Binary Search

Searching an Array

Linear Search

- Go through the array position by position until we find x

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
/*@ensures (\result  == -1 && !is_in(x, A, 0, n))
                 || (0 <= \result && \result < n && A[\result] == x); @*/
{
  for (int i = 0; i < n; i++) {
    if (A[i] == x)  return i;
  }
 return -1;
}
```

Loop invariants omitted

- Worst case complexity: ***O(n)***

Linear Search on *Sorted* Arrays

- Stop early if we find an element greater than x

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (\result == -1 && !is_in(x, A, 0, n))
                  || (0 <= \result && \result < n && A[\result] == x); @*/
{
  for (int i = 0; i < n; i++) {
    if (A[i] == x)  return i;
    if (x < A[i])  return -1;
    //@assert A[i] < x;
  }
 return -1;
}
```

Loop invariants omitted

- Worst case complexity: still ***O(n)***
  - e.g., if x is larger than any element in A

Can we do Better on *Sorted* Arrays?

Piece of cake!

- Look in the middle!
  - compare the midpoint element with x
  - if found, great!
  - if x is smaller, look for x in the lower half
  - if x is bigger, look for x in the upper half

- This is

Binary Search

- Why better?
  - we are throwing out half of the array each time!
    - with linear search, we were throwing out just one element!
  - if array has length *n*, we can halve it only *log n* times

A Cautionary Tale

Jon Bentley

Only 10% of programmers can write binary search

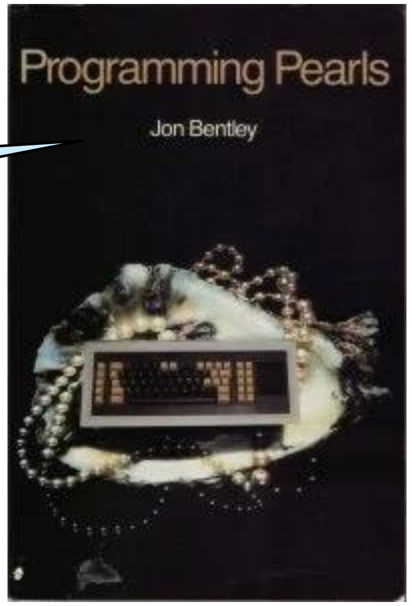
- ○ 90% had bugs!

- Binary search dates back to 1946 (at least)
  - ○ First *correct* description in 1962

- Jon Bentley wrote the **definitive** binary search
  - ➢ and proved it correct

I've assigned this problem in courses at Bell Labs and IBM. Professional programmers had a couple of hours to convert the above description into a program in the language of their choice; a high-level pseudocode was fine. At the end of the specified time, almost all the programmers reported that they had correct code for the task. We would then take thirty minutes to examine their code, which the programmers did with test cases. In several classes and with over a hundred programmers, the results varied little: ninety percent of the programmers found bugs in their programs (and I wasn't always convinced of the correctness of the code in which no bugs were found).

Jon Bentley, Algorithms professor at CMU in the 1980s



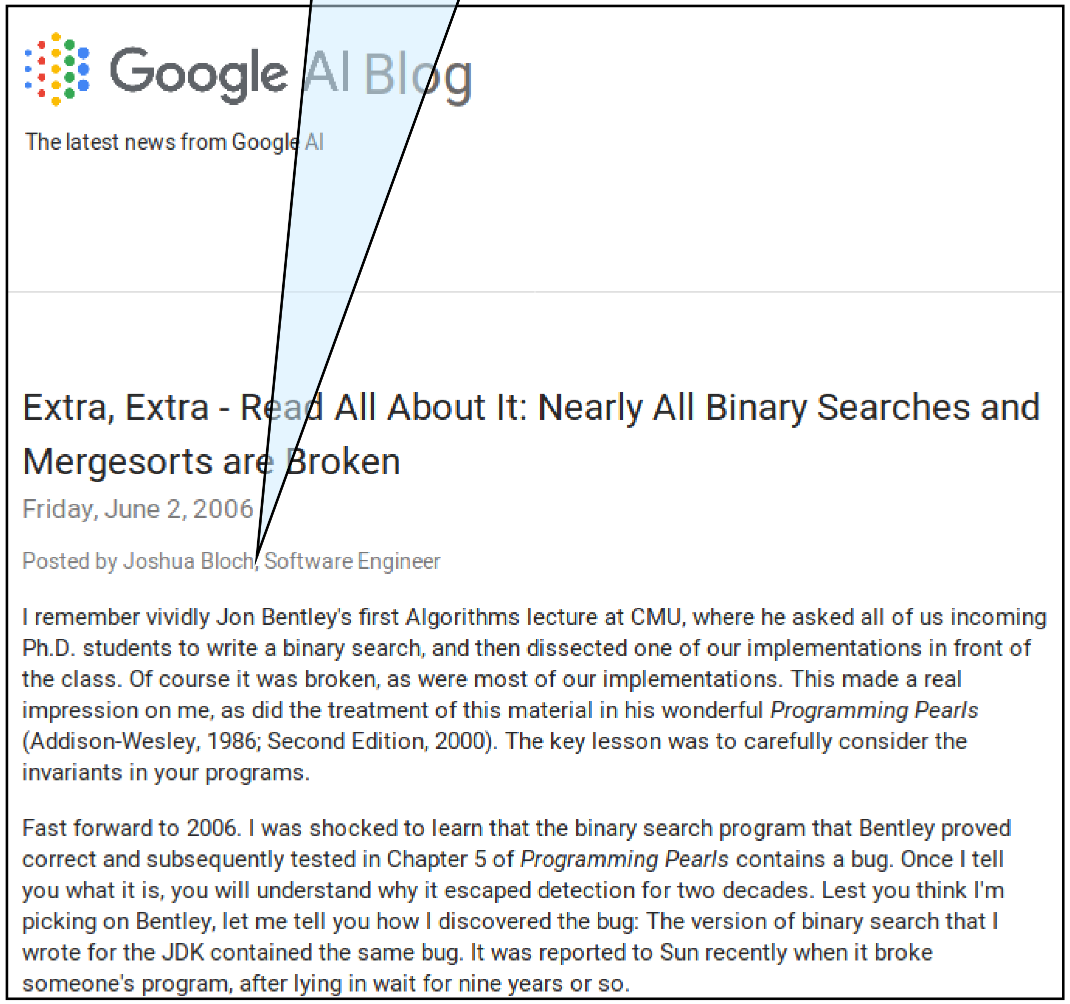
Read more at
https://reprog.wordpress.com/2010/04/19/are-you-one-of-the-10-percent/

More of a Cautionary Tale

Joshua Bloch

- Joshua Bloch finds a **bug** in Jon Bentley's definitive binary search!
  - that Bentley had proved correct!!!
- Went on to implementing several searching and sorting algorithms used in Android, Java and Python
  - e.g., TimSort

Joshua Bloch,
- student of Jon Bentley
- works at Google
- occasionally adjunct prof. at CMU



Google AI Blog

The latest news from Google AI

Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken

Friday, June 2, 2006

Posted by Joshua Bloch, Software Engineer

I remember vividly Jon Bentley's first Algorithms lecture at CMU, where he asked all of us incoming Ph.D. students to write a binary search, and then dissected one of our implementations in front of the class. Of course it was broken, as were most of our implementations. This made a real impression on me, as did the treatment of this material in his wonderful *Programming Pearls* (Addison-Wesley, 1986; Second Edition, 2000). The key lesson was to carefully consider the invariants in your programs.

Fast forward to 2006. I was shocked to learn that the binary search program that Bentley proved correct and subsequently tested in Chapter 5 of *Programming Pearls* contains a bug. Once I tell you what it is, you will understand why it escaped detection for two decades. Lest you think I'm picking on Bentley, let me tell you how I discovered the bug: The version of binary search that I wrote for the JDK contained the same bug. It was reported to Sun recently when it broke someone's program, after lying in wait for nine years or so.

Read more at
https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html

Even More of a Cautionary Tale

- Researchers find a **bug** in Joshua Bloch's code for TimSort
  - Implemented it in a language with contracts (JML – Java Modelling Language)
  - Tried to prove correctness using KeY theorem prover

Proving that Android's, Java's and Python's sorting algorithm is broken (and showing how to fix it)

February 24, 2015 Envisage Written by Stijn de Gouw. $s

```
/*@ private normal_behavior
  @ requires
  @   n >= MIN_MERGE;
  @ ensures
  @   \result >= MIN_MERGE/2;
  @*/

private static int /*@ pure @*/ minRunLength(int n) {
  assert n >= 0;
  int r = 0;      // Becomes 1 if any 1 bits are shifted off
  /*@ loop_invariant n >= MIN_MERGE/2 && r >=0 && r<=1;
    @ decreases n;
    @ assignable \nothing;
    @*/
  while (n >= MIN_MERGE) {
    r |= (n & 1);
    n >>= 1;
  }
  return n + r;
}
```

Some of the same contract mechanisms as C0 (and a few more)

(we borrowed our contracts of them)

Read more at
http://www.envisage-project.eu/proving-android-java-and-python-sorting-algorithm-is-broken-and-how-to-fix-it/

Piece of cake?

- Implementing binary search is not as simple as it sounds
  - many professionals have failed!
- We want to proceed carefully and methodically
- Contracts will be our guide!

Binary Search

Binary Search

- A is sorted
- Looking for x = **4**

find midpoint of A[0,7)
- index 3
- A[3] = 9

4 < 9
- ignore A[4,7)
- ignore also A[3]

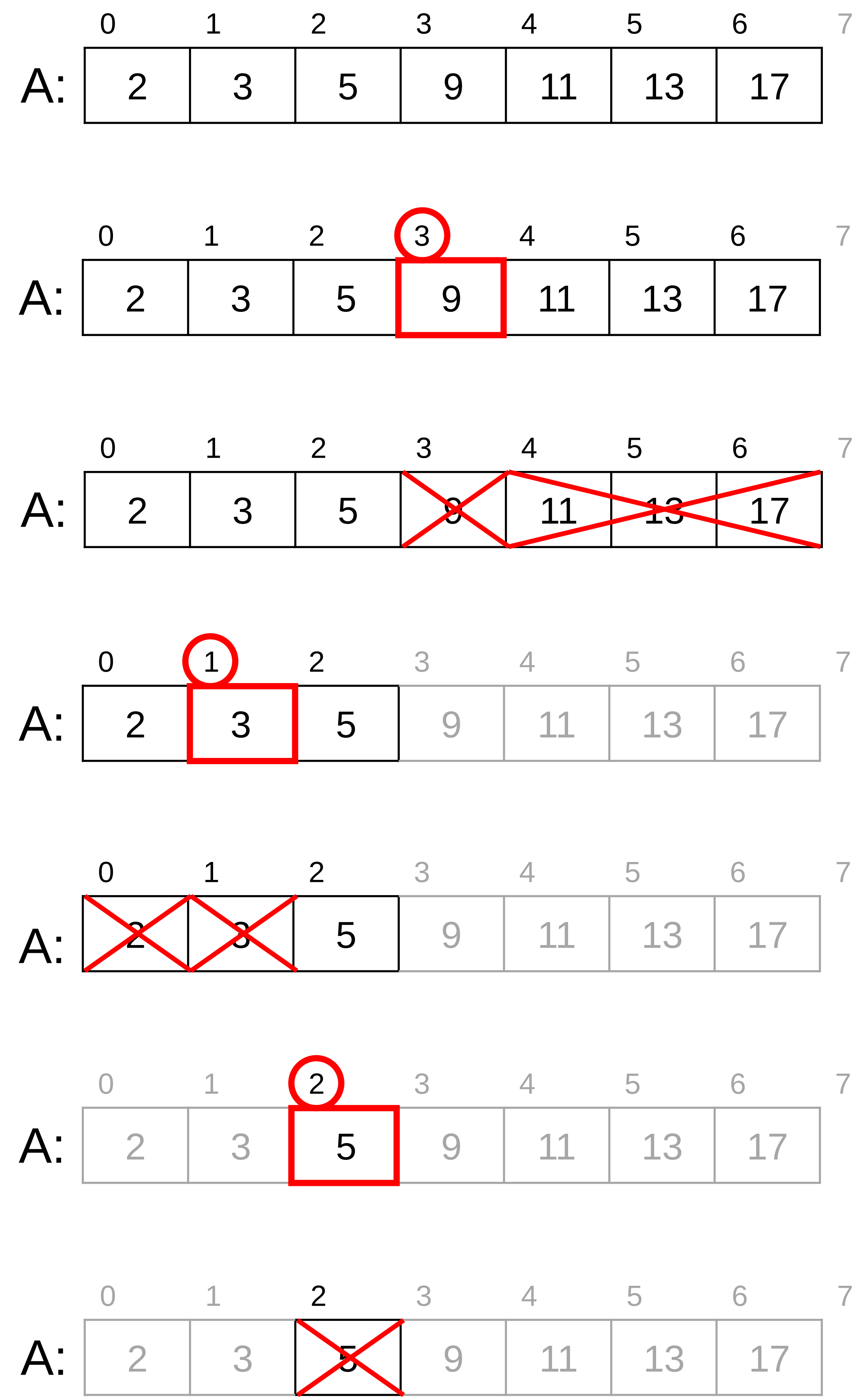
find midpoint of A[0,3)
- index 1
- A[1] = 3

