Sorting and Searching Algorithms

	Miles Jones	**MTThF 8:30-9:50~~pm~~** am	**CSE 4258**

August ~~1~~ 2, 2016

Sorting (or Ordering)

Section 3.1 in Rosen

vs.

* Assume elements of the set to be sorted have some underlying order

General questions to ask about algorithms

1) **What** problem are we solving? PROBLEM SPECIFICATION

2) **How** do we solve the problem? ALGORITHM DESCRIPTION

3) **Why** do these steps solve the problem? CORRECTNESSS

4) **When** do we get an answer? RUNNING TIME PERFORMANCE

Sorting: Specification: WHAT

Rosen page 196

Given a list

$$a_1,\ a_2,\ \ldots,\ a_n$$

rearrange the values so that

$$a_1 <= a_2 <= \ldots <= a_n$$

Values can be any type (with underlying total order). For simplicity, use integers.

Your approaches: HOW

- Selection (min) sort
- Bubble sort
- Insertion sort
- Bucket sort
- Merge sort
- Bogo sort
- Quick sort
- Bin~~e~~ary search tree traversal

https://en.wikipedia.org/wiki/Sorting_algorithm

Why so many algorithms?

Why so many algorithms?

Practice for homework / exam / job interviews.

Some algorithms are better than others. Wait, *better*?

Sorting is ubiquitous because it organizes data.

Knuth: “Computer manufacturers of the 1960’s estimated that more than 25% of the running time of their computers was spent on sorting.”

That was 50 years ago but now modern data centers spend Much of their time sorting.

Why so many algorithms?

Sorting is important in different environments with different constraints. E.g. low power for your iphone or needing to sort in place when there is no excess memory.

Different algorithms are optimized for different information, e.g. Bucket sort is useful when data is uniformly distributed.

The more you know about your data set, the better you can optimize

From "How" to "Why"

What makes this algorithm work?

How do you know that the resulting list will be sorted?

For loop-based algorithms:

What's the effect of each loop iteration on the list?

Have we made progress?

Loop Invariants

A loop invariant is a property that remains true after each time the body of a loop is executed.

For an iterative algorithm:

- **Step 1: Look for a loop invariant**
  - State the property precisely
- **Step 2: Prove that it is invariant**
  - It must be true after any number of loop iterations
- **Step 3: Use the invariant to prove correctness**
  - Show that when the loop is finished, the invariant guarantees that we've reached a solution

3 Step Plan

Selection Sort (MinSort)

"Find the first name alphabetically, move it to the front. Then look for the next one, move it, etc."

Selection Sort (MinSort) Pseudocode

Rosen page 203, exercises 41-42

```
procedure selection sort($a_1$, $a_2$, ..., $a_n$: real numbers with n >=2 )
for i := 1 to n-1
    m := i
    for j:= i+1 to n
        if ( $a_j$ < $a_m$ ) then m := j
    interchange $a_i$ and $a_m$

{ $a_1$, ..., $a_n$ is in increasing order}
```

Selection Sort Example

```
procedure selection sort($a_1$, $a_2$, ..., $a_n$: real numbers with n >=2 )
for i := 1 to n-1
    m := i
    for j:= i+1 to n
        if ( $a_j$ < $a_m$ ) then m := j
    interchange $a_i$ and $a_m$

{ $a_1$, ..., $a_n$ is in increasing order}
```

A

a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8

20, 9, 44, 13, 5, 11, 22, 10

$i := 1$ $m = 1$ $j = 2$

$a_2 < a_1$ yes $m := 2$

$j = 3$

$a_3 < a_2$ no

$j = 4$

$a_4 < a_2$ no

$j = 5$

$a_5 < a_2$ yes $m := 5$

at the end of $i = 1$

$m := 5$

interchange a_1 and a_5

$\boxed{5}$ $\underbrace{9\ 44\ 13\ 20\ 11\ 22\ 10}$

lowest number.

$\boxed{i := 2}$

5 9 44 13 20 11 22 10

Selection Sort

In groups of 3 or so …. Discuss possible invariants for Selection sort.

Remember that an invariant is true for all iterations, not just at the end.

After k iterations (of the outer loop.)

All elements from $\underbrace{a_1 \dots a_k}$ are sorted

and they are the k smallest numbers.

