Big-Oh + Divide and Conquer

CMPUT 275 - Winter 2018

University of Alberta

Big-Oh Notation

We have been using $O()$ notation frequently to discuss the running times of our algorithms.

But we have never seen a **proper** definition! Understanding this is important, especially because $O()$ notation is used in other places than run-time analysis:

- **Function approximation**: one can say $\sum_{k=1}^{n} \frac{1}{k} = \ln n + O(1)$, the error between the sum and $\ln n$ is bounded by a constant.
- **Convergence rates**: many numerical algorithms only get "close" to the intended solution, with a tradeoff between running time and quality (example later this lecture).

Big-Oh Notation

Before defining $O()$, we have to understand what it applies to.

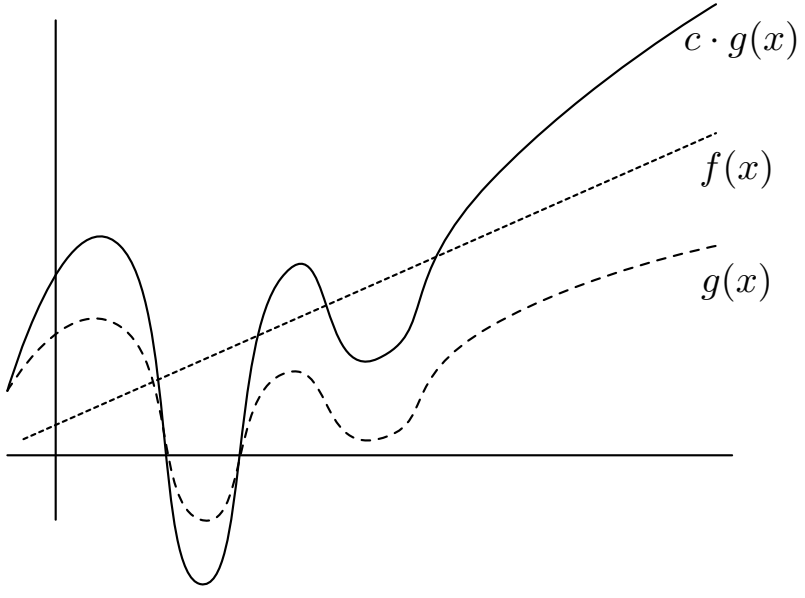
Definition
Call a function f that maps some values $x \in \mathbb{N}$ (the natural numbers, including 0) to a value $f(x) \in \mathbb{R}$ **eventually positive** if there is some value N such that $f(x)$ is defined and is positive for all $x > N$.

Examples:

- $f(x) = \ln x$, choosing $N = 1$ suffices
- $f(x) = x - 3$, choosing $N = 3$ suffices
- $f(x) = x^2 - 10000 \cdot \sqrt{x}$, choosing $N = (10000)^{2/3}$ suffices

Big-Oh Notation

Let g be an eventually-positive function. $O(g)$ is the **set** of eventually-positive functions f such that there exists constants $c, N > 0$ satisfying $f(x) \leq c \cdot g(x)$ for all $x \geq N$.



The value N should be at least when both f and g become positive.

Intuitively it means f is eventually dominated by some multiple of g.

Examples

Earlier we would say things like $3x^2 + 2$ "is" $O(x^2)$. Let's check this carefully!

Pick $c = 4, N = 2$. The claim is then:

$$3x^2 + 2 \leq 4x^2$$

for all $x \geq 2$.

This is equivalent to saying $2 \leq x^2$ for all $x \geq 2$, which is the case!

Properly Speaking: Thus, $3x^2 + 2 \in O(x^2)$.

Examples

Consider 8^n versus $n!$, which is $O()$ of the other?

Idea: Observe

$$8^n = 8 \cdot 8 \cdot 8 \cdot \ldots \cdot 8$$

and

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n.$$

For all but the first 8 terms in both products, the term in $n!$ is at least the corresponding term in 8^n.

So choosing $c = 8^8, N = 8$ more than suffices: for $n \geq 8$:

$$8^8 \cdot n! \geq 8^8 \cdot 9 \cdot 10 \cdot \ldots \cdot n \geq 8^8 \cdot 8^{n-8} = 8^n.$$

Examples

Show $x \in O(x^{1.001} - 10^9 \cdot x)$.

Discovering c might be a bit of a nuisance, but the polynomial in the $O()$ has a (slightly) larger exponent so intuitively we think any $c > 0$ should work. Let's try $c = 1$.

If $x \leq x^{1.001} - 10^9 \cdot x$, then $(1 + 10^9) \cdot x \leq x^{1.001}$. Cancelling x,

$$1 + 10^9 \leq x^{0.001}.$$

Exponentiating:

$$(1 + 10^9)^{1000} \leq x$$

This holds for all $x \geq N := (1 + 10^9)^{1000}$.

Cousins of $O()$: Little-Oh

For an eventually-positive function g we say $o(g)$ (little-Oh) is the set of all functions f such that for **any** $c > 0$, there is some N with $f(x) \leq c \cdot g(x)$.

(**Note**: I'm going to stop saying eventually positive and just assume it is true for all functions we talk about).

Intuitively, f is (eventually) negligible when compared to g.

Example
Show $\sqrt{x} \in o(x)$.

For any $c > 0$, saying $\sqrt{x} \leq c \cdot x$ means $x \geq 1/c^2$. So take $N = 1/c^2$.

Cousins of $O()$: Big-Omega

We say $\Omega(g)$ (big-Omega) is the set of all functions f such that there are constants $c, N > 0$ where $f(x) \geq c \cdot g(x)$ for all $x \geq N$.

Nothing really new to show here: $f \in O(g)$ if and only if $g \in O(f)$. Why?

If $f(x) \leq c \cdot g(x)$ for all $x \geq N$ then $g(x) \geq \frac{1}{c} \cdot f(x)$ for all $x \geq N$ (recall $c > 0$). The converse holds too.

Example: $n^3 \in \Omega(n^2 \log n)$.
Because eventually $n \geq \log_2 n$ for $n \geq 1$ so we can just take $c = 1$ and $N = 1$.

Cousins of $O()$: Little-Omega

We say $\omega(g)$ (little-Omega) is the set of all functions f such that for any constant $c > 0$ there is some $x \geq N$ such that $f(x) \geq c \cdot g(x)$ for $x \geq N$.

Again, nothing really new to show here: $f \in \omega(g)$ if and only if $g \in o(f)$.

Example: $n^3 \in \omega(n^2 \log n)$.
Because for any $c > 0$ we eventually have $n \geq c \log_2 n$.

Cousins of $O()$: Theta

We say $\Theta(g)$ (Theta) is the set of all functions f such that $f \in O(g)$ and $g \in O(f)$.

