Hello,

I propose Range#bsearch for binary search:

```ruby
ary = [0, 4, 7, 10, 12]
p (0..4).bsearch {|i| ary[i] >= 4 } #=> 1
p (0..4).bsearch {|i| ary[i] >= 6 } #=> 2

p (0.0...Float::INFINITY).bsearch {|x| Math.log(x) >= 0 } #=> 1.0
```

Rdoc:

```
call-seq:
  rng.bsearch {block}  -> element *

  - Returns the minimum value in range which meets the block by binary search.
  - The block must be monotone for arguments; in other words, it must have any
    following properties:
      - there is a value so that the block returns false for any smaller value
      - than the value, and that the block returns true for any bigger (or
        equal) value than the value.
      - the block always return true, or
      - the block always return false.
      - If the block is not monotone, the behavior is not unspecified.
    - Returns nil when there is no value that meets the block.

  ary = [0, 4, 7, 10, 12]
  (0...ary.size).bsearch {|i| ary[i] >= 4 } #=> 1
  (0...ary.size).bsearch {|i| ary[i] >= 6 } #=> 2
  (0...ary.size).bsearch {|i| ary[i] >= 8 } #=> 3
  (0...ary.size).bsearch {|i| ary[i] >= 100 } #=> nil
  (0.0...Float::INFINITY).bsearch {|x| Math.log(x) >= 0 } #=> 1.0

  Note that this method does not stop search immediately when the block
  returns true. This is because this method find the minimum value:

  ary = [0, 100, 100, 100, 200]
  i = (0..ary.size).bsearch {|i| condition(ary[i]) }
  p ary[i]

  matz hated Array#bsearch because its precondition (Array must be sorted)
  seems too strong (to him). [ruby-dev:17125]
  Range#bsearch has the same precondition in effect (the block must be
  monotone), but there is a difference that this condition is for "code",
  not "state". In fact, Array#sort has a similar condition.
```

Discussion:

You might think Array#bsearch is better. But Array#bsearch has some problems:

- Binary search is usable for not only an array but also a function.
  (See the above example for Math.log)
  Nevertheless, Array#bsearch can be used only for an array.
  Range#bsearch covers both. You can use it for an array as follows:

  ```ruby
  ary.sort!
  i = (0...ary.size).bsearch {|i| condition(ary[i]) }
  p ary[i]
  ```

- matz hated Array#bsearch because its precondition (Array must be sorted)
  seems too strong (to him). [ruby-dev:17125]
  Range#bsearch has the same precondition in effect (the block must be
  monotone), but there is a difference that this condition is for "code",
  not "state". In fact, Array#sort has a similar condition.
I think there is demand for this feature because similar feature requests are often proposed: [ruby-dev:17122] [ruby-core:30892] [ruby-dev:38545]

More importantly, this feature is often required in many programming contests :-)

A patch is attached. Thank you for your consideration.

---

Yusuke Endoh mame@tsg.ne.jp

| Related issues: |
|-----------------|----------------|
| Related to Ruby master - Bug #7352: Array#bsearch test failure on Range (32bi... | Closed 11/15/2012 |
| Has duplicate Ruby master - Feature #3479: Array#binary_find et al | Closed 06/25/2010 |

**Associated revisions**

**Revision de07850e - 11/14/2012 03:53 PM - mame (Yusuke Endoh)**

- array.c (rb_ary_bsearch): add Array#bsearch for binary search. [ruby-core:36390] [Feature #4766]
- test/ruby/test_array.rb: add a test for above.
- range.c (range_bsearch): add Range#bsearch for binary search. [ruby-core:36390] [Feature #4766]
- test/ruby/test_range.rb: add a test for above
- NEWS: added the two new methods.

**Revision 37655 - 11/14/2012 03:53 PM - mame (Yusuke Endoh)**

- array.c (rb_ary_bsearch): add Array#bsearch for binary search. [ruby-core:36390] [Feature #4766]
- test/ruby/test_array.rb: add a test for above.
- range.c (range_bsearch): add Range#bsearch for binary search. [ruby-core:36390] [Feature #4766]
- test/ruby/test_range.rb: add a test for above
- NEWS: added the two new methods.