3 < **4**
- ignore A[0,1)
- ignore also A[1]

find midpoint of A[2,3)
- index 2
- A[2] = 5

4 < 5
- ignore A[3,3)
- ignore also A[2]

nothing left!
- A[2,2) is empty
- **4** isn't in A



Binary Search

- A is sorted
- At each step, we
  - examine a segment A[lo, hi)
  - find its midpoint mid
  - compare x with A[mid]

find midpoint of A[lo,hi)
- index mid = 3
- A[mid] = 9

x < A[mid]
- ignore A[mid+1,hi)
- ignore also A[mid]

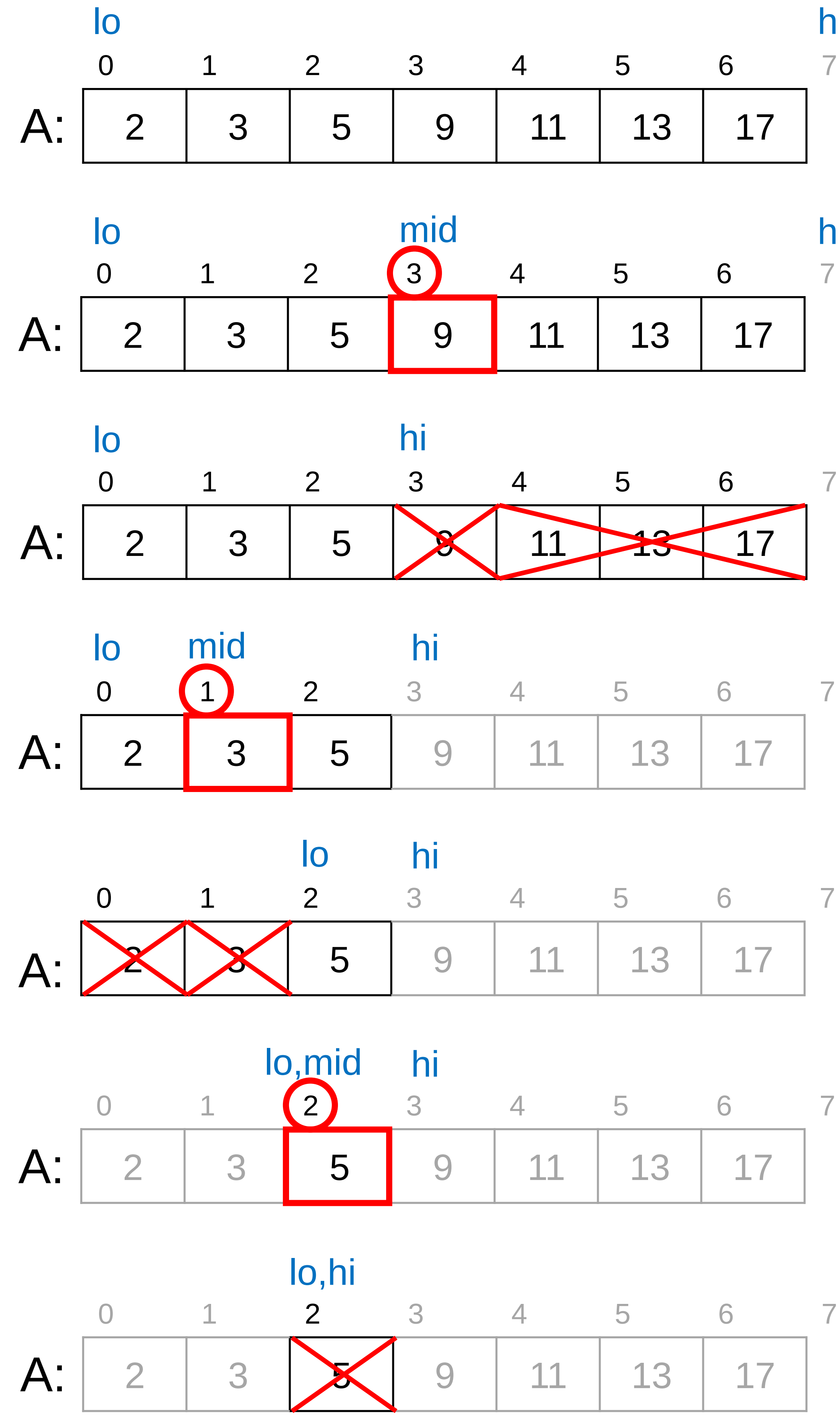
find midpoint of A[lo,hi)
- index mid = 1
- A[mid] = 3

A[mid] < x
- ignore A[lo,mid)
- ignore also A[mid]

find midpoint of A[lo,hi)
- index mid = 2
- A[mid] = 5

x < A[mid]
- ignore A[mid+1,hi)
- ignore also A[mid]

nothing left!
- A[lo,hi) is empty
- x isn't in A



Binary Search

- Let's look for
  x = **11**
- At each step, we
  - examine a segment A[lo, hi)
  - find its midpoint mid
  - compare x with A[mid]

find midpoint of A[lo,hi)
- index mid = 3
- A[mid] = 9

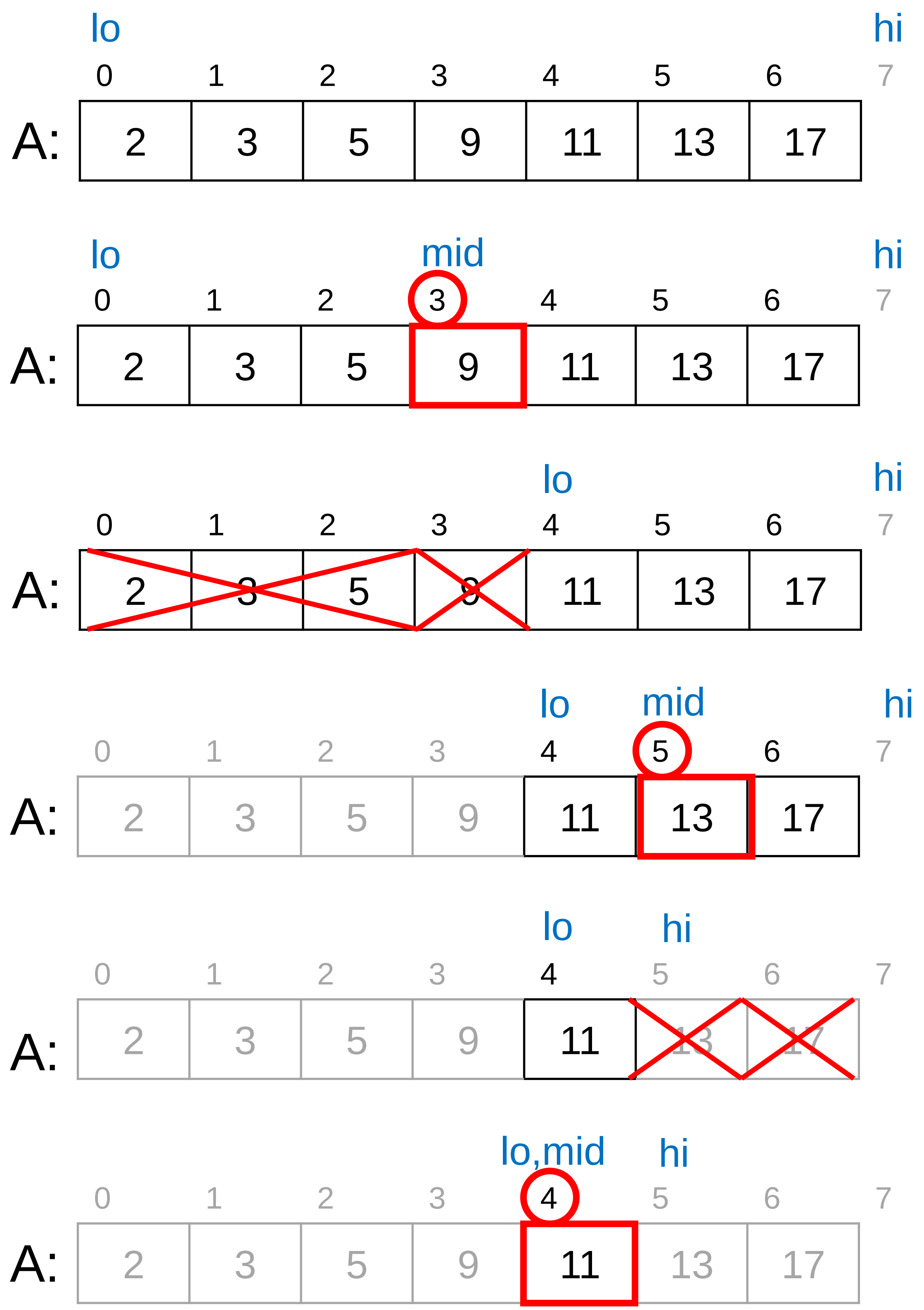
A[mid] < x
- ignore A[lo,mid)
- ignore also A[mid]

find midpoint of A[lo,hi)
- index mid = 5
- A[mid] = 13

x < A[mid]
- ignore A[mid+1,hi)
- ignore also A[mid]

find midpoint of A[lo,hi)
- index mid = 4
- A[mid] = 11

x = A[mid]
- found!
- return 4



Implementing Binary Search

Setting up Binary Search

```
int binsearch(int x, int[] A, int n)
//@requires n == \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (\result  == -1 && !is_in(x, A, 0, n))
           || (0 <= \result && \result < n && A[\result] == x); @*/
{
  int lo = 0;
  int hi = n;
  while (lo < hi)
  {
   …
  }
 return -1;
}
```

Same contracts as linear search: different algorithm to solve the **same problem**