That is, saying $f \in \Theta(g)$ means they have the same **asymptotic growth rate**.

Examples

- $n^3 \notin \Theta(n^2 \log n)$
- $0.0001 \cdot n^3 + 2n + 1 \in \Theta(n^3)$.

Summary

For two **eventually-positive** functions f, g:

- $f \in O(g)$ if for **some** $c > 0$, eventually $f(x) \leq c \cdot g(x)$.
- $f \in o(g)$ if for **any** $c > 0$, eventually $f(x) \leq c \cdot g(x)$.
- $f \in \Omega(g)$ if $g \in O(f)$.
- $f \in \omega(g)$ if $g \in o(f)$.
- $f \in \Theta(g)$ if $f \in O(g)$ and $f \in \Omega(g)$.

No Little-Oh Analog of Theta

Claim
No two functions f, g satisfy $f \in o(g)$ and $g \in o(f)$.

Why? Just work with the definitions.
If there was such f, g, then for, say, $c = 2$ there is some N where:

- $f(x) \geq 2 \cdot g(x)$ for $x \geq N$
- $g(x) \geq 2 \cdot f(x)$ for $x \geq N$

But then for $x \geq N$.

$$f(x) \geq 2 \cdot g(x) \geq 4 \cdot f(x).$$

This is impossible as $f(x) > 0$ for large enough x.

One More Example

Show $n \log_2 n = \Theta(\log_2 n!)$.

On one hand, for $n \geq 1$:

$$\log_2 n! = \sum_{k=1}^{n} \log_2 k \leq \sum_{k=1}^{n} \log_2 n = n \log_2 n.$$

So taking $c = 1$ and $N = 1$ in the definition of $O()$ shows $\log_2 n! \in O(n \log n)$.

Conversely, for $n \geq 1$:

$$\log_2 n! = \sum_{k=1}^{n} \log_2 k \geq \sum_{n/2 \leq k \leq n} \log_2 \frac{n}{2} \geq \frac{n}{2} \log_2 \frac{n}{2} = \frac{n}{2} \log_2 n - \frac{n}{2}.$$

One More Example

We just saw:
$\log_2 n! \geq \frac{n}{2} \log_2 n - \frac{n}{2}$.

For $n \geq 4$ we have $\log_2 n \geq 2$ so $\frac{n}{2} \leq \frac{1}{4} n \log_2 n$.

Plugging this in above:
$\log_2 n! \geq \frac{n}{2} \log_2 n - \frac{n}{4} \log_2 n = \frac{n}{4} \log_2 n$.

So by taking $c = 4$ and $N = 4$, we see $\log n! \in \Omega(n \log n)$.

Collectively, these two $O()$ and $\Omega()$ bounds show
$n \log_2 n \in \Theta(\log_2 n!)$.

Why do we worry about the precise definitions?

Well, some running times are strange. There are algorithms with running times bounds of:

- $O(n \cdot 1.71^n)$.
- $O(n^2 \cdot 2^{\sqrt{n}})$.

Which function grows slowest?
Upon first glance, it takes a moment to see which one will perform better. The definitions provide concreteness in our arguments.

Another common question: is $2^n = \Theta(3^n)$?

An ugly bound for the (known) fastest integer factoring (non-quantum) algorithm
`http://en.wikipedia.org/wiki/General_number_field_sieve`

Conventions

A common abuse. People frequently write:

$$f = O(g)$$

in place of

$$f \in O(g).$$

Also, people typically say the running time of a function **is** $O(f)$ as opposed to the more cumbersome statement that the running time **lies in** $O(f)$.

I do this all the time. These are a socially-acceptable conventions :) Just don't forget the core definitions!

More Justification

Why do we use $O()$ notation when discussing an algorithm? Isn't real performance what matters?

Well, yes. At the end of the day, running fastest **in practice** is what we want.

But if your starting point is an algorithm with a worse $\Theta()$ running time bound, you have no hope to optimize it and beat an algorithm with a faster $\Theta()$ running time bound if you expect large input.

Sometimes we intentionally opt for asymptotically slower algorithms on small input. In `C++`, bubble sort is used if the array to be sorted has size ≤ 5 because it runs faster for small inputs.

Another motivation: while we can spend some time optimizing for practical performance, it is still challenging to know precise constants.

There are a number of factors perhaps beyond our control:

- **Compiler**: The instructions produced when through compilation.
- **Hardware**: Quality of the bus, cache, CPU itself!
- **Operating System**: How is the memory space of the program mapped to physical memory? How many processes are running?

So $O()$ notation more to gives a sense of how the running time will scale. Using an algorithm with better $O()$ guarantees is frequently a good starting point.

Of course, there is lively discussion around which of the "fastest" algorithms to choose: AVL trees vs. Red-Black trees; merge sort vs. quick sort vs. heap sort.

Divide and Conquer

Now we switch to studying an algorithm design paradigm: divide-and-conquer algorithms.

The idea is that you split the problem into smaller parts, solve those parts, an (perhaps) spend some time recombining the solution.

Examples You Have Already Seen:

Intro By Example

There is no strict definition of **divide and conquer**, it's more that the term evokes a sense of what the algorithm is doing.

So we introduce using the `hello world` of divide-and-conquer algorithms:

Binary Search

Basic Problem
You have a list of integers that is already sorted. Given a **query** integer q, determine if q is in the list and, if so, where?

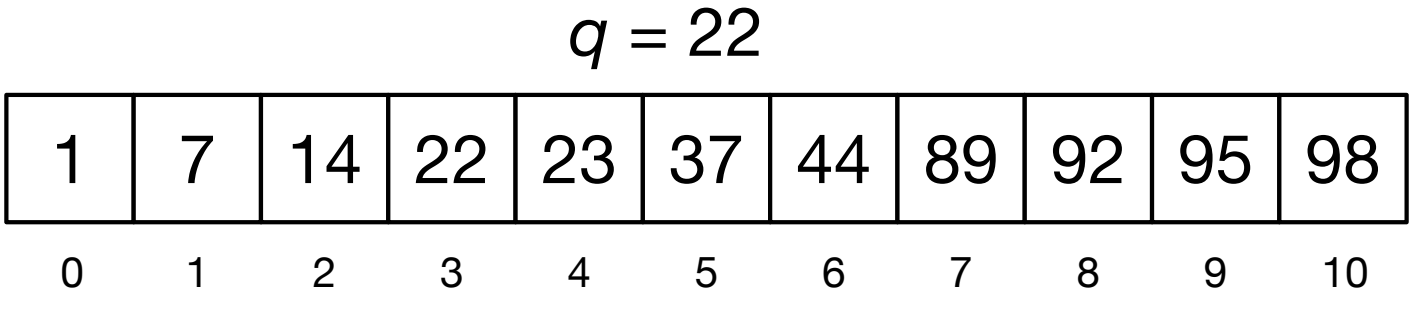
Obviously we can scan the list linearly and find it in $O(n)$ time.