**Revision 37655 - 11/14/2012 03:53 PM - mame (Yusuke Endoh)**

- array.c (rb_ary_bsearch): add Array#bsearch for binary search. [ruby-core:36390] [Feature #4766]
- test/ruby/test_array.rb: add a test for above.
- range.c (range_bsearch): add Range#bsearch for binary search. [ruby-core:36390] [Feature #4766]
- test/ruby/test_range.rb: add a test for above
- NEWS: added the two new methods.
array.c (rb_ary_bsearch): add Array#bsearch for binary search.
[ruby-core:36390] [Feature #4766]

test/ruby/test_array.rb: add a test for above.

range.c (range_bsearch): add Range#bsearch for binary search.
[ruby-core:36390] [Feature #4766]

test/ruby/test_range.rb: add a test for above

NEWS: added the two new methods.
range.c (range_bsearch): fix some bugs: a documentation bug, a wrong condition, missed break in switch/case, and workaround for GCC optimization. See [ruby-core:49364] in detail. A great patch from Heesob Park. [Bug #7352] [Feature #4766]

array.c (rb_ary_bsearch): fix similar bug (missed break).

test/ruby/test_range.rb: add two test cases for above.

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@37662 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 37662 - 11/15/2012 01:50 PM - mame (Yusuke Endoh)

range.c (range_bsearch): fix some bugs: a documentation bug, a wrong condition, missed break in switch/case, and workaround for GCC optimization. See [ruby-core:49364] in detail. A great patch from Heesob Park. [Bug #7352] [Feature #4766]

array.c (rb_ary_bsearch): fix similar bug (missed break).

test/ruby/test_range.rb: add two test cases for above.

Revision 37662 - 11/15/2012 01:50 PM - mame (Yusuke Endoh)

range.c (range_bsearch): fix some bugs: a documentation bug, a wrong condition, missed break in switch/case, and workaround for GCC optimization. See [ruby-core:49364] in detail. A great patch from Heesob Park. [Bug #7352] [Feature #4766]

array.c (rb_ary_bsearch): fix similar bug (missed break).

test/ruby/test_range.rb: add two test cases for above.

Revision 37662 - 11/15/2012 01:50 PM - mame (Yusuke Endoh)

range.c (range_bsearch): fix some bugs: a documentation bug, a wrong condition, missed break in switch/case, and workaround for GCC optimization. See [ruby-core:49364] in detail. A great patch from Heesob Park. [Bug #7352] [Feature #4766]

array.c (rb_ary_bsearch): fix similar bug (missed break).

test/ruby/test_range.rb: add two test cases for above.
Heesob Park. [Bug #7352] [Feature #4766]

- array.c (rb_ary_bsearch): fix similar bug (missed break).
- test/ruby/test_range.rb: add two test cases for above.

Revision 37662 - 11/15/2012 01:50 PM - mame (Yusuke Endoh)

- range.c (range_bsearch): fix some bugs: a documentation bug, a wrong condition, missed break in switch/case, and workaround for GCC optimization. See [ruby-core:49364] in detail. A great patch from Heesob Park. [Bug #7352] [Feature #4766]
- array.c (rb_ary_bsearch): fix similar bug (missed break).
- test/ruby/test_range.rb: add two test cases for above.

Revision 37662 - 11/15/2012 01:50 PM - mame (Yusuke Endoh)

- range.c (range_bsearch): fix some bugs: a documentation bug, a wrong condition, missed break in switch/case, and workaround for GCC optimization. See [ruby-core:49364] in detail. A great patch from Heesob Park. [Bug #7352] [Feature #4766]
- array.c (rb_ary_bsearch): fix similar bug (missed break).
- test/ruby/test_range.rb: add two test cases for above.

History

#1 - 05/23/2011 10:23 AM - mrkn (Kenta Murata)

Hi,

On 2011年5月23日月曜日 at 1:52, Yusuke Endoh wrote:

You might think Array#bsearch is better. But Array#bsearch has some problems:
I think Range is better position than Array in which the bsearch method is, too.