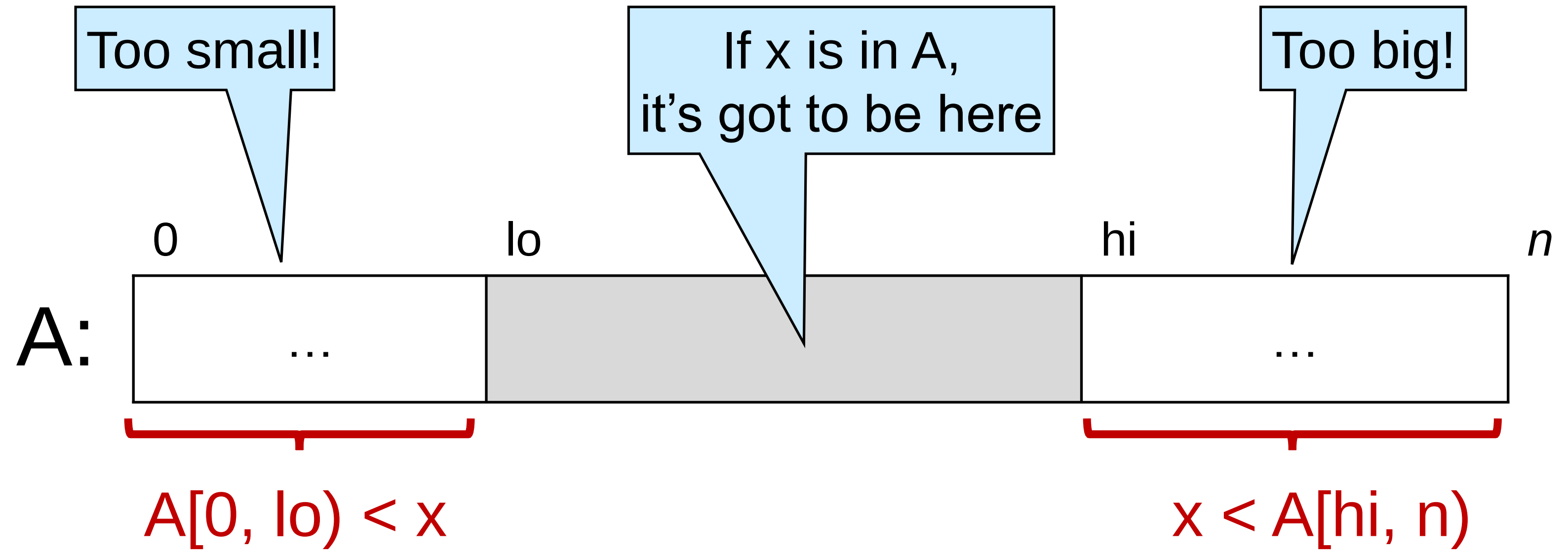
lo starts at 0, hi at n

bunch of steps

returns -1 if x not found

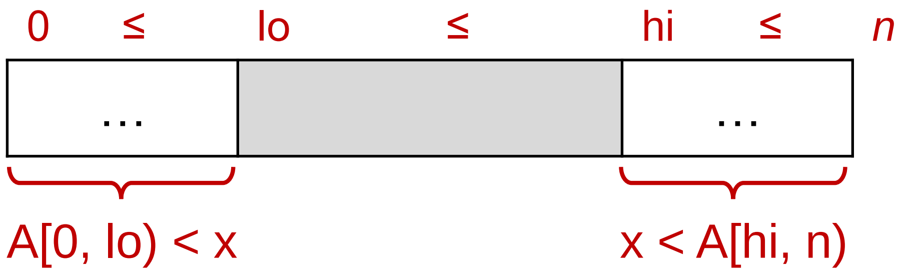
What do we Know at Each Step?

- At an arbitrary iteration, the picture is:



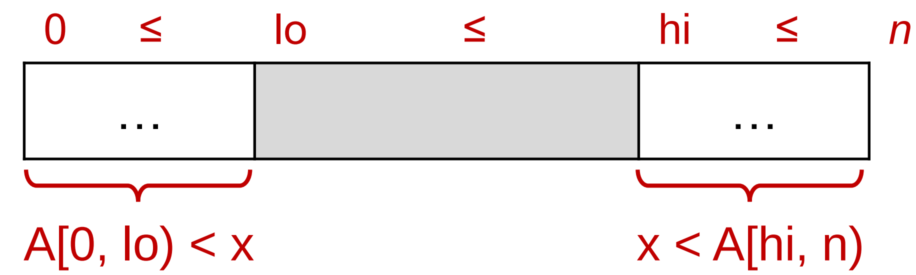
- These are *candidate* loop invariant:
  - gt_seg(x, A, 0, lo): that's A[0, lo) < x
  - lt_seg(x, A, hi, n): that's x < A[hi, n)
  - and of course 0 <= lo && lo <= hi && hi <= n

Adding Loop Invariants




```
int binsearch(int x, int[] A, int n)
//@requires n == \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (\result  == -1 && !is_in(x, A, 0, n))
                  || (0 <= \result && \result < n && A[\result] == x); @*/
{
  int lo = 0;
  int hi = n;
  while (lo < hi)
  //@loop_invariant 0 <= lo && lo <= hi && hi <= n;
  //@loop_invariant gt_seg(x, A, 0, lo);
  //@loop_invariant lt_seg(x, A, hi, n);
  {
   …
  }
 return -1;
}
```

Are these *Useful* Loop Invariants?



Can they help prove the postcondition?

- Is return -1 correct?

  (assuming invariants are valid)

  - **To show**: if preconditions are met, then $x \notin A[0, n)$

A. $lo \geq hi$ by line 9 (negation of loop guard)

B. $lo \leq hi$ by line 10 (LI 1)

C. $lo = hi$ by math on A, B

D. $A[0,lo) < x$ by line 11 (LI 2)

E. $x \notin A[0,lo)$ by math on D

F. $x < A[hi,n)$ by line 12 (LI 3)

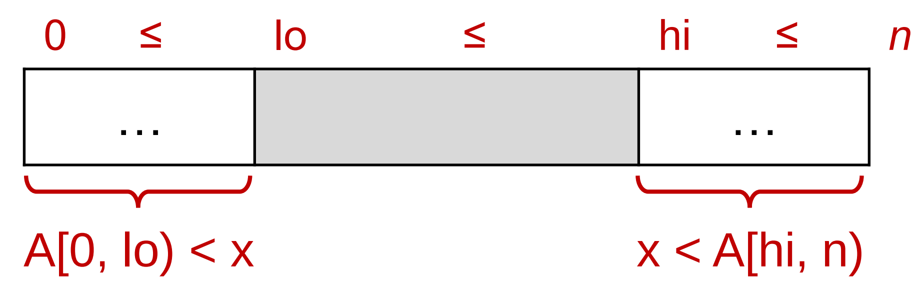
G. $x \notin A[hi,n)$ by math on F

H. $x \notin A[0,n)$ by math on C, E, G

- This is a standard EXIT argument

```
1.  int binsearch(int x, int[] A, int n)
2.  //@requires n == \length(A);
3.  //@requires is_sorted(A, 0, n);
4.  /*@ensures (\result  == -1 && !is_in(x, A, 0, n))
5.                || (0 <= \result && \result < n && A[\result] == x); @*/
6.  {
7.    int lo = 0;
8.    int hi = n;
9.    while (lo < hi)
10.   //@loop_invariant 0 <= lo && lo <= hi && hi <= n;
11.   //@loop_invariant gt_seg(x, A, 0, lo);
12.   //@loop_invariant lt_seg(x, A, hi, n);
13.   {
        …
      }
    return -1;
  }
```

Are the Loop Invariants Valid?



INIT

- lo = 0 by line 7 and hi = n by line 8
  - **To show**: 0 ≤ 0 by math
  - **To show**: 0 ≤ n by line 2 (preconditions) and \length
  - **To show**: n ≤ n by math
  - **To show**: A[0, 0) < x
  - **To show**: x < A[n, n)
    - by math (empty intervals)

PRES

- Trivial
  - body is empty
  - nothing changes!!!

```
1.  int binsearch(int x, int[] A, int n)
2.  //@requires n == \length(A);
3.  //@requires is_sorted(A, 0, n);
4.  /*@ensures (\result  == -1 && !is_in(x, A, 0, n))
5.                  || (0 <= \result && \result < n && A[\result] == x); @*/
6.  {
7.    int lo = 0;
8.    int hi = n;
9.    while (lo < hi)
10.   //@loop_invariant 0 <= lo && lo <= hi && hi <= n;
11.   //@loop_invariant gt_seg(x, A, 0, lo);
12.   //@loop_invariant lt_seg(x, A, hi, n);
13.   {
        …
      }
     //@assert lo == hi;
     return -1;
    }
```

from correctness proof

Is binsearch Correct?

- EXIT
- INIT
- PRES
- Termination
  - Infinite loop!

- Let's implement what happens in a binary search step
  - compute the midpoint
  - compare its value to x

```
int binsearch(int x, int[] A, int n)
//@requires n == \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (\result  == -1 && !is_in(x, A, 0, n))
                  || (0 <= \result && \result < n && A[\result] == x); @*/
{
  int lo = 0;
  int hi = n;
  while (lo < hi)
  //@loop_invariant 0 <= lo && lo <= hi && hi <= n;
  //@loop_invariant gt_seg(x, A, 0, lo);
  //@loop_invariant lt_seg(x, A, hi, n);
  {
    …
  }
 //@assert lo == hi;
 return -1;
}
```

Adding the Body

```
int binsearch(int x, int[] A, int n)
//@requires n == \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (\result  == -1 && !is_in(x, A, 0, n))
                  || (0 <= \result && \result < n && A[\result] == x); @*/
{
  int lo = 0;
  int hi = n;
  while (lo < hi)
  //@loop_invariant 0 <= lo && lo <= hi && hi <= n;
  //@loop_invariant gt_seg(x, A, 0, lo);
  //@loop_invariant lt_seg(x, A, hi, n);
  {
    int mid = (lo + hi) / 2;

    if (A[mid] == x)  return mid;
    if (A[mid] < x) {
     lo = mid + 1;
    } else {   //@assert A[mid] > x;
      hi = mid;
    }
 }
  //@assert lo == hi;
 return -1;
}
```

by high-school math

if A[mid] not == x and not < x, then A[mid] > x

Is it Safe?

- A[mid] must be in bounds
  - 0 ≤ mid < \length(A)

- We expect lo ≤ mid < hi
  - **not** mid ≤ hi
    - otherwise we could have mid == \length(A) by lines 2, 9
- *Candidate* assertion: lo <= mid && mid < hi
  - We will check it later

```
1.  int binsearch(int x, int[] A, int n)
2.  //@requires n == \length(A);
3.  //@requires is_sorted(A, 0, n);
4.  /*@ensures … @*/
5.  {
6.    int lo = 0;
7.    int hi = n;
8.    while (lo < hi)
9.    //@loop_invariant 0 <= lo && lo <= hi && hi <= n;
10.   //@loop_invariant gt_seg(x, A, 0, lo);
11.   //@loop_invariant lt_seg(x, A, hi, n);
12.   {
13.     int mid = (lo + hi) / 2;
14.
15.     if (A[mid] == x)  return mid;
16.     if (A[mid] < x) {
17.      lo = mid + 1;
18.     } else {   //@assert A[mid] > x;
19.      hi = mid;
20.     }
21.  }
22.   //@assert lo == hi;
23.  return -1;
24. }
```

Are the LI Valid?