We can answer the query much faster than this!

Divide

Query the middle of the list, say the value stored there is *mid*.

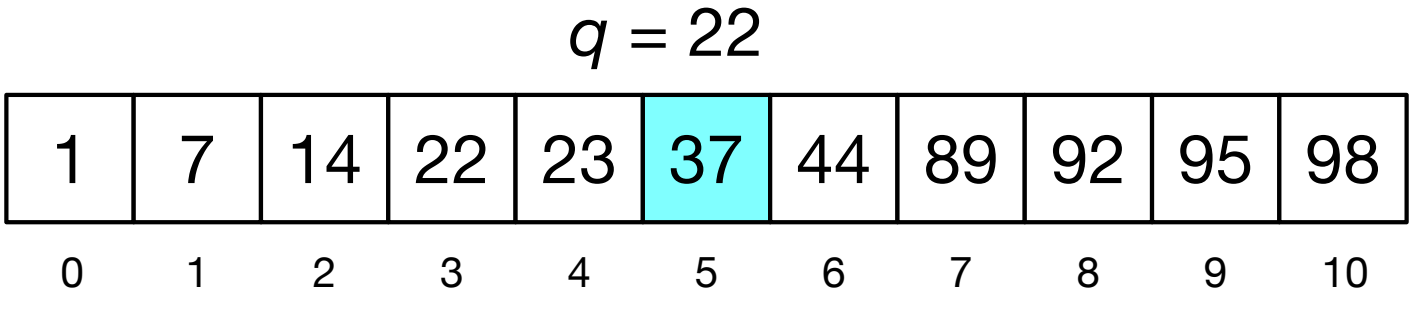
- If $q == mid$, great! We are done!
- If $q < mid$, then **if** q is in the list it must be in the first half.
- If $q > mid$, then **if** q is in the list it must be in the second half.



Divide

Query the middle of the list, say the value stored there is *mid*.

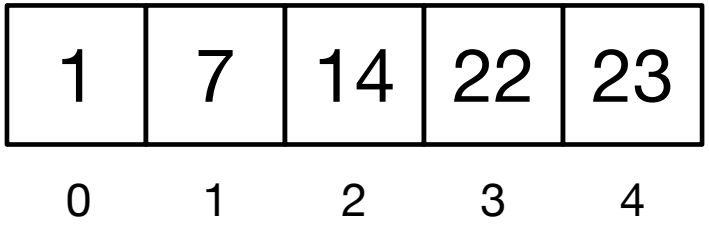
- If $q == mid$, great! We are done!
- If $q < mid$, then **if** q is in the list it must be in the first half.
- If $q > mid$, then **if** q is in the list it must be in the second half.



And Conquer

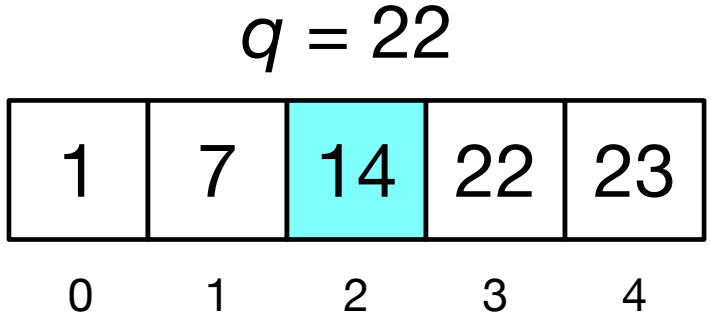
Repeat, but only with the half of the list containing the number.

$q = 22$



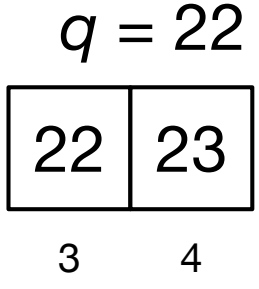
And Conquer

Repeat, but only with the half of the list containing the number.



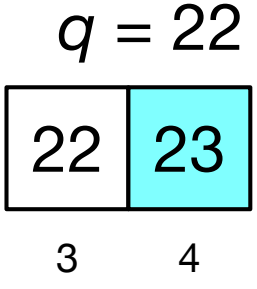
And Conquer

Repeat, but only with the half of the list containing the number.



And Conquer

Repeat, but only with the half of the list containing the number.



And Conquer

Repeat, but only with the half of the list containing the number.

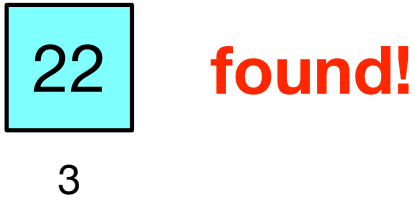
$q = 22$

3

And Conquer

Repeat, but only with the half of the list containing the number.

$q = 22$



The number of steps is at most $\log_2 n$.

That is, each step that does not find q will cut the size of the list to search in half.

Important Note
Don't actually splice the list. A single splice in the middle of the list takes $\Omega(n)$ time. Just maintain an index referring to the start and length of the list you have yet to search.

The pictures were just showing the relevant part of the list to search, but it was the original list.

Algorithm 1 Simple binary search of a list L for an integer q.

```
low, num ← 0, len(L)
while num > 0 do
    mid ← num/2                    # integer division: the quotient
    if q == L[low + mid] then
        return low + mid
    else if q < L[low + mid] then
        num ← mid
    else
        low, num ← low + mid + 1, num − mid − 1
return not found
```

Invariant to Maintain

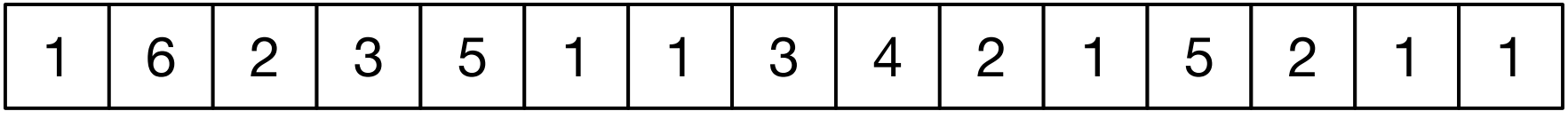
If q is in ℓ, it lies between indices *low* and $low + num - 1$.

Progress: Each iteration replaces *num* with a value at most $num/2$.