I want this feature.

--
Kenta Murata
Sent with Sparrow

#2 - 06/01/2011 04:20 PM - matz (Yukihiro Matsumoto)

- Status changed from Open to Rejected

I seriously considered about the issue, but I feel something wrong about this feature proposal. Binary search requires the target to be ordered. In that sense, Arrays in general are not ordered, ranges are appeared to be ordered. But "order" is in the eyes of beholder. Once you supply the block, the ordered attribute of a range does not mean anything. Besides that, as the example in the document shows, Range#bsearch is more indirect than say, Array#bsearch.

If we have to assume ordered status of the target anyway,
min = ary.bsearch{|i| i > 4}
is much better than
n = (0..ary.size).bsearch{|i| ary[i] > 4}
min = ary[n]
isn't it?

matz.
Thank you for your time.

Once you supply the block, the ordered attribute of a range does not mean anything.

I'm not sure if I could understand this correctly. If the block handles its parameter in terms of the ordered attribute, is there no problem?

Else, any method must work even if a supplied block ignores the context? Array#sort violates the rule, I think.

Besides that, as the example in the document shows, Range#bsearch is more indirect than say, Array#bsearch.

Yes, for typical cases. But Array#bsearch is not enough since it cannot be used over Float or big range.

Anyway, I'm NOT against Array#bsearch. It is a good thing. But You rejected it. Now do you accept Array#bsearch?

Binary search is a practical matter than philosophy. Hope my proposal to get reconsidered.

--

Yusuke Endoh mame@tsg.ne.jp

#4 - 06/01/2011 08:23 PM - rkh (Konstantin Haase)
It might make sense for SortedSet (part if the stdlib), though.

Konstantin

On Jun 1, 2011, at 13:07 , Yusuke Endoh wrote:

Issue #4766 has been updated by Yusuke Endoh.

Target version changed from 1.9.3 to 1.9.x

Thank you for your time.

Once you supply the block, the ordered attribute of a range does not mean anything.

I'm not sure if I could understand this correctly. If the block handles its parameter in terms of the ordered attribute, is there no problem?

Else, any method must work even if a supplied block ignores the context? Array#sort violates the rule, I think.

Besides that, as the example in the document shows, Range#bsearch is more indirect than say, Array#bsearch.

Yes, for typical cases. But Array#bsearch is not enough since it cannot be used over Float or big range.

Anyway, I'm NOT against Array#bsearch. It is a good thing. But You rejected it. Now do you accept Array#bsearch?

Binary search is a practical matter than philosophy. Hope my proposal to get reconsidered.

--

Yusuke Endoh mame@tsg.ne.jp
Hello,

I propose Range#bsearch for binary search:

```ruby
#5 - 07/17/2011 12:25 AM - mame (Yusuke Endoh)
- Status changed from Rejected to Assigned
- Assignee changed from matz (Yukihiro Matsumoto) to mame (Yusuke Endoh)

Hello,

Today I talked with matz and got his approval about this ticket, under the following two condition:

- commit it to trunk
- provide not only Range#bsearch but also Array#bsearch
- avoid C-level undefined behavior (e.g., SEGV)

Once matz expressed opposition because Array#bsearch may return unspecified result when the array is not sorted. But Array#sort may also return unspecified result when the block is inconsistent. So Array#bsearch is not so strange API in the aspect, and actually useful. So matz approved it eventually.

So I'm changing this ticket to accepted, and will write a patch. If anyone has any idea, feel free to express your opinions.

--
Yusuke Endoh mame@tsg.ne.jp

#6 - 07/17/2011 08:36 AM - trans (Thomas Sawyer)

How is this useful? It's basically "find minimum" value, right? What difference does it make that it uses a binary search?

If binary searching is important, then why not have a #beach --which can then be used to extrapolate any variety of other methods.

#7 - 07/17/2011 10:23 AM - mame (Yusuke Endoh)

How is this useful? It's basically "find minimum" value, right? What difference does it make that it uses a binary search?

If binary searching is important, then why not have a #beach --which can then be used to extrapolate any variety of other methods.

Maybe I'm not sure your point. It finds minimum value by using binary search. Does the name bsearch' matter? Isbeach' really suitable?

I add a practical example of Range#bsearch for Float domain. The following code obtains the interior radius between 3 circles of radii a,b,c that are circumscribed each other, by using law of cosines three times:

```ruby
(0.0 .. [a,b,c].min).bsearch do |t|
  u, v = (t+a+b)**t, a**b
  s = Math.acos((u - v) / (u + v))
  u, v = (t+b+c)**t, b**c
  s += Math.acos((u - v) / (u + v))
  u, v = (t+c+a)**t, c**a
  s += Math.acos((u - v) / (u + v))
  s <= Math::PI * 2
end
```

--
Hi, thanks for example. I may not fully understand bsearch, but basically what I mean is that if finding minimum value is the important thing, then why not call it #find_minimum? (and what about #find_maximum?). But if binary search is what is important than could not a #beach (b-each) be combined with a variety of uses.

to_enum(:beach).min
to_enum(:beach).max
to_enum(:beach).map
 etc.

The method is not about finding the minimum, it's about binary search. It has the additional behavior of finding the minimum value at the boundary condition for the block. If multiple values match the boundary condition the minimum is chosen.

Read the example in the proposed RDoc

I understand you. In principle, a method name should not represent how it does but what it does. But in this case, the feature has a strict pre-condition: the array must be sorted. So, the name #find_minimum is too generic. We may call it #find_minimum_from_sorted_array or so, but I believe #bsearch is more suitable.

---

No, it's not about finding any minimum value, it's about binary search:

I propose Range#bsearch for binary search

and

Returns [...] by binary search

The important information in English does not necessarily come first. To make it more clear maybe it should read:

Performs a binary search using the block as criteria. If multiple items match the criteria the minimum (first) value is chosen.

You do know what a "binary search" is, right?

http://en.wikipedia.org/wiki/Binary_search

When performing a binary search you may have multiple matching values, the three 100 values from the example. For the proposed implementation the first one is chosen for consistency.

Nobody knows what the #beach method you keep talking about would be. Maybe you mean binary-search-each? That doesn't make any sense as binary search will not yield each item in the enumerator. If you think that this method will yield each item please read about binary search first.
When performing a binary search you may have multiple matching values, the three 100 values from the example. For the proposed implementation the first one is chosen for consistency.

Nobody knows what the #beach method you keep talking about would be. Maybe you mean binary-search-each? That doesn't make any sense as binary search will not yield each item in the enumerator.

I wouldn't say "nobody". I think what he's suggesting is a more flexible binary-search-each which yields all elements for which the block is true. Hear this out:

The current implementation's result is equivalent to Enumerable#min, except it requires sorted order so it can use binary search. This adds the asymptotic speedup to the #min method.

This is handy, but what if someone wanted the maximum? Or just the first element that matches?

Those are equivalent, respectively, to Enumerable#max and Enumerable#find. Personally, I think I'd usually just want to get the first one that matches, and not use extra cycles computing the minimum. There's room for choice which the current implementation lacks. Further, they can't be built based on the new method. They have to be written from scratch.

These could be implemented with a base, binary_matches method, with selection logic added on. Assume binary_matches takes a block, and returns an enumerator for all elements for which the block yields true. It can do so in O(log(n)) time. This is just a thrown together API, a better API likely exists. If you had such a method, you could implement the existing proposal (min), max, and find as such, giving all the log(n) speedup:

```ruby
# The current bsearch proposal renamed to bsearch_min.
def bsearch_min(&blk)
    binary_matches(&blk).min
end

# Returns the largest elt for which the block yields true
def bsearch_max(&blk)
    binary_matches(&blk).max
end

# Returns the first-discovered elt for which the block yields true
def bsearch_find(&blk)
    binary_matches(&blk).each do |elt|
        return elt if blk.call(elt)
    end
end
```

This seems useful to me.

Michael Edgar
adgar@carboni.ca
http://carboni.ca/

---

Returns the first-discovered elt for which the block yields true