INIT: unchanged ✓

PRES

- **To show**: if 0 ≤ lo ≤ hi ≤ n,
  then 0 ≤ lo’ ≤ hi’ ≤ n

- if A[mid] == x, *nothing to prove*
- if A[mid] < x
  - A. lo’ = mid+1 — by line 17
  - B. hi’ = hi — (unchanged)
  - C. 0 ≤ lo — by line 9 (LI1)
  - D. lo ≤ mid — by line 14 (*to be checked*)
  - E. mid < hi — by line 14 (*to be checked*)
  - F. mid < mid+1 — by math on E (no overflow)
  - G. 0 ≤ lo’ — by A, C, D, F
  - H. lo’ ≤ hi’ — by math on A, B, E
  - I. hi’ ≤ n — by B and assumption
- If A[mid] > x — *Left as exercise*

```
1.  int binsearch(int x, int[] A, int n)
2.  //@requires n == \length(A);
3.  //@requires is_sorted(A, 0, n);
4.  /*@ensures … @*/
5.  {
6.    int lo = 0;
7.    int hi = n;
8.    while (lo < hi)
9.    //@loop_invariant 0 <= lo && lo <= hi && hi <= n;
10.   //@loop_invariant gt_seg(x, A, 0, lo);
11.   //@loop_invariant lt_seg(x, A, hi, n);
12.   {
13.     int mid = (lo + hi) / 2;
14.     //@assert lo <= mid && mid < hi;  // Added
15.     if (A[mid] == x)  return mid;
16.     if (A[mid] < x) {
17.       lo = mid + 1;
18.     } else {   //@assert A[mid] > x;
19.       hi = mid;
20.     }
21.  }
22.  //@assert lo == hi;
23.  return -1;
24. }
```

Are the LI Valid?

PRES *(continued)*

- **To show**: if A[0, lo) < x,
  then A[0, lo’) < x

  - if A[mid] == x, *nothing to prove*
  - if A[mid] < x
    A. lo’ = mid+1 by line 17
    B. A[0,n) sorted by line 3
    C. A[0,mid) ≤ A[mid] by B
    D. A[0, mid+1) < x by math on C and line 16
  - If A[mid] > x
    A. lo’ = lo (unchanged)
    B. A[0,lo) < x by assumption

- **To show**: if x < A[hi, n), then x < A[hi’, n)

Left as exercise

✓

```
1. int binsearch(int x, int[] A, int n)
2. //@requires n == \length(A);
3. //@requires is_sorted(A, 0, n);
4. /*@ensures … @*/
5. {
6.   int lo = 0;
7.   int hi = n;
8.   while (lo < hi)
9.   //@loop_invariant 0 <= lo && lo <= hi && hi <= n;
10.  //@loop_invariant gt_seg(x, A, 0, lo);
11.  //@loop_invariant lt_seg(x, A, hi, n);
12.  {
13.    int mid = (lo + hi) / 2;
14.    //@assert lo <= mid && mid < hi;
15.    if (A[mid] == x)  return mid;
16.    if (A[mid] < x) {
17.     lo = mid + 1;
18.    } else {   //@assert A[mid] > x;
19.      hi = mid;
20.    }
21. }
22.  //@assert lo == hi;
23. return -1;
24. }
```

Does it Terminate?

The quantity **hi-lo decreases** in an arbitrary iteration of the loop and never **gets smaller** than 0

- This is the usual operational argument
- We can also give a **point-to** argument
  - ➢ **To show**: if $0 < hi - lo$,
    then $0 \leq hi' - lo' < hi - lo$
  - ○ if A[mid] == x, *nothing to prove*
  - ○ if A[mid] < x
    - A. $hi' - lo' = hi - (mid+1)$ by line 17 (and hi unchanged)
    - B. $< hi - mid$ by math
    - C. $\leq hi - lo$ by line 14 *(to be checked)*
    - D. $hi' - lo' = hi - (mid+1) \geq (mid+1) - (mid+1) = 0$ by lines 17, 16, 14 and math
  - ○ If A[mid] > x

Left as exercise

```
1.  int binsearch(int x, int[] A, int n)
2.  //@requires n == \length(A);
3.  //@requires is_sorted(A, 0, n);
4.  /*@ensures … @*/
5.  {
6.    int lo = 0;
7.    int hi = n;
8.    while (lo < hi)
9.    //@loop_invariant 0 <= lo && lo <= hi && hi <= n;
10.   //@loop_invariant gt_seg(x, A, 0, lo);
11.   //@loop_invariant lt_seg(x, A, hi, n);
12.   {
13.     int mid = (lo + hi) / 2;
14.     //@assert lo <= mid && mid < hi;
15.     if (A[mid] == x)  return mid;
16.     if (A[mid] < x) {
17.      lo = mid + 1;
18.     } else {   //@assert A[mid] > x;
19.       hi = mid;
20.     }
21.  }
22.   //@assert lo == hi;
23.  return -1;
24. }
```

The Midpoint Assertion

```
…
int mid = (lo + hi) / 2;
//@assert lo <= mid && mid < hi;
…
```

by high-school math

- We need to show that lo <= mid && mid < hi
- … but is it true?
  - We expect
    mid == int_max() - 1 == 2147483646
  - but we get **mid == -2** !!!!

  lo + hi overflows!

Counterexample

Linux Terminal

```
# coin -l util
--> int lo = int_max() - 2;
lo is 2147483645 (int)
--> int hi = int_max();
hi is 2147483647 (int)
--> int mid = (lo + hi) / 2;
mid is -2 (int)
```

- This is Jon Bentley's bug!
  - Google was the first company to need arrays that big
    - and Joshua Bloch worked there

The Midpoint Assertion

- Can we compute the midpoint without overflow?

```
…
int mid = lo + (hi - lo) / 2;
//@assert lo <= mid && mid < hi;
…
```

Joshua Bloch's fix

  - Does it work? *Left as exercise*
    - show that (lo + hi) / 2 is **mathematically equal** to lo + (hi - lo) / 2
    - show that lo + (hi - lo) / 2 **never overflows** for lo ≤ hi

- What about int mid = lo / 2 + hi / 2; ?
  - never overflows,
  - but not mathematically equal to (lo + hi) / 2

Left as exercise

Final Code for binsearch

- Safe
- Correct

```
int binsearch(int x, int[] A, int n)
//@requires n == \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (\result  == -1 && !is_in(x, A, 0, n))
                  || (0 <= \result && \result < n && A[\result] == x); @*/
{
  int lo = 0;
  int hi = n;
  while (lo < hi)
  //@loop_invariant 0 <= lo && lo <= hi && hi <= n;
  //@loop_invariant gt_seg(x, A, 0, lo);
  //@loop_invariant lt_seg(x, A, hi, n);
  {
    int mid = lo + (hi - lo) / 2;
    //@assert lo <= mid && mid < hi;
    if (A[mid] == x)  return mid;
    if (A[mid] < x) {
     lo = mid + 1;
    } else {   //@assert A[mid] > x;
      hi = mid;
    }
 }
  //@assert lo == hi;
 return -1;
}
```

Complexity of Binary Search

- Given an array of size *n*,
  - we **halve** the segment considered at each iteration
  - we can do this at most *log n* times before hitting the empty array

- Each iteration has constant cost

- Complexity of binary search is

O(log n)

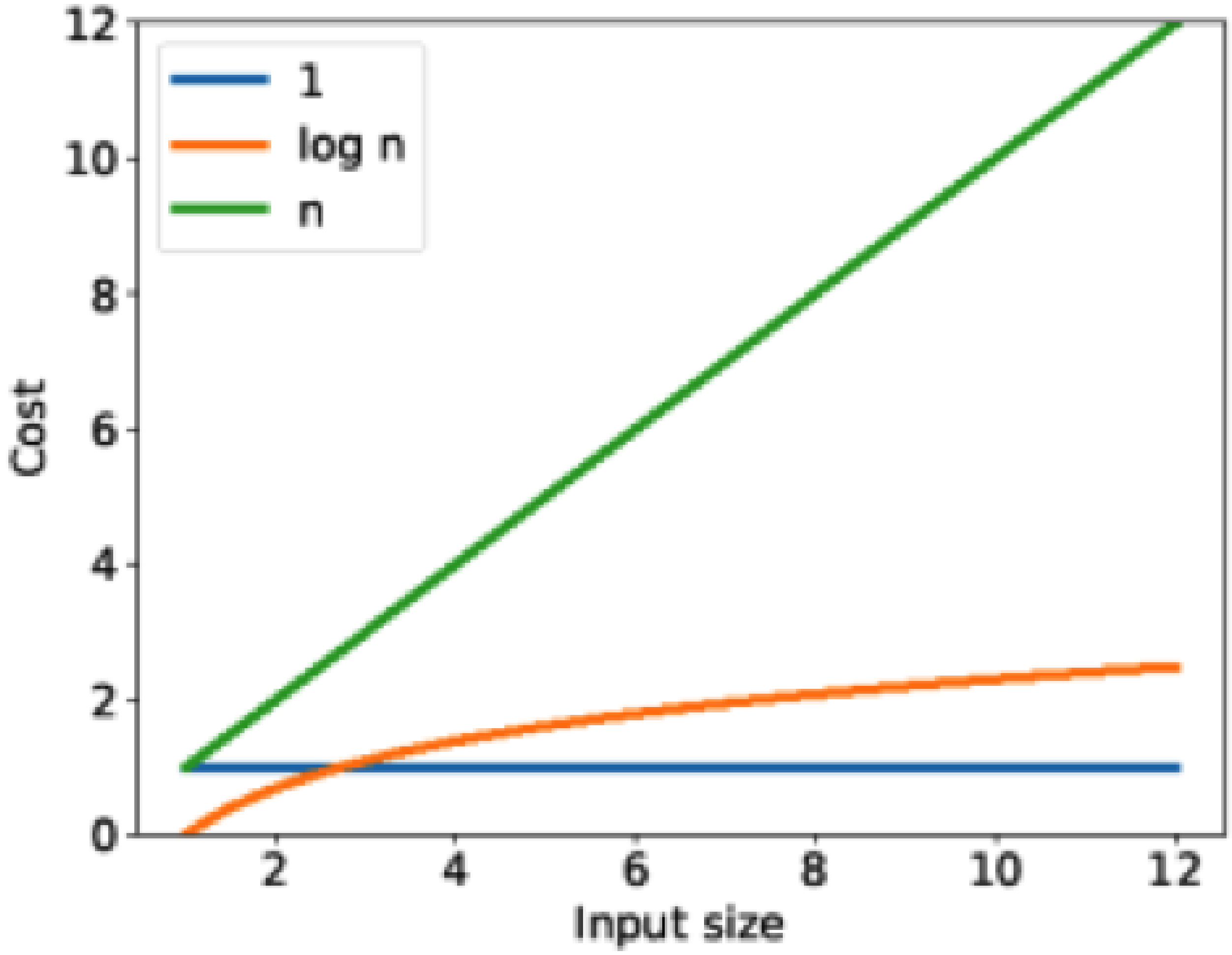
```
int binsearch(int x, int[] A, int n)
//@requires n == \length(A);
{
  int lo = 0;
  int hi = n;
  while (lo < hi) {
    int mid = lo + (hi - lo) / 2;

    if (A[mid] == x)  return mid;
    if (A[mid] < x) {
     lo = mid + 1;
    } else {
      hi = mid;
    }
  }
return -1;
}
```