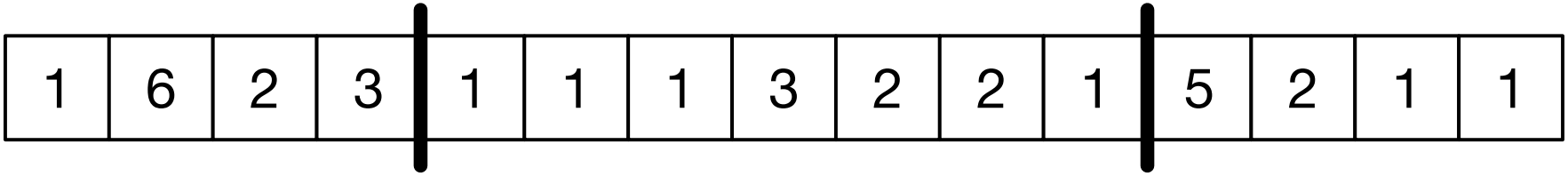
Other Uses of Binary Search

Problem: Balanced Intervals

Given a list of integers $a_1, \ldots, a_n$ and a value k, the goal is to divide the list into k different consecutive intervals (see the picture) in a way that minimizes the maximum total value of an interval.



Example: split into $k = 3$ intervals with maximum total value 12.

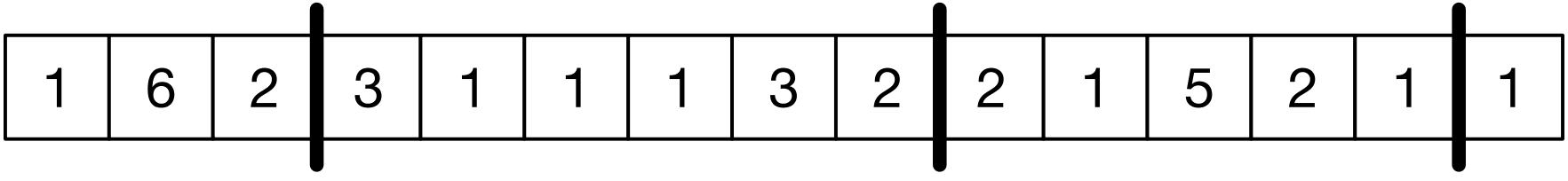


Maximum total value 11 is impossible (as we shall see).

Balanced Intervals

Idea: "flip" the question. If we are only allowed intervals with total value x, what is the minimum number of intervals we can divide the list into?

Pictured: With $x = 11$ the minimum number is 4 intervals.



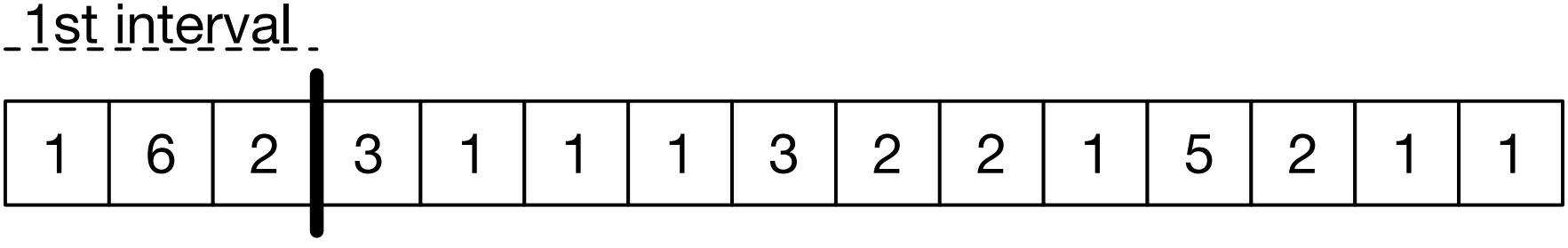
This seems much easier.

Greedily make the leftmost interval as long as possible. Repeat until done.

Balanced Intervals

While not all integers are covered:

- Build the longest possible interval, starting from the first uncovered number.



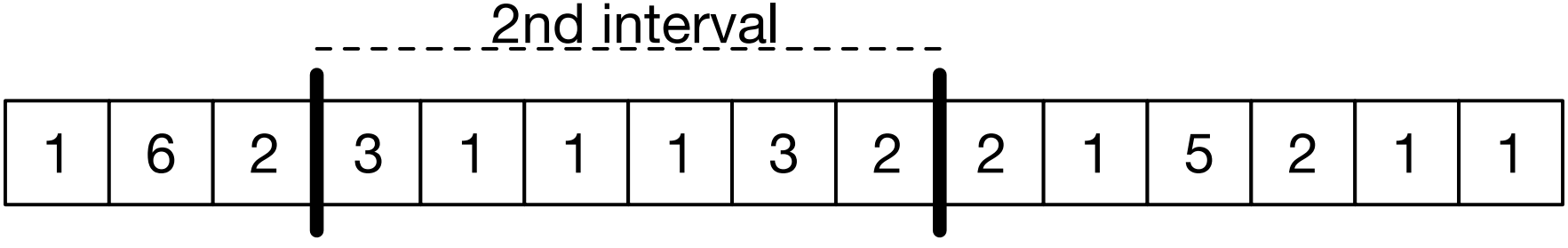
Greedy Justification

This does not make a mistake. There is an optimum with the leftmost interval being as long as possible (otherwise, we can stretch it longer and make the next interval shorter to get a different optimum).

Balanced Intervals

While not all integers are covered:

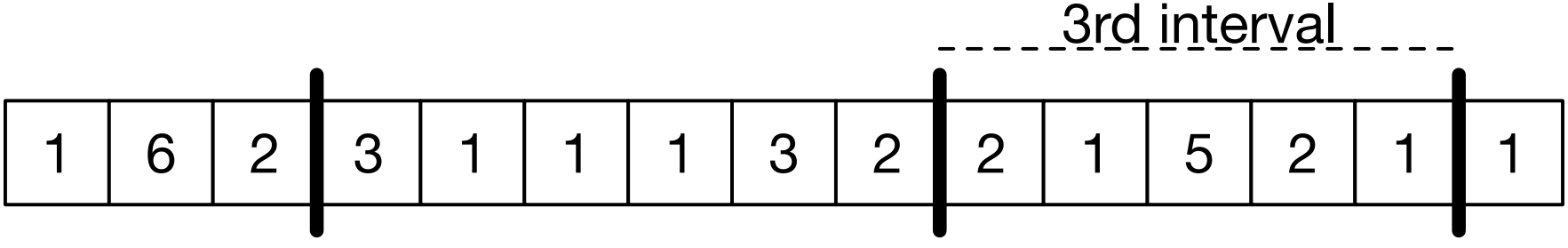
- Build the longest possible interval, starting from the first uncovered number.



