis filter-False in-set-conv-nth length-greater-0-conv)

  define J where J = {k. k < length xs ∧ xs ! k ∈ I}

  have card-J-min: 2*card J > length xs
    using assms(1) by (simp add:J-def length-filter-conv-card)

  consider
    (a) xs ! (length xs div 2) ∈ I |
    (b) xs ! (length xs div 2) ∉ I ∧ v > (length xs div 2) |
    (c) xs ! (length xs div 2) ∉ I ∧ v < (length xs div 2)
    by (metis linorder-cases v-2)
  thus ?thesis
  proof (cases)
    case a
    then show ?thesis by simp
  next
    case b
    have p:⋀k. k ≤ length xs div 2 ⟹ xs ! k ∉ I
      using b v-2 sorted-int[OF assms(2) assms(3) v-1, where j=length xs div 2]
```

apply *simp* **by** *blast*
hence *card J* ≤ *card* {*Suc* (*length xs div 2*)..<*length xs*}
unfolding *J-def* **using** *not-less-eq-eq*[*symmetric*] **by** (*intro card-mono subsetI*) *auto*
hence *card J* ≤ *length xs* − (*Suc* (*length xs div 2*))
using *card-atLeastLessThan* **by** *metis*
hence *length xs* ≤ *2*∗(*length xs* − (*Suc* (*length xs div 2*)))
using *card-J-min* **by** *linarith*
hence *False* **using** *b v-1* **by** *auto*
then show *?thesis* **by** *simp*
next
case *c*
have ⋀*k*. *k* ≥ *length xs div 2* ⟹ *k* < *length xs* ⟹ *xs* ! *k* ∉ *I*
using *c v-1 v-2 sorted-int*[*OF assms*(*2*,*3*), **where** *i* =*v* **and** *j*=*length xs div 2*]
by *simp blast*
hence *card J* ≤ *card* {*0*..<(*length xs div 2*)}
unfolding *J-def* **using** *linorder-le-less-linear* **by** (*intro card-mono subsetI*) *auto*
hence *card J* ≤ (*length xs div 2*)
using *card-atLeastLessThan* **by** *simp*
then show *?thesis* **using** *card-J-min* **by** *linarith*
qed
qed

lemma *median-est*:
assumes *interval I*
assumes *2*∗*card* {*k*. *k* < *n* ∧ *f k* ∈ *I*} > *n*
shows *median n f* ∈ *I*
proof −
have {*k*. *k* < *n* ∧ *f k* ∈ *I*} = {*i*. *i* < *n* ∧ *map f* [*0*..<*n*] ! *i* ∈ *I*} **by** *auto*
thus *?thesis* **using** *assms* **unfolding** *median-def*
by (*intro mid-in-interval*[*OF - assms*(*1*),**where** *xs*=*sort* (*map f* [*0*..<*n*]), *simplified*])
(*simp-all add*:*filter-sort comp-def length-filter-conv-card*)
qed

lemma *median-est-rev*:
assumes *interval I*
assumes *median n f* ∉ *I*
shows *2*∗*card* {*k*. *k* < *n* ∧ *f k* ∉ *I*} ≥ *n*
proof (*rule ccontr*)
assume *a*: ¬(*2*∗*card* {*k*. *k* < *n* ∧ *f k* ∉ *I*} ≥ *n*)

have *2* ∗ *n* = *2* ∗ *card* {*k*. *k* < *n*} **by** *simp*
also have ... = *2* ∗ *card* ({*k*. *k* < *n* ∧ *f k* ∈ *I*} ∪ {*k*. *k* < *n* ∧ *f k* ∉ *I*})
by (*intro arg-cong2*[**where** *f*=(∗)] *refl arg-cong*[**where** *f*=*card*]) *auto*
also have ... = *2* ∗ *card* {*k*. *k* < *n* ∧ *f k* ∈ *I*} + *2* ∗ *card* {*k*. *k* < *n* ∧ *f k* ∉ *I*}
by (*subst card-Un-disjoint*) *auto*