Contracts omitted

The Logarithmic Advantage

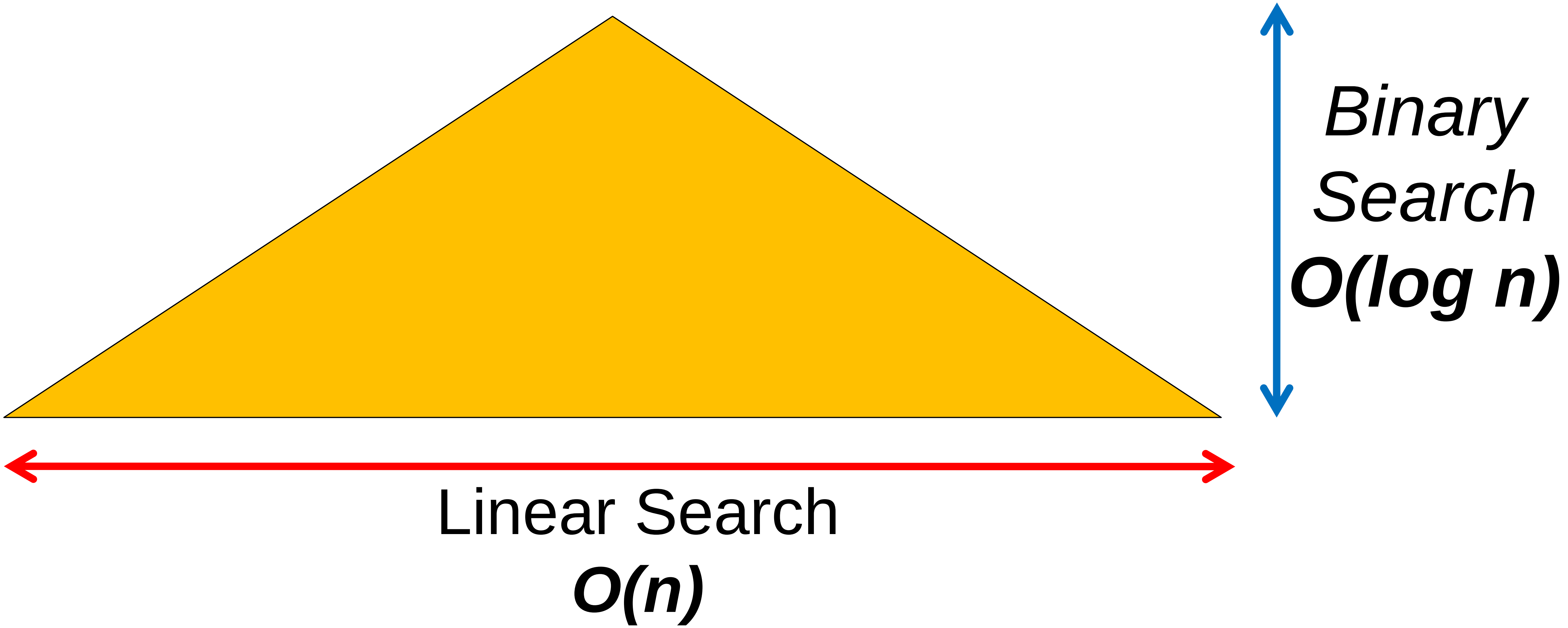
Is *O(log n)* a Big Deal?



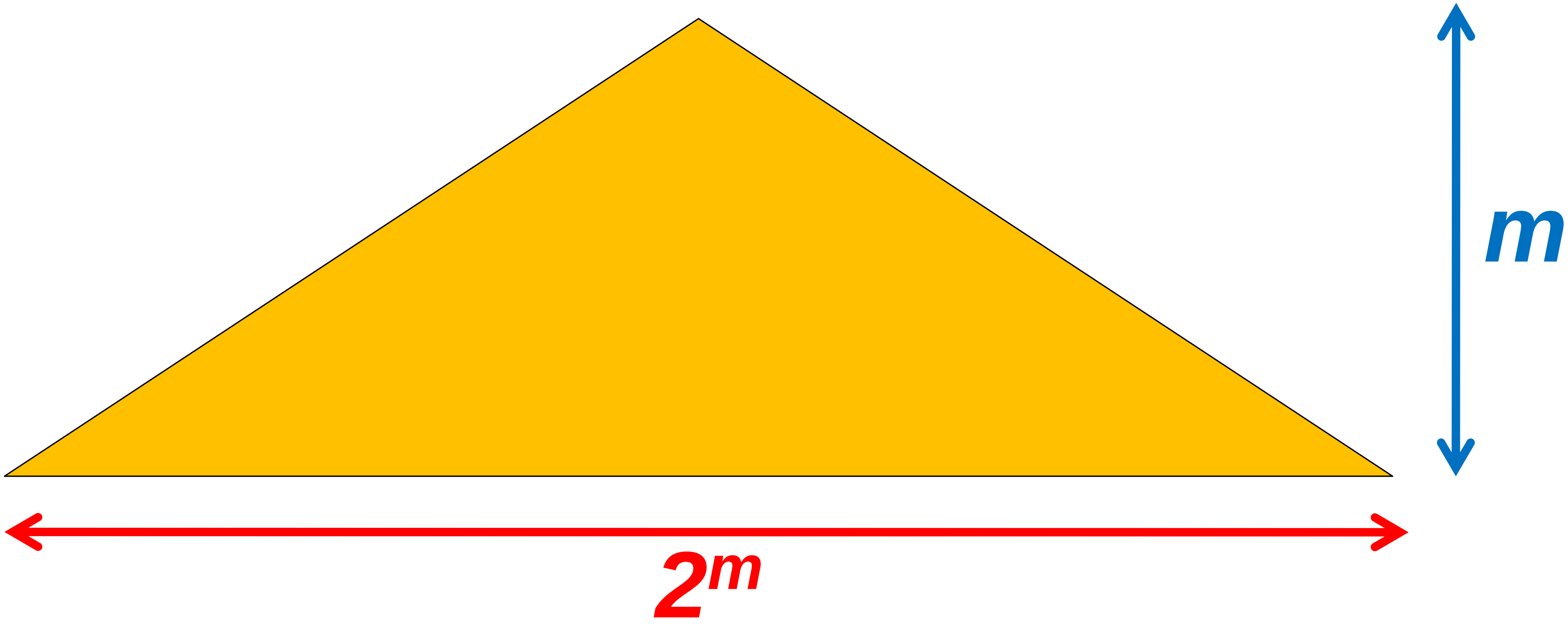
Just some boring functions
we learned in math classes?

- What does *log n* mean in practice?