Selection Sort (MinSort) Correctness: WHY

Loop invariant: After the k^{th} time through the outer loop, the first k elements of the list are the k smallest list elements in order.

Selection Sort (MinSort) Correctness: WHY

Loop invariant: After the k^{th} time through the outer loop, the first k elements of the list are the k smallest list elements in order.

induction

How can we show that this loop invariant is true?

Once we do, why can we conclude that the program is correct?

(stating the loop invariant with $k=n$.)

Selection Sort (MinSort) Correctness: WHY

Loop invariant: After the k^{th} time through the outer loop, the first k elements of the list are the k smallest list elements in order.

How can we show that this loop invariant is true?

Once we do, why can we conclude that the program is correct?

Selection Sort (MinSort) Correctness: WHY

Loop invariant: After the $\texttt{k}^{\text{th}}$ time through the outer loop, the first `k` elements of the list are the `k` smallest list elements in order.

Selection Sort (MinSort) Correctness: WHY

Loop invariant: After the k^{th} time through the outer loop, the first k elements of the list are the k smallest list elements in order.

How can we show that this loop invariant is true?

Induction ☺

Once we do, why can we conclu ram is correct?

Selection Sort (MinSort) Correctness: WHY

Loop invariant: After the $\mathtt{k}^{\text{th}}$ time through the outer loop, the first $\mathtt{k}$ elements of the list are the $\mathtt{k}$ smallest list elements in order.

How can we show that this loop invariant is true?

Induction on the number of passes through the loop.

Statement: After the k^{th} time through the outer loop, the first k elements of the list are the k smallest list elements in order.

Induction variable (k): *the number of times through the loop.*

Base case: **Need to show** the statement holds for ~~k=0~~, ~~before the loop~~
k=1

After the 1st time through the first element is the smallest

you can start at k=1 or k=0.

Statement: After the k^{th} time through the outer loop, the first k elements of the list are the k smallest list elements in order.

Inductive step: Let k be a positive integer.

Induction hypothesis: Suppose the statement holds after k-1 times through the loop.

Need to show that the statement holds after k times through the loop.

after k-1 times through the first k-1 elements are the k-1 smallest elements in order. When i=k. the body of the loop finds the smallest list element in $a_k \ldots a_n$. Then it puts it in position a_k. Therefore $a_1 \cdots a_k$ are the k smallest elements in order.

Statement: After the k^{th} time through the outer loop, the first k elements of the list are the k smallest list elements in order.

Now prove that the algorithm works:

We have proved the above statement by induction for all k=0..n

What does it mean when k=n?

Statement: After the k^{th} time through the outer loop, the first k elements of the list are the k smallest list elements in order.

Now prove that the algorithm works:

We have proved the above statement by induction for all k=0..n

What does it mean when k=n?

After the nth iteration (after final iteration) the first n elements (all the elements) are the n smallest list elements in order.

all elements .

IOW: After the final iteration, all elements are in order!!!!

Proof by loop invariant

1. State the loop invariant.
2. Prove the loop invariant using induction.
3. Show that after the final loop, the loop invariant coincides with the goal of the algorithm.

Proving Loop Invariants

Induction variable (k): *the number of times through the loop.*

Base case: **Need to show** the statement h ls for k before the loop

Inductive step: Let k be a positive

Very common pattern.
Need help ?

Induction hypothesis: Suppose the sta ment h ds after k-1 times through the loop.

Need to show that the statement holds after k times through the loop.

Insertion Sort Pseudocode

Rosen page 198

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
20	9	44	13	5	11	22	18

```
procedure insertion sort($a_1$, $a_2$, ..., $a_n$: real numbers with n >=2 )
for j := 2 to n
    i := 1
    while $a_j$ > $a_i$
        i := i+1
    m := $a_j$
    for  k := 0 to j-i-1
        $a_{j-k}$ := $a_{j-k-1}$
    $a_i$ := m
```

{ a_1, ..., a_n is in increasing order}

$j=2$ $i=1$ is $a_2 > a_1$ no

$m := 9$

$k := 0 \ldots 2-1-1=0$

$a_2 := a_1 = 20$

$a_1 := 9$

Insertion Sort Loop invariant

Rosen page 198

State the loop invariant:

after k iterations

$a_1 \ldots a_{k+1}$ is a list of sorted elements.

Insertion Sort Loop invariant

Rosen page 198

State the loop invariant:

after the kth iteration, the elements a_1,…a_(k+1) are in order.