```
  also have ... ≤ n + 2 * card {k. k < n ∧ f k ∉ I}
    using median-est[OF assms(1)] assms(2) not-less by (intro add-mono) auto
  also have ... < n + n
    using a by (intro add-strict-left-mono) auto
  finally show False by auto
qed

lemma prod-pmf-bernoulli-mono:
  assumes finite I
  assumes ⋀i. i ∈ I ⟹ 0 ≤ f i ∧ f i ≤ g i ∧ g i ≤ 1
  assumes ⋀x y. x ∈ A ⟹ (∀ i ∈ I. x i ≤ y i) ⟹ y ∈ A
   shows measure (Pi-pmf I d (bernoulli-pmf ∘ f)) A ≤ measure (Pi-pmf I d
(bernoulli-pmf ∘ g)) A
    (is ?L ≤ ?R)
proof −
  define q where q i = pmf-of-list [(0::nat, f i), (1, g i − f i), (2, 1 − g i)] for i

  have wf:pmf-of-list-wf [(0::nat, f i), (1, g i − f i), (2, 1 − g i)] if i ∈ I for i
    using assms(2)[OF that] by (intro pmf-of-list-wfI) auto

  have 0: bernoulli-pmf (f i) = map-pmf (λx. x = 0) (q i) (is ?L1 = ?R1)
    if i ∈ I for i
  proof −
    have 0 ≤ f i f i ≤ 1 using assms(2)[OF that] by auto
    hence pmf ?L1 x = pmf ?R1 x for x
      unfolding q-def pmf-map measure-pmf-of-list[OF wf[OF that]]
      by (cases x;simp-all add:vimage-def)
    thus ?thesis
      by (intro pmf-eqI) auto
  qed

  have 1: bernoulli-pmf (g i) = map-pmf (λx. x ∈ {0,1}) (q i) (is ?L1 = ?R1)
    if i ∈ I for i
  proof −
    have 0 ≤ g i g i ≤ 1 using assms(2)[OF that] by auto
    hence pmf ?L1 x = pmf ?R1 x for x
      unfolding q-def pmf-map measure-pmf-of-list[OF wf[OF that]]
      by (cases x;simp-all add:vimage-def)
    thus ?thesis
      by (intro pmf-eqI) auto
  qed

  have 2: (λk. x k = 0) ∈ A ⟹ (λk. x k = 0 ∨ x k = Suc 0) ∈ A for x
    by (erule assms(3)) auto