Balanced Intervals

While not all integers are covered:

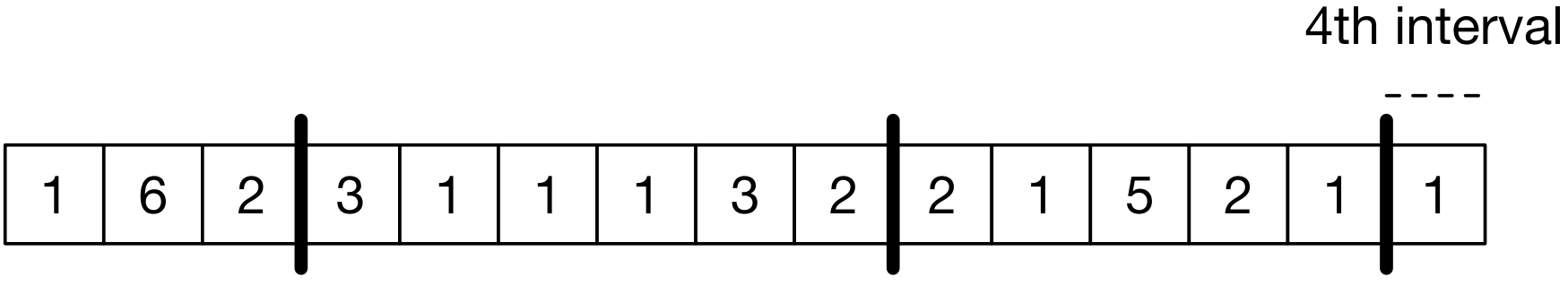
- Build the longest possible interval, starting from the first uncovered number.



Balanced Intervals

While not all integers are covered:

- Build the longest possible interval, starting from the first uncovered number.



Done with 4 intervals.

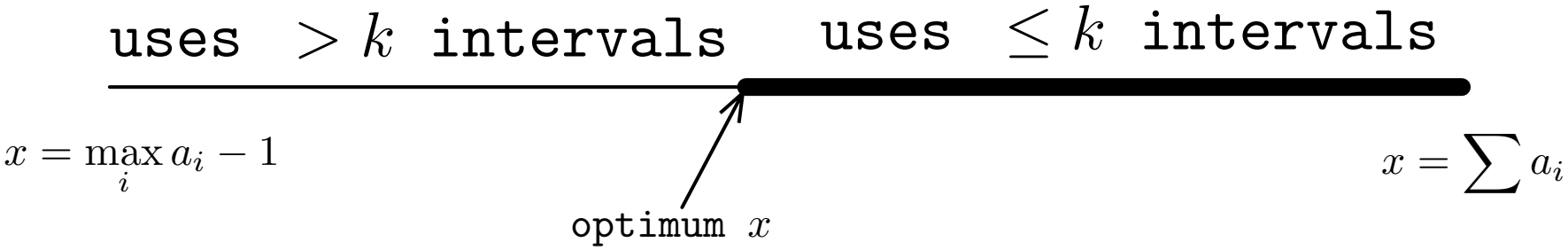
So, in $O(n)$ time we can determine the minimum number of intervals to cover all items with intervals of length $\leq x$.

Use this in a **binary search** to minimize the maximum-sum interval when dividing the integers $a_1, \ldots, a_n$ into k intervals.

$low \leftarrow \max a_i - 1$
$high \leftarrow \sum_i a_i$
while $low < high$ **do**
  $x \leftarrow (low + high)/2$ # the average, with integer division
  if the "flipped" solution with load x uses $\leq k$ intervals **then**
    $high \leftarrow x$
  else
    $low \leftarrow x$
return $high$

Invariant: Load low produces $> k$ intervals (or is just an infeasible load); load hi produces $\leq k$ intervals.

Another View



This depicts the range of x values that would result in $> k$ intervals and the range that would result in $\leq k$ intervals.

So we are binary searching over this range to find the minimum x such that we can partition into $\leq k$ intervals each having sum at most x!

This general approach works for other problems if:

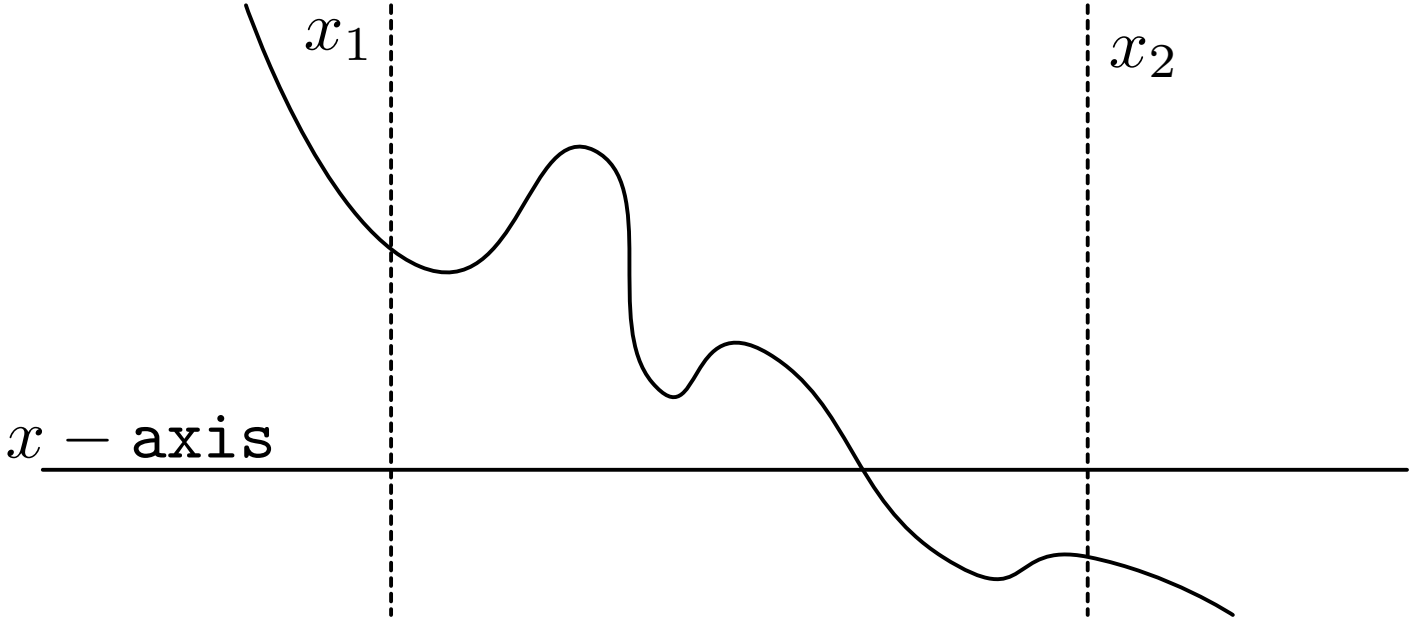
- You know a lower and upper bound on the optimum.
- You can efficiently tell if a given guess is too low or too high.