Visualizing Linear and Binary Search

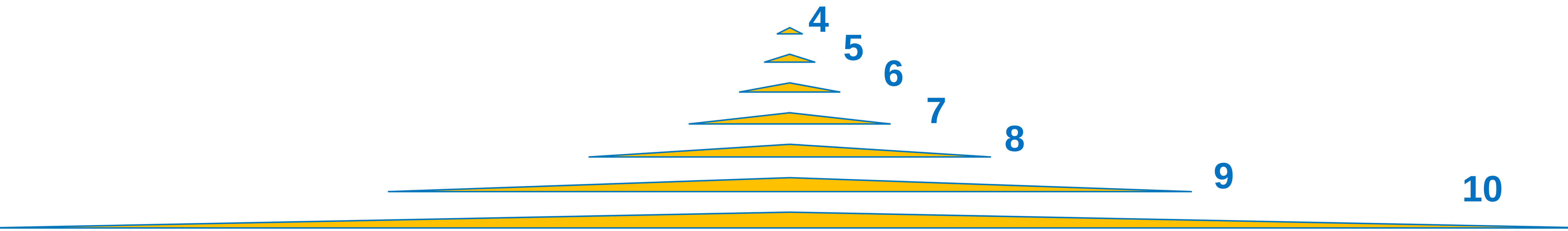


Visualizing Linear and Binary Search



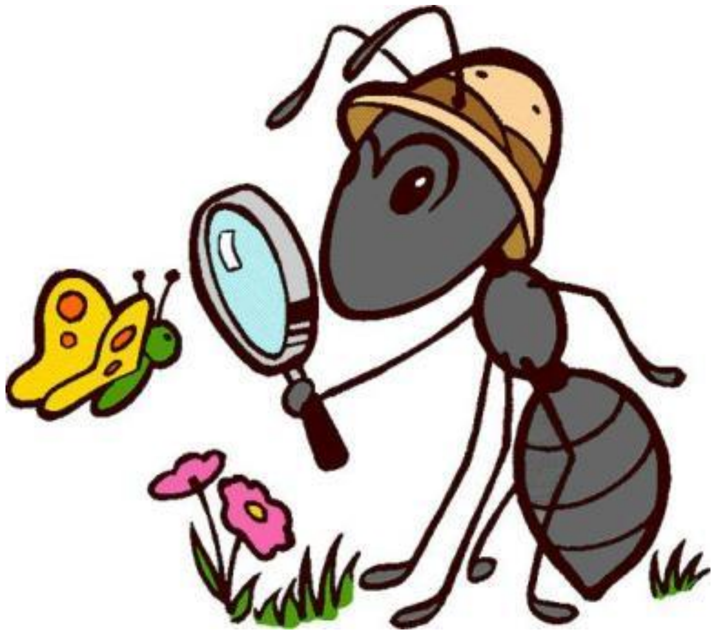
m = log n

Drawing for small values of *m*

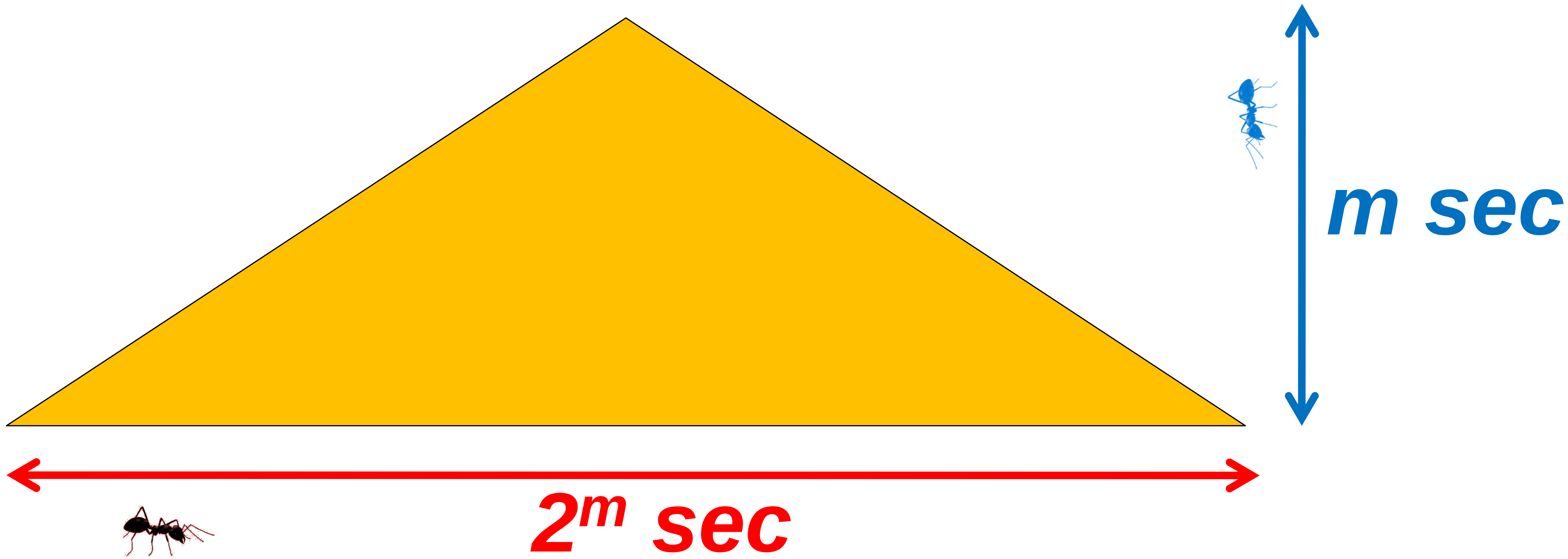


- *What do you notice?*

Searching with Ants



- Place items 1 cm apart
  - Horizontally 
  - Vertically 
- Ant walks 1cm/s



Searching 1000 items with Ants

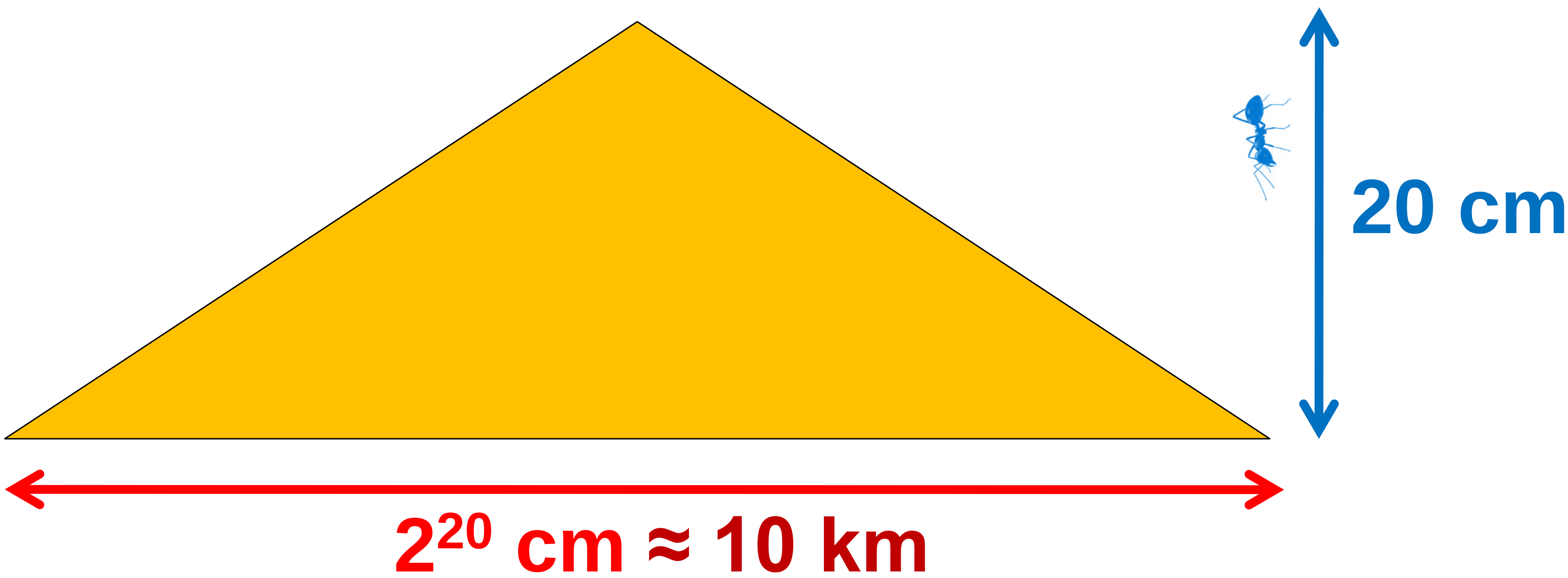
2^{10} cm ≈ 10 m

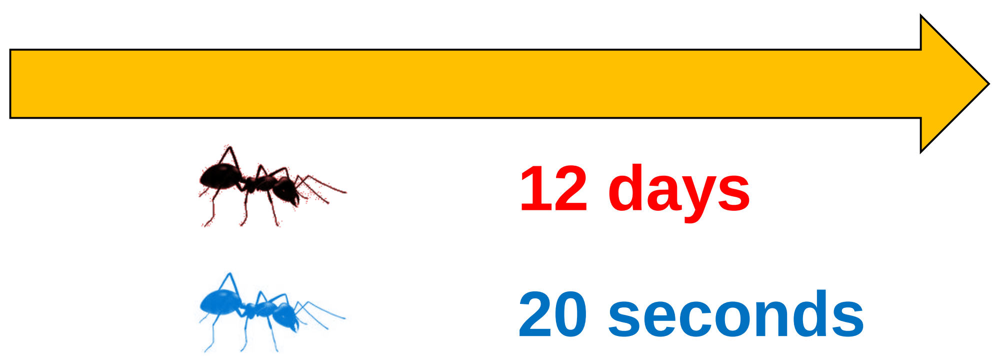
17 minutes

10 seconds