```ruby
def bsearch_find(&blk)
    binary_matches(&blk).each do |elt|
        return elt if blk.call(elt)
    end
end
```

Whoops, this is silly. It should be as follows:

```ruby
def bsearch_find(&blk)
    binary_matches(&blk).first
end
```

I rewrote the first two methods when I came up with the strawman binary_matches API, and forgot to rewrite the third.

Michael Edgar
adgar@carboni.ca
http://carboni.ca/
Hello,

Thank you for all your opinions!

2011/7/18 Michael Edgar adgar@carboni.ca:

This is handy, but what if someone wanted the maximum?

Please invert the condition. For example, if you want the maximum index holding that the element is less than 7:

    ary = [0, 4, 7, 10, 12]
    p (0..4).bisect {|i| !(ary[i] < 7) } #=> 1

Indeed it is not so cool, but similar to Array#sort. If it is possible only by inverting, I'm afraid if we need particular methods for each use case.

    Or just the first element that matches?

Good question :-) Interestingly, break can be used.

    ary = [0, 4, 7, 10, 12]
    p (0..4).bisect_matches {|i| break i if ary[i] >= 4 }
    #=> 1, 2, 3 or 4 (probably 2, but I don't think it might be changed in future extension)

an enumerator for all elements for which the block yields true.

I agree that it seems neat, but it is ambiguous to me how it works concretely. What do you expect the following code outputs?

    ary = [0, 4, 7, 10, 12]
    p (0..4).bisect_matches {|i| ary[i] >= 4 }.to_a #=> ?

And, I guess binary_matches(&blk).min takes O(m) where m is the number of results because Enumerable#min and #max find the value by iterating all elements.

--
Yusuke Endoh mame@tsg.ne.jp

Oops,

2011/7/18 Yusuke ENDOH mame@tsg.ne.jp:

    Or just the first element that matches?

    Good question :-) Interestingly, break can be used.

    ary

#16 - 07/18/2011 02:59 PM - mame (Yusuke Endoh)

Oops,

2011/7/18 Yusuke ENDOH mame@tsg.ne.jp:

    Or just the first element that matches?

    Good question :-) Interestingly, break can be used.

    ary

#17 - 07/18/2011 03:23 PM - Anonymous

On Jul 18, 2011, at 1:30 AM, Yusuke ENDOH wrote:

    I agree that it seems neat, but it is ambiguous to me how it works concretely. What do you expect the following code outputs?

I see now my misunderstanding; the "minimum value" heuristic exists not to provide additional utility, but merely to direct tie-breaking in the binary search in a consistent manner.

My proposed bisect_matches would clearly need to know which direction to break ties before it commenced (perhaps a 1/0/-1
argument), which starts to limit its usefulness. While I personally would prefer this as a base implementation for several methods to bsearch with inverted block condition or "break" in the block, I can't honestly say my idea would be worth it for a first implementation.

It does still irk me that the general name "bsearch" would have the minimum-tie-breaking built-in, without that being reflected in the name or an argument. The additional behavior may surprise users.

Michael Edgar
adgar@carboni.ca
http://carboni.ca/

#18 - 07/20/2011 11:21 AM - mame (Yusuke Endoh)
- File bsearch.patch added

Hello,

Let me summarize. There are two typical use cases for bsearch: from array sorted in ascending order,

1) find minimum i so that s <= a[i]
2) find any i so that s == a[i]

Case 2 is the point I overlooked; "find minimum" is not inherent in binary search.

In fact, the API that I proposed covers both use cases.

1) r.bsearch [i | s <= a[i] ]
2) r.bsearch [i | break i if s == a[i]; s < a[i] ]

But,

My proposed bsearch_matches would clearly need to know which direction to break ties before it commenced (perhaps a 1/0/-1 argument), which starts to limit its usefulness.

indeed, Case 2 can be shorter by using 1/0/-1.

2) r.bsearch [i | s - a[i] ]

As Michael said, it is too cryptic for casual use, but certainly more generic, and compatible to bsearch of stdlib.h.