If so, then you can binary search for the optimum!

Bisection Method

The idea is useful even in "continuous" settings.

Problem: Given a continuous $f(x)$ and values $x_1 < x_2$ such that $f(x_1) \geq 0 \geq f(x_2)$, compute a value x^* such that $f(x^*) = 0$.

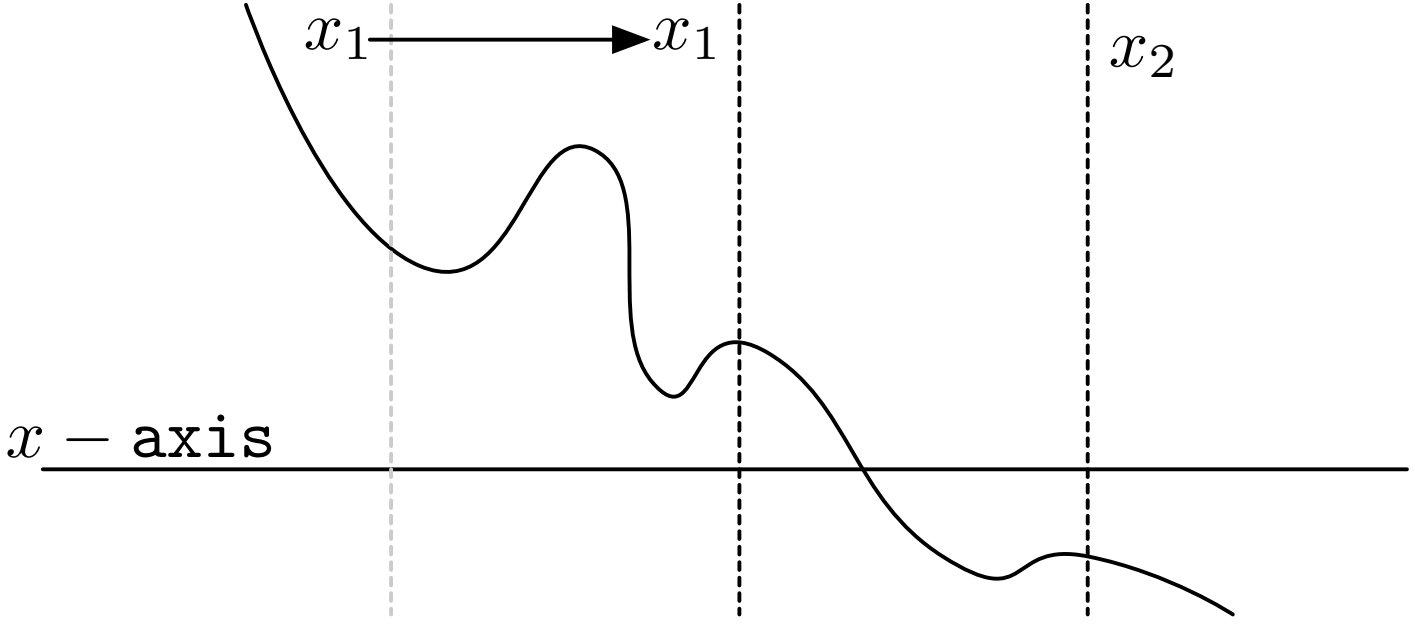


Such x^* is guaranteed to exist by the *mean value theorem*.

Bisection Method

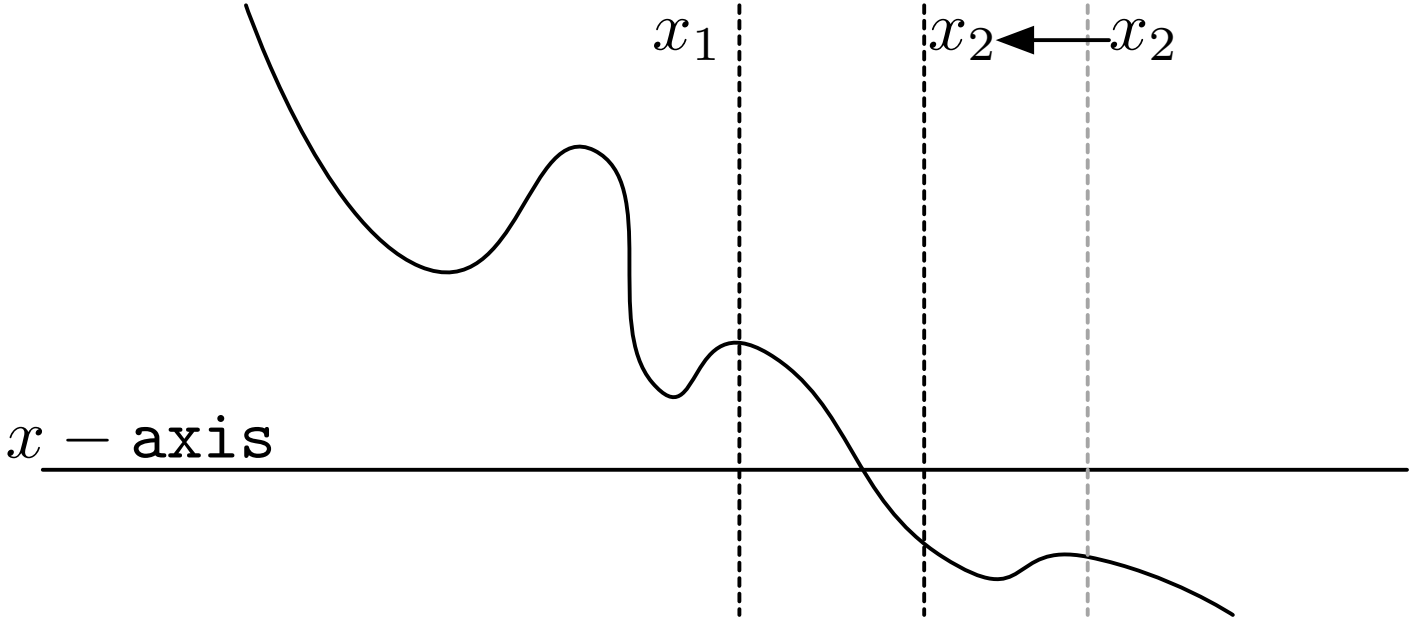
Of course, we can't exactly represent such x (what if $f(x) = x^2 - 2$, so the answer is $\sqrt{2}$) so we settle for a good approximation.

Simple approach! Plug in the midpoint and adjust either x_1 or x_2.