Insertion Sort Loop invariant

Rosen page 198

State the loop invariant:

after the kth iteration a_1,…a_(k+1) are in order.

Base Case: k=0
a_1 is trivially in order.

Insertion Sort Loop invariant

Rosen page 198

State the loop invariant:

after the kth iteration a_1,…a_(k+1) are in order.

Induction step: Assume after the kth iteration a_1…a_(k+1) are in order then show that after the (k+1)st iteration, a_1…a_(k+2) are in order.

Insertion Sort Loop invariant

State the loop invariant:
after the kth iteration a_1,…a_(k+1) are in order.

Induction step: Assume after the kth iteration a_1…a_(k+1) are in order then show that after the (k+1)st iteration, a_1…a_(k+2) are in order.

Du ing ~~after~~ the (k+1)st iteration, j = k+2 so then the algorithm inserts a_(k+2) into the sorted list of a_1…a_(k+1). Thus resulting in a sorted list of a_1…a_(k+2).

Insertion Sort correctness

We have proved the loop invariant for k = 0…(n-1):
after the kth iteration a_1,…a_(k+1) are in order.

What happens when k = n-1?

After the (n-1)st iteration (after the final iteration), a_1,…a_n are in order (the whole list.)

IOW: After the final iteration, all elements are in order.

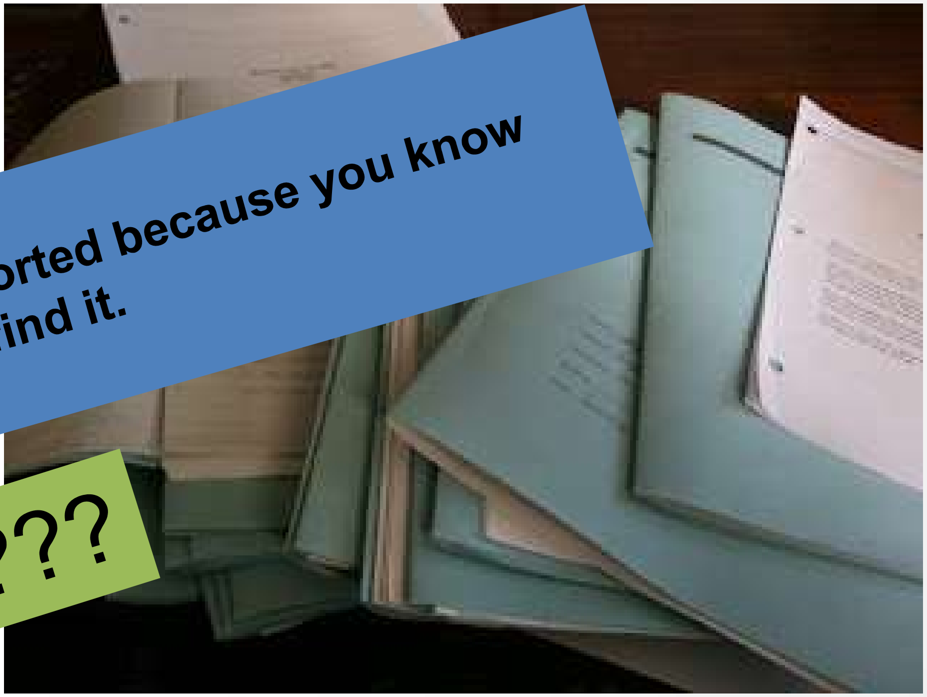
Why sort?

A TA facing a stack of exams needs to input all 400 scores into a spreadsheet where the students are listed in alphabetical order.

OR

You

value



It's easier to access data when it is sorted because you know exactly where to find it.

Really???

Sorting helps with searching

Two searching algorithms:

One that works for any data, sorted or not

One that is much faster, but relies on the data being sorted

searching

- Given an array `A[1], A[2], A[3], ..., A[n]` and a target value `x`,
    - find an index `j` where `x = A[j]`
- or determine that no such index exists because `x` is not in the array;
- Return j=0 if no index exists