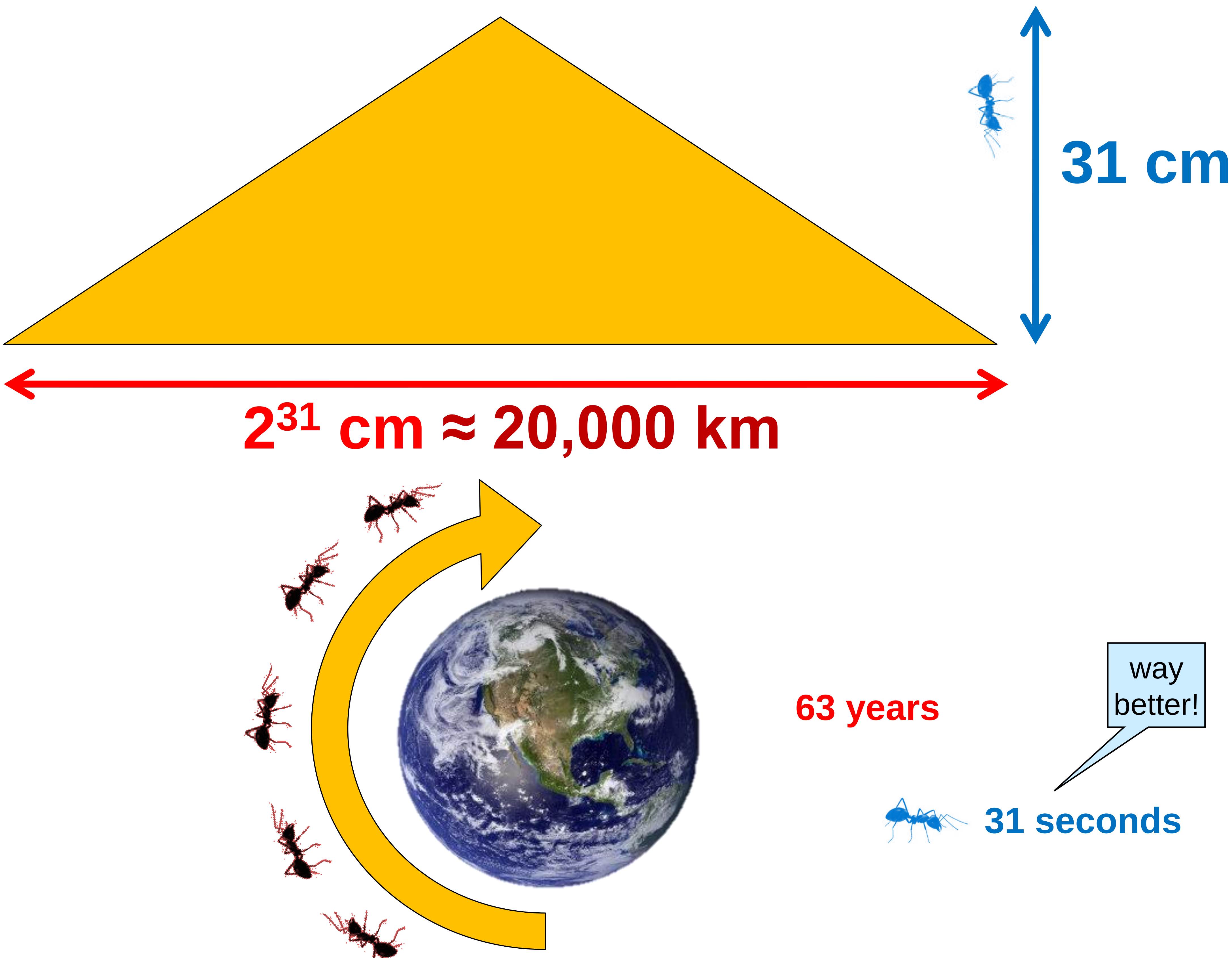
better

1 Million Items

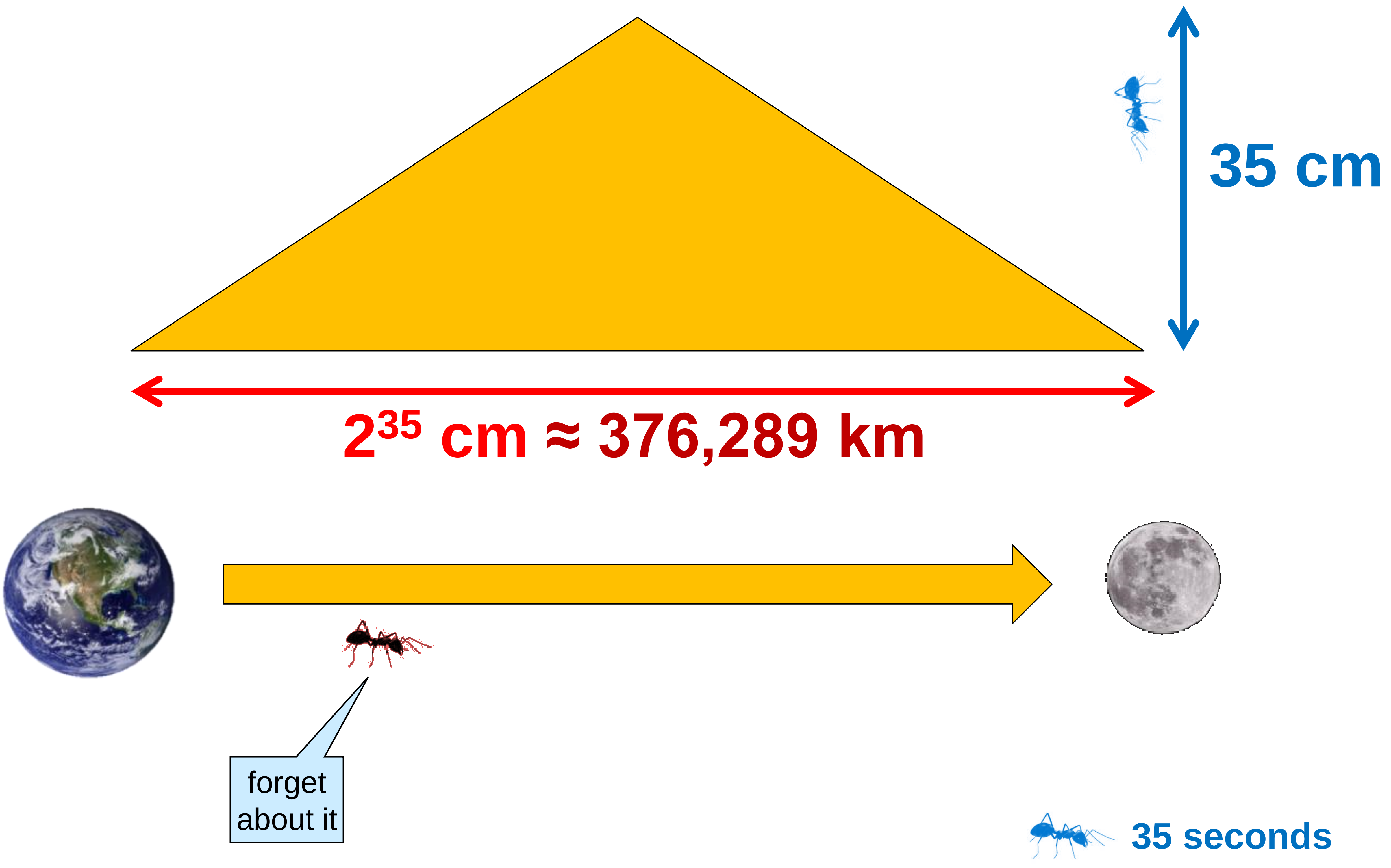




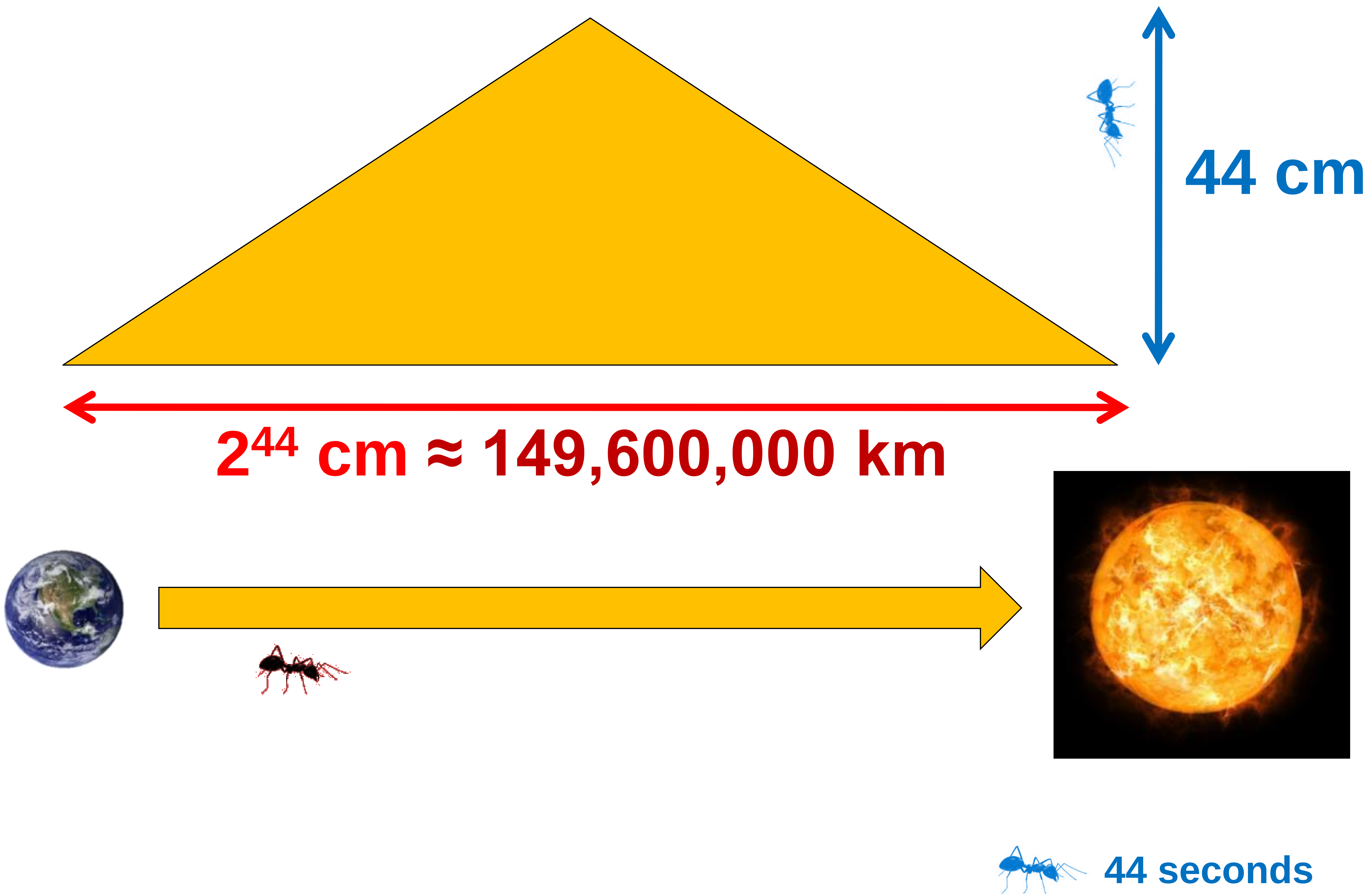
2 Billion



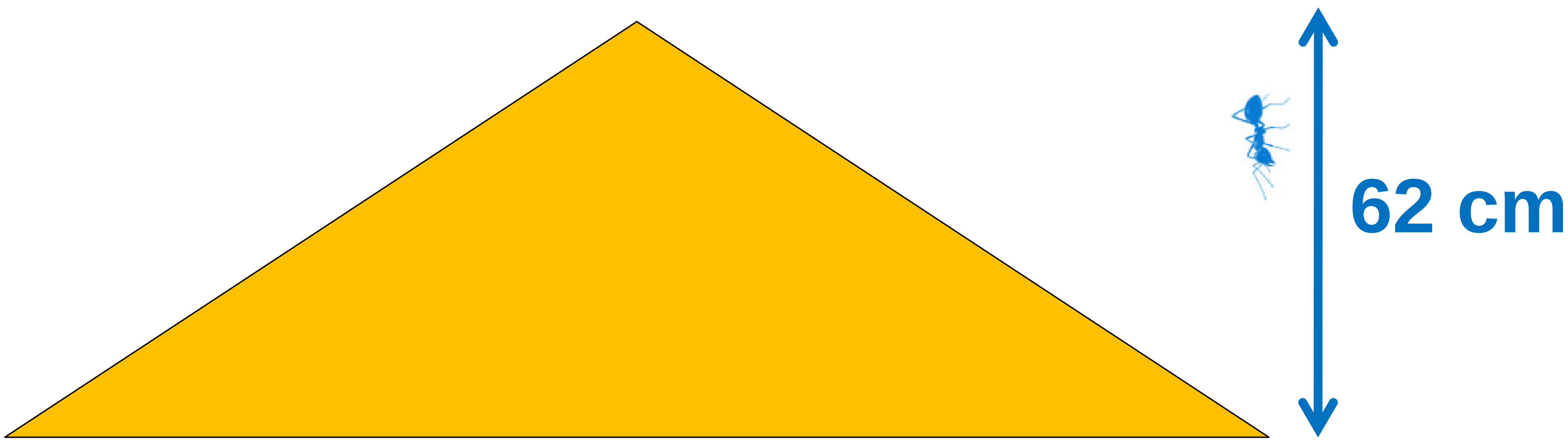
35 Billion Items



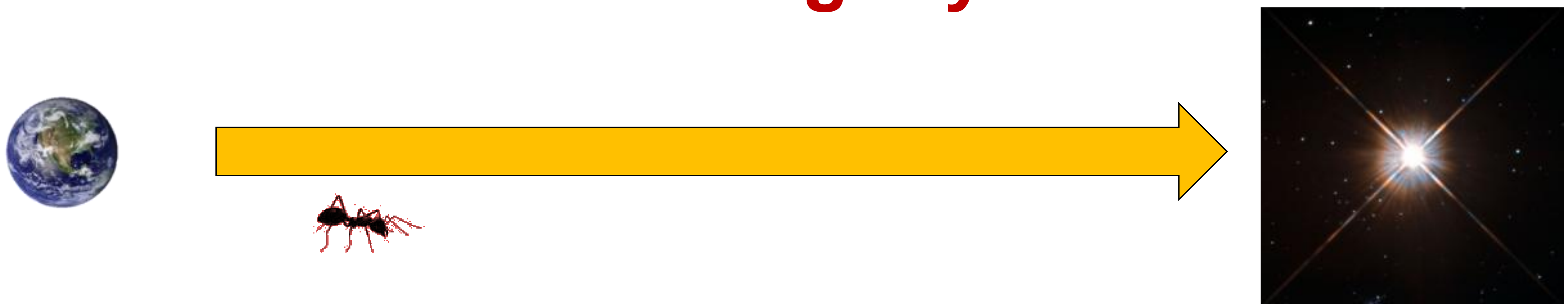
To the Sun



To the Next Star



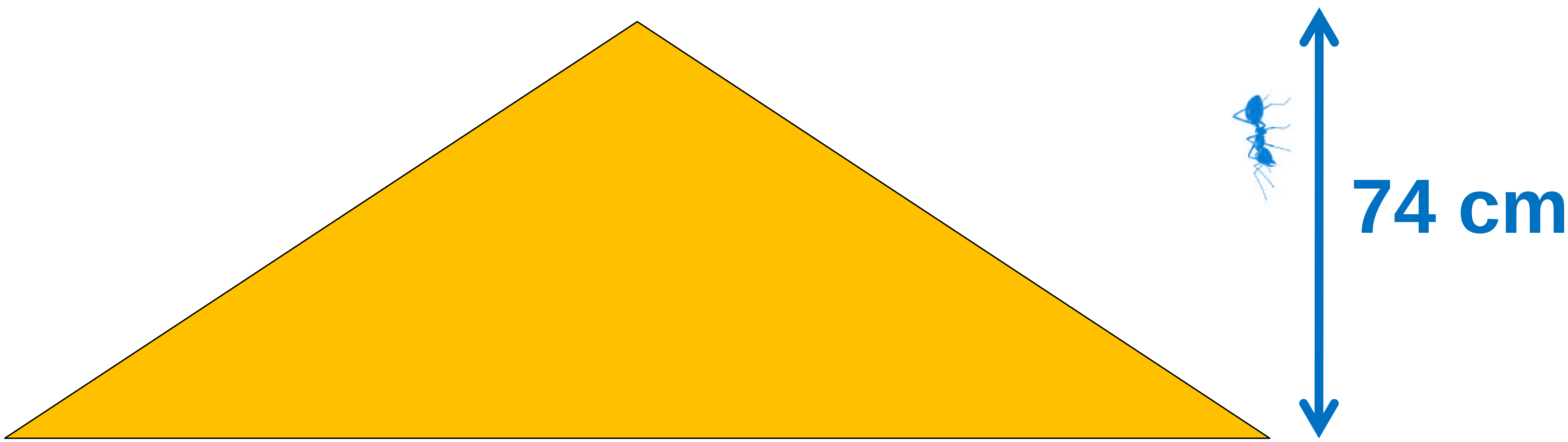
2^{62} cm ≈ 4.24 light-years



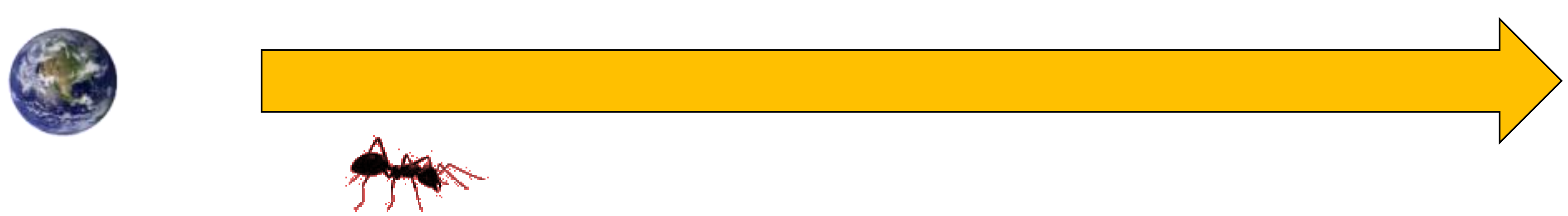
Proxima Centauri

62 seconds

To the Next Galaxy



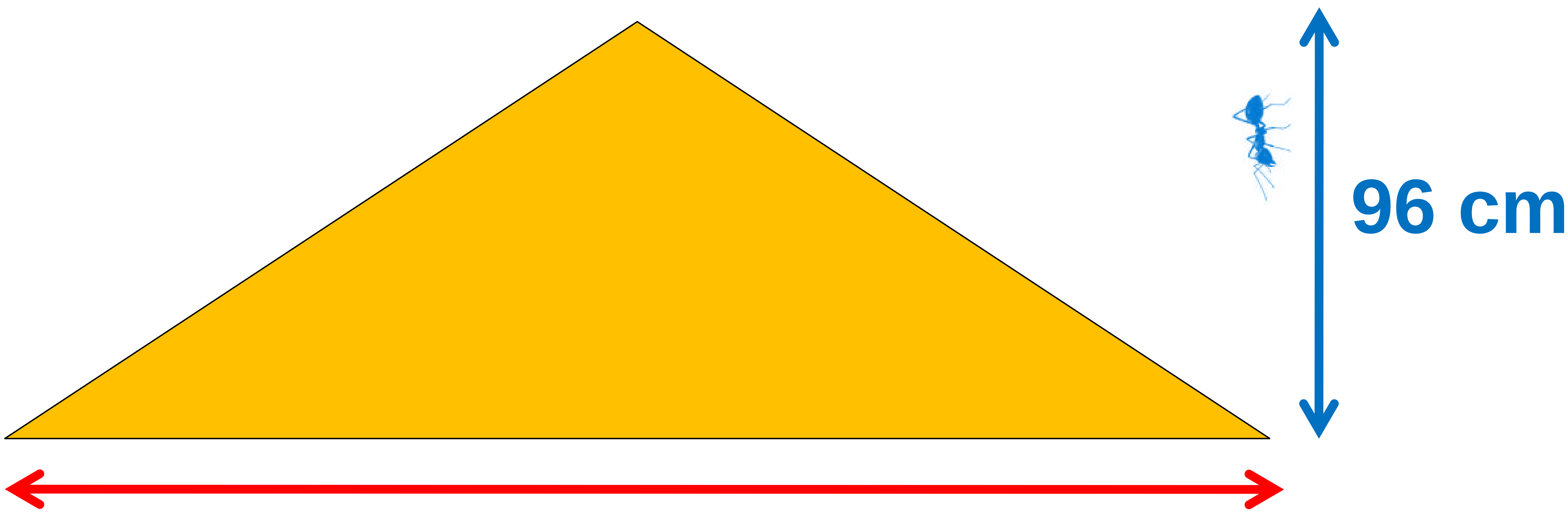
2^{74} cm ≈ *25,000 light-years*