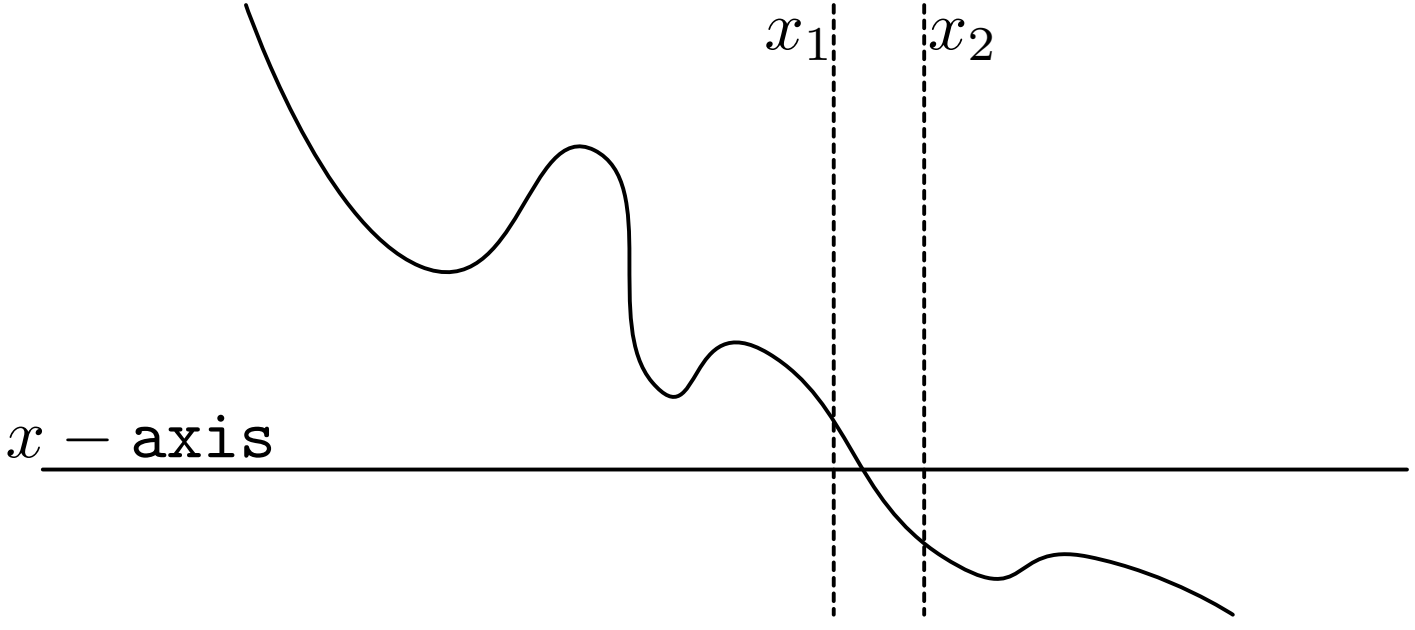
Bisection Method

Keep repeating...



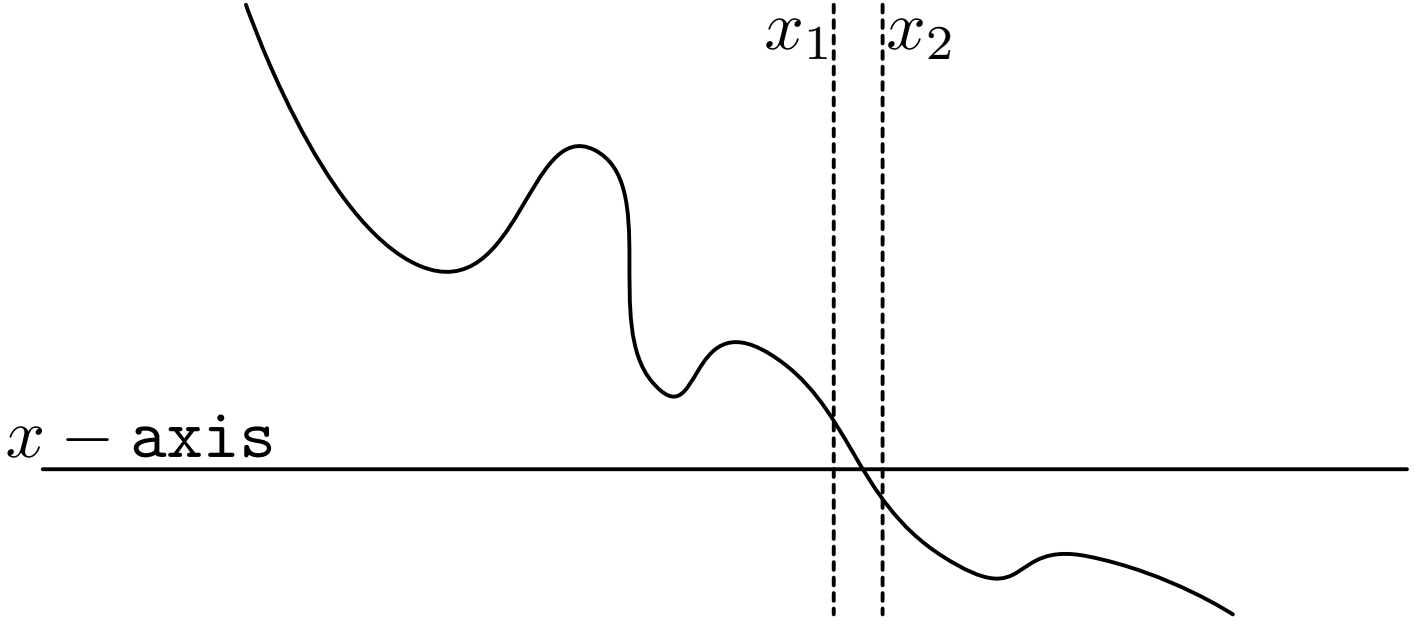
Bisection Method

Keep repeating...