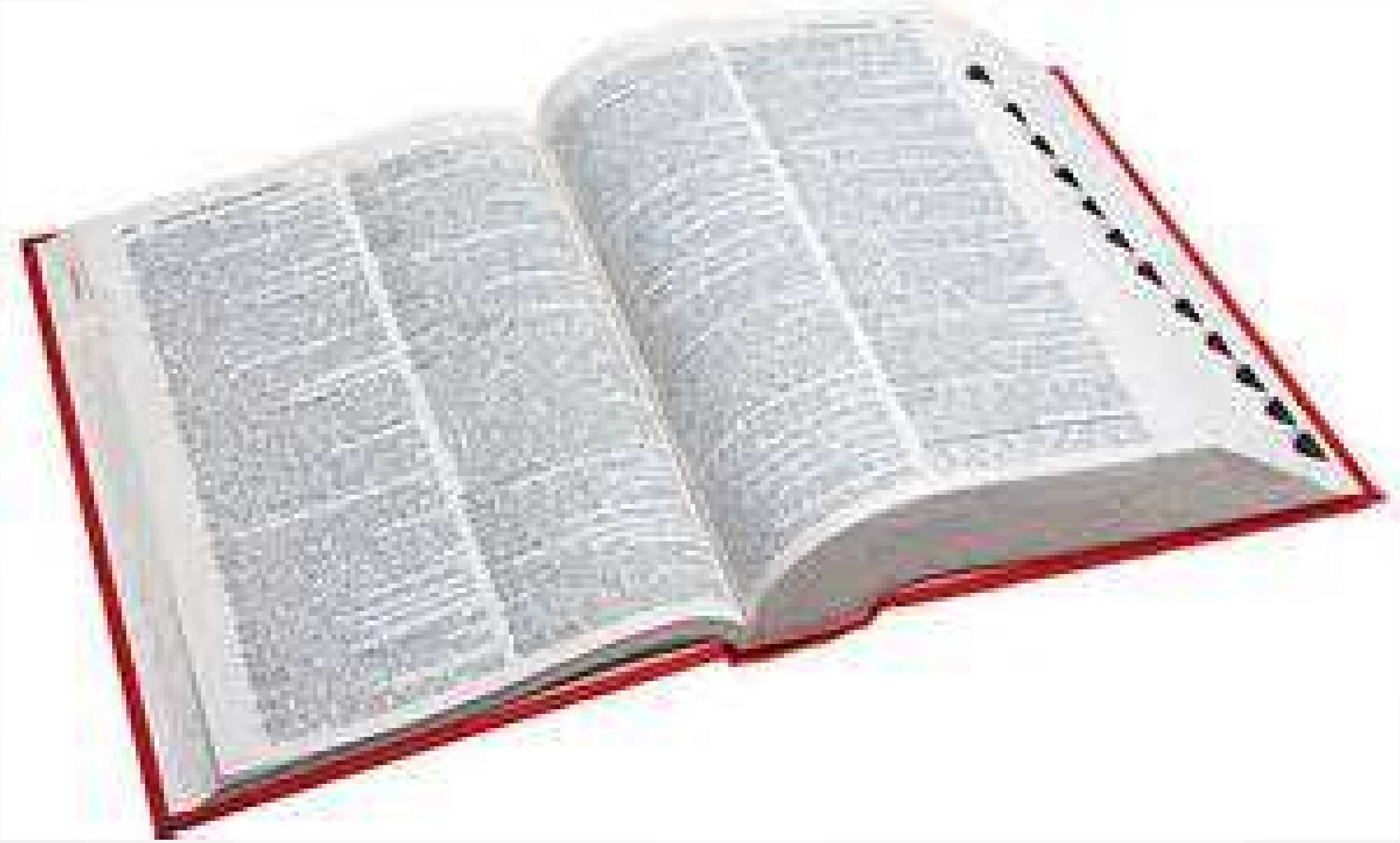
searching

- Suppose our array is
    - `A[1]=2, A[2]=5, A[3]=0, A[4]=1.`
- If `x = 1`, what `j` would the search problem find?
    - `A) j=2`
    - `B) j=4`
    - `C) j=0`
    - `D) no such j exists`