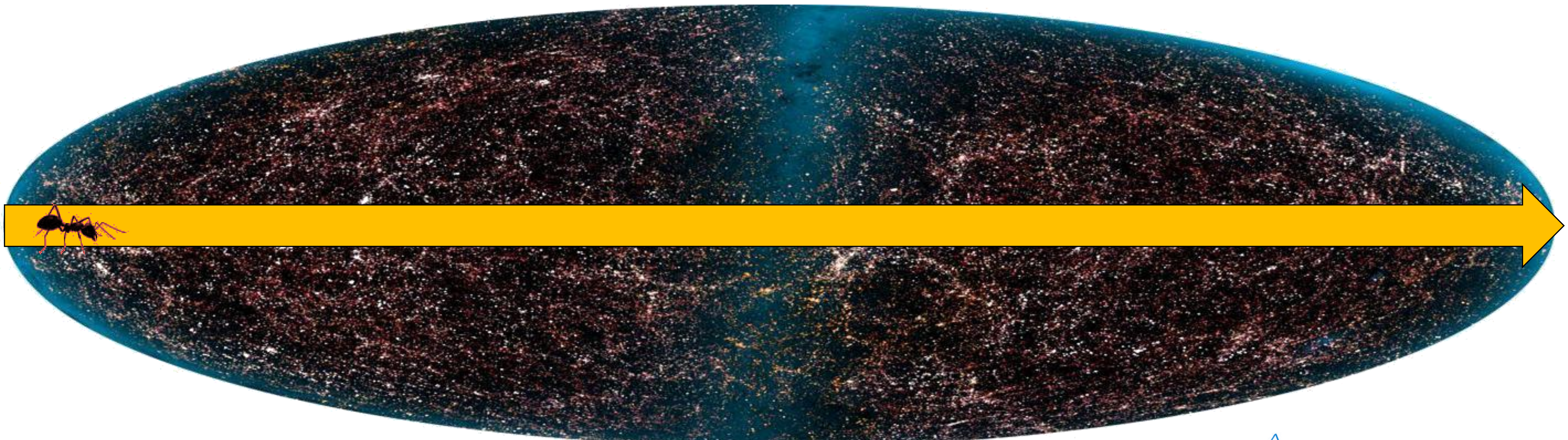
Canis Major Dwarf

 **74 seconds**

The Observable Universe

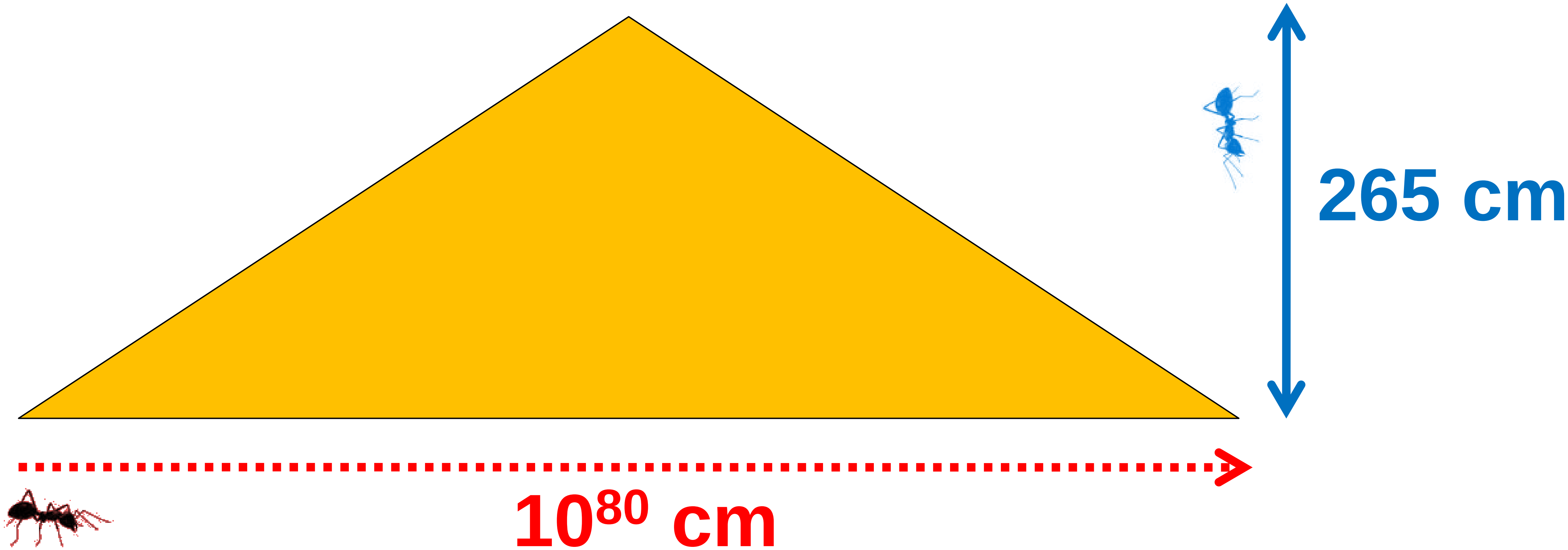


2^{96} cm ≈ *92 billion light-years*



96 seconds

All the Atoms in the Universe



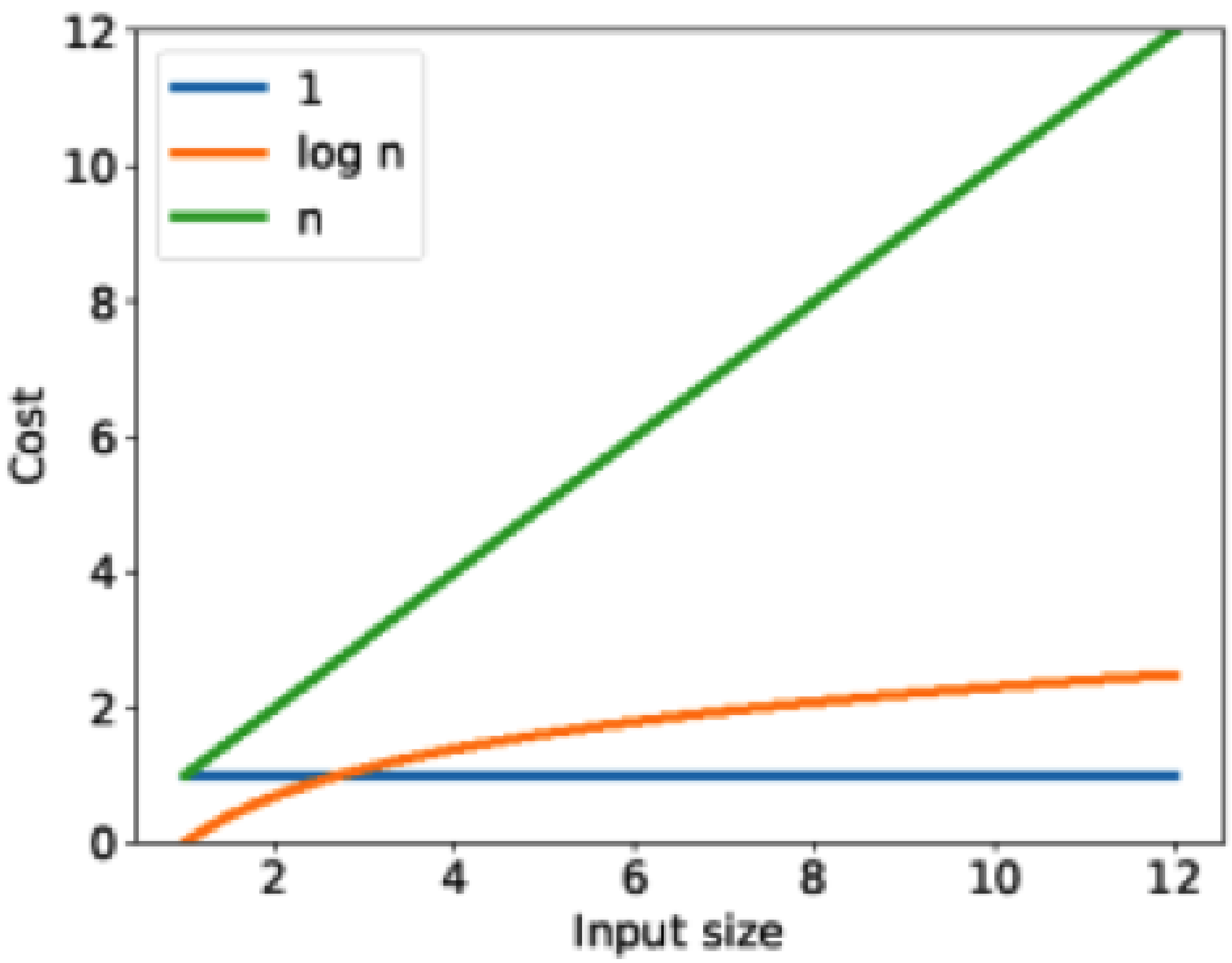
There is nothing else
we could possibly search …

265 seconds

Is *O(log n)* a Big Deal?

YES

- Constant for practical purposes
  - It takes just 265 steps to search all atoms in the universe!



log n is
really neat if you are
a computer scientist!