  have ?L = measure (Pi-pmf I d (λi. map-pmf (λx. x = 0) (q i))) A
    unfolding comp-def by (simp add:0 cong: Pi-pmf-cong)
  also have ... = measure (map-pmf ((∘) (λx. x = 0)) (Pi-pmf I (if d then 0 else
2) q)) A
```

by (*intro arg-cong2*[**where** *f*=*measure-pmf.prob*] *Pi-pmf-map*[*OF assms*(*1*)])
auto
also have ... = *measure* (*Pi-pmf I* (*if d then 0 else 2*) *q*) {*x*. (λ*k*. *x k* = *0*) ∈ *A*}
by (*simp add*:*comp-def vimage-def*)
also have ... ≤ *measure* (*Pi-pmf I* (*if d then 0 else 2*) *q*) {*x*. (λ*k*. *x k* ∈ {*0*,*1*})
∈ *A*}
using *2* **by** (*intro measure-pmf.finite-measure-mono subsetI*) *auto*
also have ... = *measure* (*map-pmf* ((∘) (λ*x*. *x* ∈ {*0*,*1*})) (*Pi-pmf I* (*if d then 0
else 2*) *q*)) *A*
by (*simp add*:*vimage-def comp-def*)
also have ... = *measure* (*Pi-pmf I d* (λ*i*. *map-pmf* (λ*x*. *x* ∈ {*0*,*1*}) (*q i*))) *A*
by (*intro arg-cong2*[**where** *f*=*measure-pmf.prob*] *Pi-pmf-map*[*OF assms*(*1*),
symmetric]) *auto*
also have ... = *?R*
unfolding *comp-def* **by** (*simp add*:*1 cong*: *Pi-pmf-cong*)
finally show *?thesis* **by** *simp*
qed

lemma *discrete-measure-eqI*:
assumes *sets M* = *count-space UNIV*
assumes *sets N* = *count-space UNIV*
assumes *countable* Ω
assumes ⋀*x*. *x* ∈ Ω ⟹ *emeasure M* {*x*} = *emeasure N* {*x*} ∧ *emeasure M* {*x*}
≠ ∞
assumes *AE x in M*. *x* ∈ Ω
assumes *AE x in N*. *x* ∈ Ω
shows *M* = *N*
proof −
define *E* **where** *E* = *insert* {} ((λ*x*. {*x*}) ‘ Ω)

have *0*: *Int-stable E* **unfolding** *E-def* **by** (*intro Int-stableI*) *auto*
have *1*: *countable E* **using** *assms*(*3*) **unfolding** *E-def* **by** *simp*

have *E* ⊆ *Pow* Ω **unfolding** *E-def* **by** *auto*
have *emeasure M A* = *emeasure N A* **if** *A-range*: *A* ∈ *E* **for** *A*
using *that assms*(*4*) **unfolding** *E-def* **by** *auto*
moreover have *sets M* = *sets N* **using** *assms*(*1*,*2*) **by** *simp*
moreover have Ω ∈ *sets M* **using** *assms*(*1*) **by** *simp*
moreover have *E* ≠ {} **unfolding** *E-def* **by** *simp*
moreover have ⋃ *E* = Ω **unfolding** *E-def* **by** *simp*
moreover have *emeasure M a* ≠ ∞ **if** *a* ∈ *E* **for** *a*
using *that assms*(*4*) **unfolding** *E-def* **by** *auto*
moreover have *sets* (*restrict-space M* Ω) = *Pow* Ω
using *assms*(*1*) **by** (*simp add*:*sets-restrict-space range-inter*)
moreover have *sets* (*restrict-space N* Ω) = *Pow* Ω
using *assms*(*2*) **by** (*simp add*:*sets-restrict-space range-inter*)
moreover have *sigma-sets* Ω *E* = *Pow* Ω
unfolding *E-def* **by** (*intro sigma-sets-singletons-and-empty assms*(*3*))
ultimately show *?thesis*

by (*intro measure-eqI-restrict-generator*[*OF 0 - - - - - - assms*(5,6) *1*]) *auto*
qed

Main results of this section:

The next theorem establishes a bound for the probability of the median of independent random variables using the binomial distribution. In a follow-up step, we will establish tail bounds for the binomial distribution and corresponding median bounds.

This two-step strategy was suggested by Yong Kiam Tan. In a previous version, I only had verified the exponential tail bound (see theorem `median_bound` below).

theorem (**in** *prob-space*) *median-bound-raw*:
fixes $I :: ('b :: \{linorder\text{-}topology, second\text{-}countable\text{-}topology\})\ set$
assumes $n > 0\ p \geq 0$
assumes *interval I*
assumes *indep-vars* (λ-. *borel*) $X\ \{0..<n\}$
assumes $\bigwedge i.\ i < n \Longrightarrow \mathcal{P}(\omega\ in\ M.\ X\ i\ \omega \in I) \geq p$
shows $\mathcal{P}(\omega\ in\ M.\ median\ n\ (\lambda i.\ X\ i\ \omega) \in I) \geq 1 - measure\ (binomial\text{-}pmf\ n\ p)\ \{..n\ div\ 2\}$
(**is** *?L* $\geq$ *?R*)
proof −
let *?pi* = *Pi-pmf* $\{..<n\}$ *undefined*
define q **where** $q\ i = \mathcal{P}(\omega\ in\ M.\ X\ i\ \omega \in I)$ **for** i

have *n-ge-1*: $n \geq 1$ **using** *assms*(*1*) **by** *simp*

have *0*: $\{k.\ k < n \wedge (k < n \longrightarrow X\ k\ \omega \in I)\} = \{k.\ k < n \wedge X\ k\ \omega \in I\}$ **for** ω
by *auto*

have *countable* ($\{..<n\} \rightarrow_E$ (*UNIV* :: *bool set*))
by (*intro countable-PiE*) *auto*
hence *countable-ext*: *countable* (*extensional* $\{..<n\}$ :: (*nat* $\Rightarrow$ *bool*) *set*)
unfolding *PiE-def* **by** *auto*

have *m0*: $I \in sets\ borel$
using *interval-borel*[*OF assms*(*3*)] **by** *simp*

have *m1*: *random-variable borel* ($\lambda x.\ X\ k\ x$) **if** $k \in \{..<n\}$ **for** k
using *assms*(*4*) *that* **unfolding** *indep-vars-def* **by** *auto*

have *m2*: $(\lambda x.\ x \in I) \in borel \rightarrow_M (measure\text{-}pmf\ ((bernoulli\text{-}pmf \circ q)\ k))$
for k **using** *m0* **by** *measurable*
hence *m3*: *random-variable* (*measure-pmf* ((*bernoulli-pmf* $\circ$ q) k)) ($\lambda x.\ X\ k\ x \in I$)
if $k \in \{..<n\}$ **for** k
by (*intro measurable-compose*[*OF m1*] *that*)

hence *m4*: *random-variable* (*PiM* {..<*n*} (*bernoulli-pmf* ∘ *q*)) (λω. λ*k*∈{..<*n*}. *X k* ω ∈ *I*)
by (*intro measurable-restrict*) *auto*
moreover have *A* ∈ *sets* (*Pi*$_M$ {..<*n*} (λ*x*. *measure-pmf* (*bernoulli-pmf* (*q x*))))
if *A* ⊆ *extensional* {..<*n*} **for** *A*
proof −
have *A* = (⋃ *a* ∈ *A*. {*a*}) **by** *auto*
also have ... = (⋃ *a* ∈ *A*. *PiE* {..<*n*} (λ*k*. {*a k*}))
using *that* **by** (*intro arg-cong*[**where** *f*=*Union*] *image-cong refl PiE-singleton*[*symmetric*]) *auto*
also have ... ∈ *sets* (*Pi*$_M$ {..<*n*} (λ*x*. *measure-pmf* (*bernoulli-pmf* (*q x*))))
using *that countable-ext countable-subset*
by (*intro sets.countable-Union countable-image image-subsetI sets-PiM-I-finite*) *auto*
finally show *?thesis* **by** *simp*
qed
hence *m5*: *id* ∈ (*PiM* {..<*n*} (*bernoulli-pmf* ∘ *q*)) →$_M$ (*count-space UNIV*)
by (*intro measurableI*) (*simp-all add*:*vimage-def space-PiM PiE-def*)
ultimately have *random-variable* (*count-space UNIV*) (*id* ∘ (λω. λ*k*∈{..<*n*}. *X k* ω ∈ *I*))
by (*rule measurable-comp*)
hence *m6*: *random-variable* (*count-space UNIV*) (λω. λ*k*∈{..<*n*}. *X k* ω ∈ *I*) **by** *simp*

have *indep*: *indep-vars* (*bernoulli-pmf* ∘ *q*) (λ*i x*. *X i x* ∈ *I*) {*0*..<*n*}
by (*intro indep-vars-compose2*[*OF assms*(*4*)] *m2*)

have *measure M* {*x* ∈ *space M*. (*X k x* ∈ *I*) = ω} = *measure* (*bernoulli-pmf* (*q k*)) {ω}
if *k* < *n* **for** ω *k*
proof (*cases* ω)
case *True*
then show *?thesis* **unfolding** *q-def* **by** (*simp add*:*measure-pmf-single*)
next
case *False*