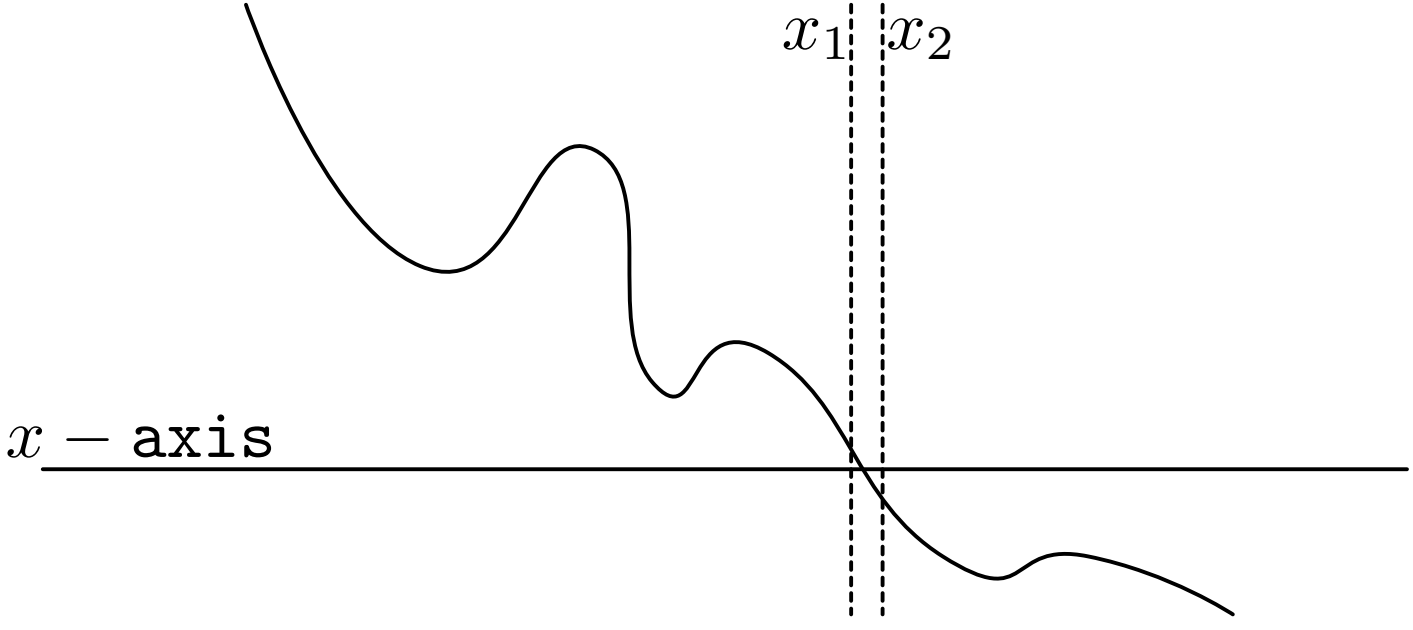
Bisection Method

Keep repeating...



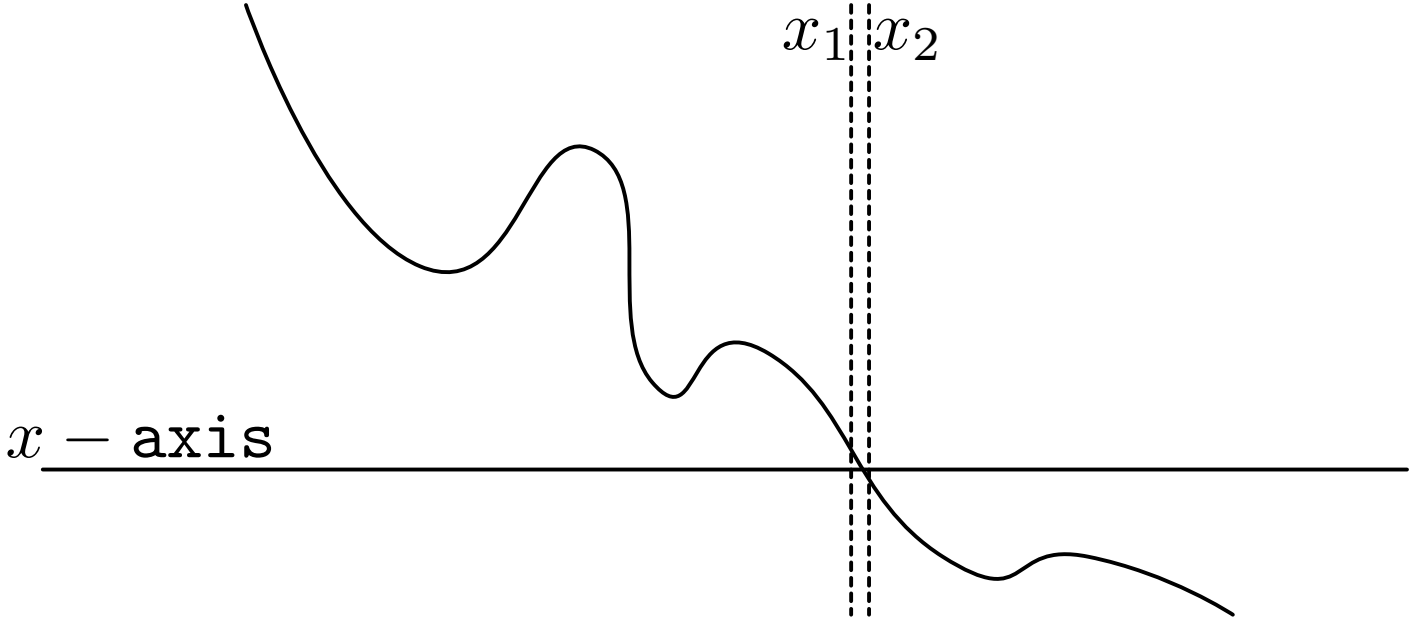
Bisection Method

Keep repeating...



Bisection Method

Stop when the range x_1 to x_2 is "small enough".



Bisection Method

For some "tolerance" $\epsilon > 0$

while $x_2 - x_1 > \epsilon$

- **if** $f\left(\frac{x_1+x_2}{2}\right) \geq 0$ **then** $x_1 \leftarrow \frac{x_1+x_2}{2}$
- **else** $x_2 \leftarrow \frac{x_1+x_2}{2}$

return x_1 (or any value between x_1 and x_2)

Smaller ϵ gives more accurate answers, but requires more iterations.

Notes

- The returned value $\overline{x}$ satisfies $|x^* - \overline{x}| \leq \epsilon$ for some x^* with $f(x^*) = 0$.
- The initial range of values has length $x_2 - x_1$ (the initial x_1, x_2).
- The final range has length $\leq \epsilon$.
- The length of the range is halved in each iteration.

Conclusion

The algorithm finds a point that is within ϵ of some root of f in $\log_2 \frac{x_2 - x_1}{\epsilon}$ iterations. A **quality-vs-running time tradeoff**.

Of course, there are numerical considerations due to rounding error. That's beyond this class.

Try it!

- Find an approximate root of $x^2 - x \cdot \log_2(x) - 10$ between 1 and 10.

- The function $f(x) = \sqrt{x} + \log_2(x+1)$ is nonnegative and increasing in the range $x \geq 0$ and satisfies $f(0) = 0$.

  So there is an "inverse" function g with $f(g(x)) = x$ for all $x \geq 0$.

  Implement a function in Python called g that, given a value $x \geq 0$, will return an approximation to $g(x)$.

  Hint: Can check if a proposed value for $g(x)$ is too low or high. What is a good initial range?

Use some small ϵ, like $\epsilon = 0.000001$.