How about combining 1/0/-1 and my proposal? That is,

if the block returns zero, the current value satisfies the condition; return the current value immediately

if the block returns an integer less than zero, the current value is too big to satisfy the condition; continue to search smaller values

if the block returns an integer greater than zero, the current value too big to satisfy the condition; continue to search larger values

if the block returns true, the current value satisfies the condition; continue to search minimum bound

if the block returns false or nil, same as -1

It is still needed to invert the condition when you want the maximum bound, but I guess this is acceptable compromise.

I updated rdoc and a patch. As you know I'm not a native speaker, so could you check it?

/*
call-seq:
rng.bsearch {obj} block } -> element
By using binary search, finds a value in range which meets the given
condition in O(n log n) where n = (rng.begin - rng.end).
The given block receives a current value, determines if it meets the
condition and controls search.
When the condition is satisfied and you want to stop search, the block
should return zero, and then this method return the value immediately.
When the condition is satisfied and you want to find minimum bound,
the block should return true. When the condition is not satisfied and
the current value is smaller than wanted, the block should return false,
il or an integer greater than zero. When the condition is not satisfied
and the current value is larger than wanted, the block should return an
integer less than zero.
Unless the block returns zero, the search will continue until a minimum
bound is found or no match is found. Returns the minimum bound if any,
or returns nil when no match is found.
The block must be monotone; there must be two values a and b so that
the block returns:
- false, nil or an integer greater than zero for all x of [begin of
range, a), and
- zero or true for all x of [a, b), and
- an integer less than zero for all x of [b, end of range),
If the block is not monotone, the result is unspecified.
This method takes O(n log n), but it is unspecified which value is
actually picked up at each iteration.
ary = [0, 4, 7, 10, 12]
(0...ary.size).bsearch {i|ary[i] >= 4 } #=> 1
(0...ary.size).bsearch {i|ary[i] >= 6 } #=> 2
(0...ary.size).bsearch {i|ary[i] >= 8 } #=> 3
(0...ary.size).bsearch {i|ary[i] >= 100 } #=> nil
(0...Float::INFINITY).bsearch {x|Math.log(x) >= 0 } #=> 1.0
ary = [0, 100, 100, 100, 200]
(0..4).bsearch {i| 100 - i } #=> 1, 2 or 3
(0..4).bsearch {i| 300 - i } #=> nil
(0..4).bsearch {i| 50 - i } #=> nil */

Thanks,
--
Yusuke Endoh mame@tsg.ne.jp

#19 - 07/20/2011 11:21 AM - mame (Yusuke Endoh)
- File deleted (range-bsearch.patch)

#20 - 04/06/2012 02:25 AM - rogerdpack (Roger Pack)
Hope this can be committed at some point...

#21 - 10/27/2012 06:11 AM - ko1 (Koichi Sasada)
ping. status?

#22 - 11/15/2012 11:37 AM - mame (Yusuke Endoh)
- Status changed from Assigned to Closed
- % Done changed from 0 to 100

This issue was solved with changeset r37655.
Yusuke, thank you for reporting this issue.
Your contribution to Ruby is greatly appreciated.
May Ruby be with you.
NEWS: added the two new methods.

#23 - 11/15/2012 04:52 PM - usa (Usaku NAKAMURA)
- Status changed from Closed to Assigned

In range.c, the definition of macro BSEARCH_CHECK includes:

```c
switch (rb_cmpint(rb_funcall(v, id_cmp, 1, INT2FIX(0)), v, INT2FIX(0)) < 0) { \
    case 0: return val; \
    case 1: smaller = 1; \n    case -1: smaller = 0; \n}
```

But, in C, the value of compare expression is only 0 or 1, so this code is nonsense.
What do you want to do here?

#24 - 11/15/2012 05:02 PM - usa (Usaku NAKAMURA)
- Status changed from Assigned to Closed

Oh, Park-san perfectly pointed out this problem in [ruby-core:49364].
So I close this ticket.

Files

| bsearch.patch | 12.9 KB | 07/20/2011 | mame (Yusuke Endoh) |