have {*x* ∈ *space M*. *X k x* ∈ *I*} ∈ *events*
using *that m0* **by** (*intro measurable-sets-Collect*[*OF m1*]) *auto*
hence *prob* {*x* ∈ *space M*. *X k x* ∉ *I*} = *1* − *prob* {ω ∈ *space M*. *X k* ω ∈ *I*}
by (*subst prob-neg*) *auto*
thus *?thesis* **using** *False* **unfolding** *q-def* **by** (*simp add*:*measure-pmf-single*)
qed

hence *1*: *emeasure M* {*x* ∈ *space M*. (*X k x* ∈ *I*) = ω} = *emeasure* (*bernoulli-pmf* (*q k*)) {ω}
if *k* < *n* **for** ω *k*
using *that* **unfolding** *emeasure-eq-measure measure-pmf*.*emeasure-eq-measure* **by** *simp*

interpret *product-sigma-finite* (*bernoulli-pmf* ∘ *q*)

by *standard*

have *distr M* (*count-space UNIV*) ($\lambda\omega$. ($\lambda k\in\{..<n\}$. *X k* $\omega \in I$)) = *distr*
(*distr M* (*PiM* $\{..<n\}$ (*bernoulli-pmf* $\circ$ *q*)) ($\lambda\omega$. $\lambda k\in\{..<n\}$. *X k* $\omega \in I$))
(*count-space UNIV*) *id*
by (*subst distr-distr*[*OF m5 m4*]) (*simp add*:*comp-def*)
also have ... = *distr* (*PiM* $\{..<n\}$ (λi. (*distr M* ((*bernoulli-pmf* $\circ$ *q*) *i*) ($\lambda\omega$. *X*
i $\omega \in I$))))
(*count-space UNIV*) *id*
using *assms*(*1*) *indep atLeast0LessThan* **by** (*intro arg-cong2*[**where** *f*=λx *y*.
distr x y id]
iffD1[*OF indep-vars-iff-distr-eq-PiM'*] *m3*) *auto*
also have ... = *distr* (*PiM* $\{..<n\}$ (*bernoulli-pmf* $\circ$ *q*)) (*count-space UNIV*) *id*
using *m3 1* **by** (*intro distr-cong PiM-cong refl discrete-measure-eqI*[**where**
Ω=*UNIV*])
(*simp-all add*:*emeasure-distr vimage-def Int-def conj-commute*)
also have ... = *?pi* (*bernoulli-pmf* $\circ$ *q*)
proof (*rule discrete-measure-eqI*[**where** Ω=*extensional* $\{..<n\}$], *goal-cases*)
case *1* **show** *?case* **by** *simp*
next
case *2* **show** *?case* **by** *simp*
next
case *3* **show** *?case* **using** *countable-ext* **by** *simp*
next
case (*4 x*)
have *emeasure* (Pi_M $\{..<n\}$ (*bernoulli-pmf* $\circ$ *q*)) $\{x\}$ =
emeasure (Pi_M $\{..<n\}$ (*bernoulli-pmf* $\circ$ *q*)) (*PiE* $\{..<n\}$ (λk. $\{x\ k\}$))
using *PiE-singleton*[*OF 4*] **by** *simp*
also have ... = ($\prod i<n$. *emeasure* (*measure-pmf* (*bernoulli-pmf* (*q i*))) $\{x\ i\}$)
by (*subst emeasure-PiM*) *auto*
also have ... = *emeasure* (*Pi-pmf* $\{..<n\}$ *undefined* (*bernoulli-pmf*$\circ$*q*))
(*PiE-dflt* $\{..<n\}$ *undefined* (λk. $\{x\ k\}$))
unfolding *measure-pmf*.*emeasure-eq-measure*
by (*subst measure-Pi-pmf-PiE-dflt*) (*simp-all add*:*prod-ennreal*)
also have ... = *emeasure* (*Pi-pmf* $\{..<n\}$ *undefined* (*bernoulli-pmf*$\circ$*q*)) $\{x\}$
using *4* **by** (*intro arg-cong2*[**where** *f*=*emeasure*]) (*auto simp add*:*PiE-dflt-def*
extensional-def)
finally have *emeasure* (Pi_M $\{..<n\}$ (*bernoulli-pmf* $\circ$ *q*)) $\{x\}$ =
emeasure (*Pi-pmf* $\{..<n\}$ *undefined* (*bernoulli-pmf* $\circ$ *q*)) $\{x\}$
by *simp*
thus *?case* **using** *4*
by (*subst* (*1 2*) *emeasure-distr*[*OF m5*]) (*simp-all add*:*vimage-def space-PiM*
PiE-def)
next
case *5*
have *AE x in* Pi_M $\{..<n\}$ (*bernoulli-pmf* $\circ$ *q*). $x \in$ *extensional* $\{..<n\}$
by (*intro AE-I2*) (*simp add*:*space-PiM PiE-def*)
then show *?case* **by** (*subst AE-distr-iff*[*OF m5*]) *simp-all*
next