searching for a movie at Blockbuster

BLOCKBUSTER VIDEO

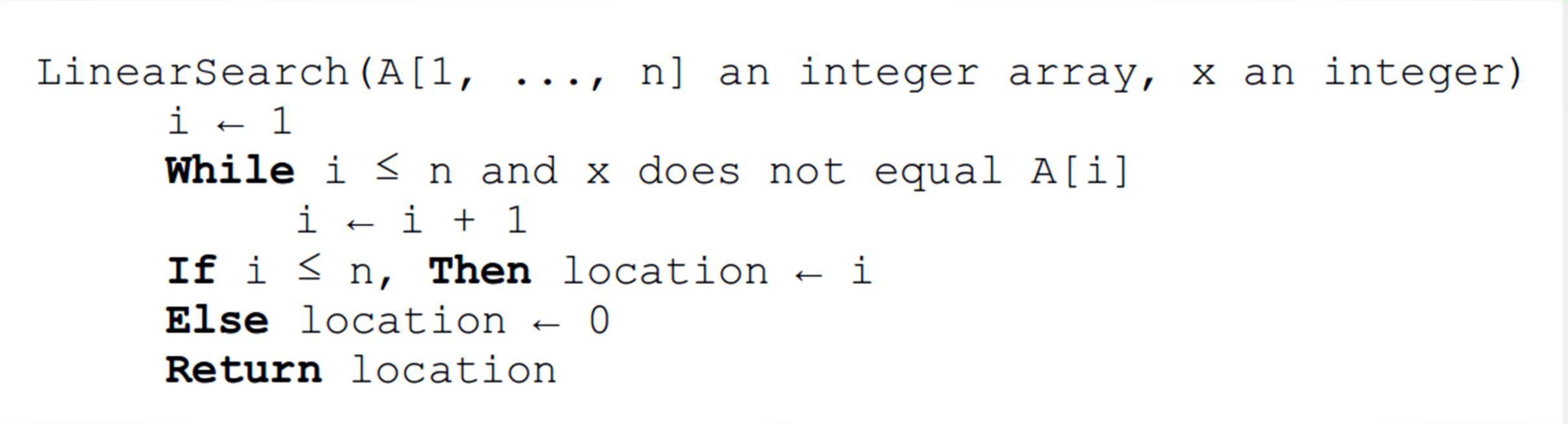
searching for a word in a dictionary



Linear Search

- Search through the array one index at a time, asking
  - *"Is this my target value?"*
- at each position.
- If you answer yes, return the position where you found it.

Linear Search



```
LinearSearch(A[1, ..., n] an integer array, x an integer)
     i ← 1
     While i ≤ n and x does not equal A[i]
          i ← i + 1
     If i ≤ n, Then location ← i
     Else location ← 0
     Return location
```

Proof of Correctness

- Why is this algorithm correct?

- Easy case: if the algorithm returns some i in the range 1 through n, it must be because x= A[i]

- Remaining case: ?

Invariant for main loop

- ``While x is not A[i] and i <= n do: i ++',

suppose x is not in the list.

- Invariant: If after the loop, i=k, then ????

After K iterations x is not in a[1] . . . a[k].

Invariant for main loop

- ``While x is not A[i] and i <= n do: i ++',

- Invariant: If after the loop, i=k, then x is not among A[1],,...A[k-1]

- Base case: ~~k=1~~; x is not among the empty set

k=0

Invariant for main loop

- ``While x is not A[i] and i <= n do: i ++',
- Invariant: If after the loop, i=k, then x is not among A[1],,...A[k-1]
- Base case: k=1; x is not among the empty set
- Induction step. Assume i=k+1. Then in the previous iteration, i was k. Thus, x is not among the first k-1 elements of the array. In the current iteration, we compared x to A[k]. Since we incremented i, x was also not A[k].

If Linear Search returns 0

- Linear search only returns 0 if i = n+1

- Then the invariant says x is not in the first n positions of the array

- But this means x is not in the array, so 0 is the correct answer

How fast is linear search?

- Suppose you have an unsorted array and you are searching for a target value `x` that you know is in the array somewhere.
- What position for `x` will cause linear search to take the longest amount of time?
  - A) `x` is the first element of the array
  - B) `x` is the last element of the array
  - C) `x` is somewhere in the middle of the array

How fast is linear search?

- Say our array `A` has `n` elements and we are searching for `x`.

- The time it takes to find `x` (or determine it is not present) depends on the number of *probes*, that is the number of entries we have to retrieve and compare to `x`.

How many Probes?

- Say our array `A` has `n` elements and we are searching for `x`.
  - Worst-case scenario: n
  - Best-case scenario: 1
  - On average: $\sim n/2$

How many Probes?

- Say our array `A` has `n` elements and we are searching for `x`.