```
    case 6
    then show ?case by (intro AE-pmfI) (simp add: set-Pi-pmf PiE-dflt-def ex-
tensional-def)
  qed
  finally have 2: distr M (count-space UNIV) (λω. (λk∈{..<n}. X k ω ∈ I)) =
?pi (bernoulli-pmf∘q)
    by simp

  have 3: n < 2 * card {k. k < n ∧ y k} if
    n < 2 * card {k. k < n ∧ x k} ⋀i. i < n ⟹ x i ⟹ y i for x y
  proof −
    have 2 * card {k. k < n ∧ x k} ≤ 2 * card {k. k < n ∧ y k}
      using that(2) by (intro mult-left-mono card-mono) auto
    thus ?thesis using that(1) by simp
  qed

  have 4: 0 ≤ p ∧ p ≤ q i ∧ q i ≤ 1 if i < n for i
    unfolding q-def using assms(2,5) that by auto

  have p-range: p ∈ {0..1}
    using 4[OF assms(1)] by auto

  have ?R = 1 − measure-pmf.prob (binomial-pmf n p) {k. 2 * k ≤ n}
     by (intro arg-cong2[where f=(−)] arg-cong2[where f=measure-pmf.prob])
auto
  also have ... = measure (binomial-pmf n p) {k. n < 2 * k}
    by (subst measure-pmf.prob-compl[symmetric]) (simp-all add:set-diff-eq not-le)
  also have ... = measure (?pi (bernoulli-pmf ∘ (λ-. p))) {ω. n < 2 * card {k. k
< n ∧ ω k}}
    using p-range by (subst binomial-pmf-altdef'[where A={..<n} and dflt=undefined])
auto
  also have ... ≤ measure (?pi (bernoulli-pmf ∘ q)) {ω. n < 2 * card {k. k < n
∧ ω k}}
    using 3 4 by (intro prod-pmf-bernoulli-mono) auto
  also have ... =
    P(ω in distr M (count-space UNIV) (λω. λk∈{..<n}. X k ω ∈ I). n<2*card
{k. k < n ∧ ω k})
    unfolding 2 by simp
  also have ... = P(ω in M. n < 2*card {k. k < n ∧ X k ω ∈ I})
   by (subst measure-distr[OF m6]) (simp-all add:vimage-def Int-def conj-commute
0)
  also have ... ≤ ?L
    using median-est[OF assms(3)] m0 m1
   by (intro finite-measure-mono measurable-sets-Collect[OF median-measurable[OF
n-ge-1]]) auto
  finally show ?R ≤ ?L by simp
qed
```

Cumulative distribution of the binomial distribution (contributed by Yong

Kiam Tan):

```
lemma prob-binomial-pmf-upto:
  assumes 0 ≤ p p ≤ 1
  shows measure-pmf.prob (binomial-pmf n p) {..m} =
    sum (λi. real (n choose i) * p^i * (1 - p) ^(n-i)) {0..m}
 by (auto simp: pmf-binomial[OF assms] measure-measure-pmf-finite intro!: sum.cong)
```

A tail bound for the binomial distribution using Hoeffding's inequality:

```
lemma binomial-pmf-tail:
  assumes p ∈ {0..1} real k ≤ real n * p
  shows measure (binomial-pmf n p) {..k} ≤ exp (- 2 * real n * (p - real k /
n) ^2)
    (is ?L ≤ ?R)
proof (cases n = 0)
  case True then show ?thesis by simp
next
  case False
  let ?A = {..<n}
  let ?pi = Pi-pmf ?A undefined (λ-. bernoulli-pmf p)

  define μ where μ = (∑ i<n. (∫ x. (of-bool (x i) :: real) ∂ ?pi))
  define ε :: real where ε = μ - k

  have μ = (∑ i<n. (∫ x. (of-bool x :: real) ∂ (map-pmf (λω. ω i) ?pi)))
    unfolding μ-def by simp
  also have ... = (∑ i<n. (∫ x. (of-bool x :: real) ∂ (bernoulli-pmf p)))
    by (simp add: Pi-pmf-component)
  also have ... = real n * p using assms(1) by simp
  finally have μ-alt: μ = real n * p
    by simp

  have ε-ge-0: ε ≥ 0
    using assms(2) unfolding ε-def μ-alt by auto

  have indep: prob-space.indep-vars ?pi (λ-. borel) (λk ω. of-bool (ω k)) {..<n}
   by (intro prob-space.indep-vars-compose2[OF prob-space-measure-pmf indep-vars-Pi-pmf])
auto
  interpret Hoeffding-ineq ?pi {..<n} λk ω. of-bool (ω k) λ-.0 λ-.1 μ
    using indep unfolding μ-def by (unfold-locales) simp-all

  have ?L = measure (map-pmf (λf. card {x ∈ ?A. f x}) ?pi) {..k}
   by (intro arg-cong2[where f=measure-pmf.prob] binomial-pmf-altdef' assms(1))
auto
  also have ... = P(ω in ?pi. (∑ i<n. of-bool (ω i)) ≤ μ - ε)
    unfolding ε-def by (simp add:vimage-def Int-def)
  also have ... ≤ exp (- 2 * ε^2 / (∑ i<n. (1 - 0)^2))
    using False by (intro Hoeffding-ineq-le ε-ge-0) auto
  also have ... = ?R
    unfolding ε-def μ-alt by (simp add:power2-eq-square field-simps)
```

```
  finally show ?thesis by simp
qed

theorem (in prob-space) median-bound:
  fixes n :: nat
  fixes I :: ('b :: {linorder-topology, second-countable-topology}) set
  assumes interval I
  assumes α > 0
  assumes ε ∈ {0<..<1}
  assumes indep-vars (λ-. borel) X {0..<n}
  assumes n ≥ − ln ε / (2 * α^2)
  assumes ⋀i. i < n ⟹ P(ω in M. X i ω ∈ I) ≥ 1/2+α
  shows P(ω in M. median n (λi. X i ω) ∈ I) ≥ 1−ε
proof −
  have 0 < −ln ε / (2 * α^2)
    using assms by (intro divide-pos-pos) auto
  also have ... ≤ real n using assms by simp
  finally have real n > 0 by simp
  hence n-ge-0:n > 0 by simp

  have d0: real-of-int ⌊real n / 2⌋ * 2 / real n ≤ 1
    using n-ge-0 by simp linarith

  hence d1: real (nat ⌊real n / 2⌋) ≤ real n * (1 / 2)
    using n-ge-0 by (simp add:field-simps)
  also have ... ≤ real n * (1 / 2 + α)
    using assms(2) by (intro mult-left-mono) auto
  finally have d1: real (nat ⌊real n / 2⌋) ≤ real n * (1 / 2 + α) by simp

  have 1/2 + α ≤ P(ω in M. X 0 ω ∈ I) using n-ge-0 by (intro assms(6))
  also have ... ≤ 1 by simp
  finally have d2: 1 / 2 + α ≤ 1 by simp

  have d3: nat ⌊real n / 2⌋ = n div 2 by linarith

  have 1 − ε ≤ 1 − exp (− 2 * real n * α^2)
     using assms(2,3,5) by (intro diff-mono order.refl iffD1[OF ln-ge-iff]) (auto
simp:field-simps)
  also have ... ≤ 1 − exp (− 2 * real n * ((1/2+α) − real (nat ⌊real n/2⌋) / real
n)^2)
     using d0 n-ge-0 assms(2)
      by (intro diff-mono order.refl iffD2[OF exp-le-cancel-iff] mult-left-mono-neg
power-mono) auto
  also have ... ≤ 1 − measure (binomial-pmf n (1/2+α)) {..nat ⌊real n/2⌋}
    using assms(2) d1 d2 by (intro diff-mono order.refl binomial-pmf-tail) auto
  also have ... = 1 − measure (binomial-pmf n (1/2+α)) {..n div 2} by (simp
add:d3)
  also have ... ≤ P(ω in M. median n (λi. X i ω) ∈ I)
   using assms(2) by (intro median-bound-raw n-ge-0 assms(1,4,6) add-nonneg-nonneg)
```

auto
finally show *?thesis* **by** *simp*
qed

This is a specialization of the above to closed real intervals.

corollary (**in** *prob-space*) *median-bound-1*:
assumes $\alpha > 0$
assumes $\varepsilon \in \{0<..<1\}$
assumes *indep-vars* (λ-. *borel*) X $\{0..<n\}$
assumes $n \geq -\ ln\ \varepsilon\ /\ (2 * \alpha^2)$
assumes $\forall\, i \in \{0..<n\}.\ \mathcal{P}(\omega\ in\ M.\ X\ i\ \omega \in (\{a..b\} :: real\ set)) \geq 1/2+\alpha$
shows $\mathcal{P}(\omega\ in\ M.\ median\ n\ (\lambda i.\ X\ i\ \omega) \in \{a..b\}) \geq 1-\varepsilon$
using *assms*(5) **by** (*intro median-bound*[*OF - assms*(1,2,3,4)]) (*auto simp*:*interval-def*)

This is a specialization of the above, where $\alpha = \frac{1}{6}$ and the interval is described using a mid point μ and radius δ. The choice of $\alpha = \frac{1}{6}$ implies a success probability per random variable of $\frac{2}{3}$. It is a commonly chosen success probability for Monte-Carlo algorithms (cf. [2, §4] or [3, §1]).

corollary (**in** *prob-space*) *median-bound-2*:
fixes $\mu\ \delta$ :: *real*
assumes $\varepsilon \in \{0<..<1\}$
assumes *indep-vars* (λ-. *borel*) X $\{0..<n\}$
assumes $n \geq -18 * ln\ \varepsilon$
assumes $\bigwedge i.\ i < n \Longrightarrow \mathcal{P}(\omega\ in\ M.\ abs\ (X\ i\ \omega - \mu) > \delta) \leq 1/3$
shows $\mathcal{P}(\omega\ in\ M.\ abs\ (median\ n\ (\lambda i.\ X\ i\ \omega) - \mu) \leq \delta) \geq 1-\varepsilon$
proof −
have *b*:*space M* − {$\omega \in$ *space M*. *X i* $\omega \in \{\mu-\delta..\mu+\delta\}$} = {$\omega \in$ *space M*. *abs* (*X i* ω − μ) > δ} **for** *i*
by *auto*

have $\bigwedge i.\ i < n \Longrightarrow 1 - \mathcal{P}(\omega\ in\ M.\ X\ i\ \omega \in \{\mu-\ \delta..\mu+\delta\}) \leq 1/3$
using *assms*
apply (*subst prob-compl*[*symmetric*])
apply (*measurable*, *simp add*:*indep-vars-def*)
by (*subst b*, *simp*)

hence *a*:$\bigwedge i.\ i < n \Longrightarrow \mathcal{P}(\omega\ in\ M.\ X\ i\ \omega \in \{\mu-\ \delta..\mu+\delta\}) \geq 2/3$ **by** *simp*

have $1-\varepsilon \leq \mathcal{P}(\omega\ in\ M.\ median\ n\ (\lambda i.\ X\ i\ \omega) \in \{\mu-\delta..\mu+\delta\})$ **using** *a assms*(3)
by (*intro median-bound-1*[*OF - assms*(1,2), **where** α=1/6]) (*simp-all add*:*power2-eq-square*)
also have ... = $\mathcal{P}(\omega\ in\ M.\ abs\ (median\ n\ (\lambda i.\ X\ i\ \omega) - \mu) \leq \delta)$
by (*intro arg-cong2*[**where** *f*=*measure*] *Collect-cong*) *auto*
finally show *?thesis* **by** *simp*
qed

4 Some additional results about the median

lemma *sorted-mono-map*:

assumes *sorted xs*
assumes *mono f*
shows *sorted* (*map f xs*)
using *assms*(*2*) **unfolding** *sorted-wrt-map*
by (*intro sorted-wrt-mono-rel*[*OF - assms*(*1*)]) (*simp add*:*mono-def*)

This could be added to *HOL.List*:

lemma *map-sort*:
assumes *mono f*
shows *sort* (*map f xs*) = *map f* (*sort xs*)
using *assms* **by** (*intro properties-for-sort sorted-mono-map*) *auto*

lemma *median-cong*:
assumes $\bigwedge i.\ i < n \Longrightarrow f\ i = g\ i$
shows *median n f* = *median n g*
unfolding *median-def* **using** *assms*
by (*intro arg-cong2*[**where** *f*=(!)] *arg-cong*[**where** *f*=*sort*] *map-cong*) *auto*

lemma *median-restrict*:
median n ($\lambda i \in \{0..<n\}$.*f i*) = *median n f*
by (*rule median-cong*, *simp*)

lemma *median-commute-mono*:
assumes *n* > *0*
assumes *mono g*
shows *g* (*median n f*) = *median n* (*g* ∘ *f*)
apply (*simp add*: *median-def del*:*map-map*)
apply (*subst map-map*[*symmetric*])
apply (*subst map-sort*[*OF assms*(*2*)])
apply (*subst nth-map*, *simp*) **using** *assms* **apply** *fastforce*
by *simp*

lemma *median-rat*:
assumes *n* > *0*
shows *real-of-rat* (*median n f*) = *median n* ($\lambda i.$ *real-of-rat* (*f i*))
apply (*subst* (*2*) *comp-def*[**where** *g*=*f*, *symmetric*])
apply (*rule median-commute-mono*[*OF assms*(*1*)])
by (*simp add*: *mono-def of-rat-less-eq*)

lemma *median-const*:
assumes *k* > *0*
shows *median k* ($\lambda i \in \{0..<k\}.$ *a*) = *a*
proof −
have *b*: *sorted* (*map* (λ-. *a*) [*0*..<*k*])
by (*subst sorted-wrt-map*, *simp*)
have *a*: *sort* (*map* (λ-. *a*) [*0*..<*k*]) = *map* (λ-. *a*) [*0*..<*k*]
by (*subst sorted-sort-id*[*OF b*], *simp*)
have *median k* ($\lambda i \in \{0..<k\}.$ *a*) = *median k* (λ-. *a*)
by (*subst median-restrict*, *simp*)

also have ... = *a* **using** *assms* **by** (*simp add*:*median-def a*)
finally show *?thesis* **by** *simp*
qed

end

References

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[2] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *Randomization and Approximation Techniques in Computer Science*, pages 1–10. Springer Berlin Heidelberg, 2002.

[3] D. M. Kane, J. Nelson, and D. P. Woodruff. An optimal algorithm for the distinct elements problem. In *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '10, pages 41–52, New York, 2010.