- Worst-case scenario: n probes

- Best-case scenario: 1 probe

- On average: around n/2 probes

Searching a sorted array

- How would you search through a pile of alphabetized papers to find the one with your name on it?

How does having a sorted array help us?

- Suppose we probe `A` at position `m`. What do we learn?
  - If `x=A[m]`,

    you found it!

  - If `x<A[m]`,

    its higher in the stack

  - If `x>A[m]`,

    lower in the stack.

How does having a sorted array help us?

- Suppose we probe `A` at position `m`. What do we learn?
  - If `x=A[m]`, we are done
  - If `x<A[m]`, then if `x` is in the array, it is in position `p` for `p<m`.

    `m-1` positions remain to be checked
  - If `x>A[m]`, then if `x` is in the array, it is in position `p` for `p>m`.

    `n-m` positions remain to be checked

How do we choose where to probe?

- If our array has 100 elements and we probe at position 5, it is most likely that the target is in the second half, so we'll have to search 95 entries in the worst case.

- How do we make the worst case as favorable as possible?
  - **Probe in the middle.**
- If we are searching the subarray `A[i], …, A[j]`, probe at index

$$\left\lfloor \frac{i+j}{2} \right\rfloor$$

Binary Search: The Approach

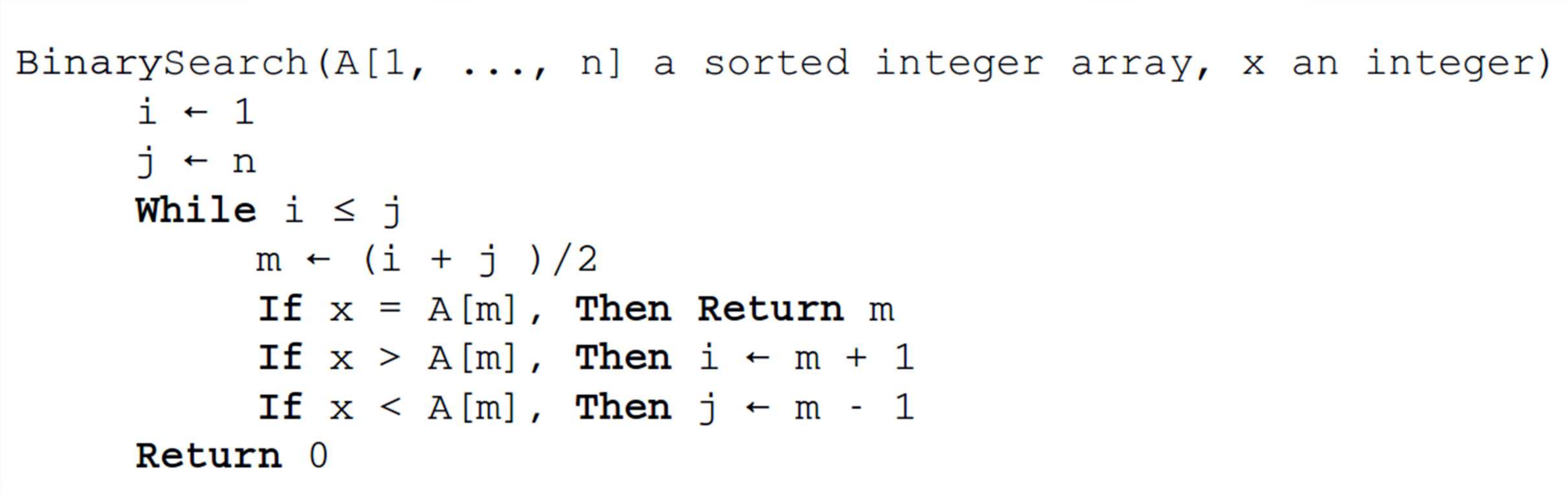
- Probe in the middle of the array.

- Based on what you find there, determine which half to search in next.

- Continue until the target is found or you can be sure the target is not in the array.

- Return the location of the target, or 0 if it's not in the array

Binary Search: The Approach



```
BinarySearch(A[1, ..., n] a sorted integer array, x an integer)
     i ← 1
     j ← n
     While i ≤ j
          m ← (i + j )/2
          If x = A[m], Then Return m
          If x > A[m], Then i ← m + 1
          If x < A[m], Then j ← m - 1
     Return 0
```

Binary Search is Telling the Truth

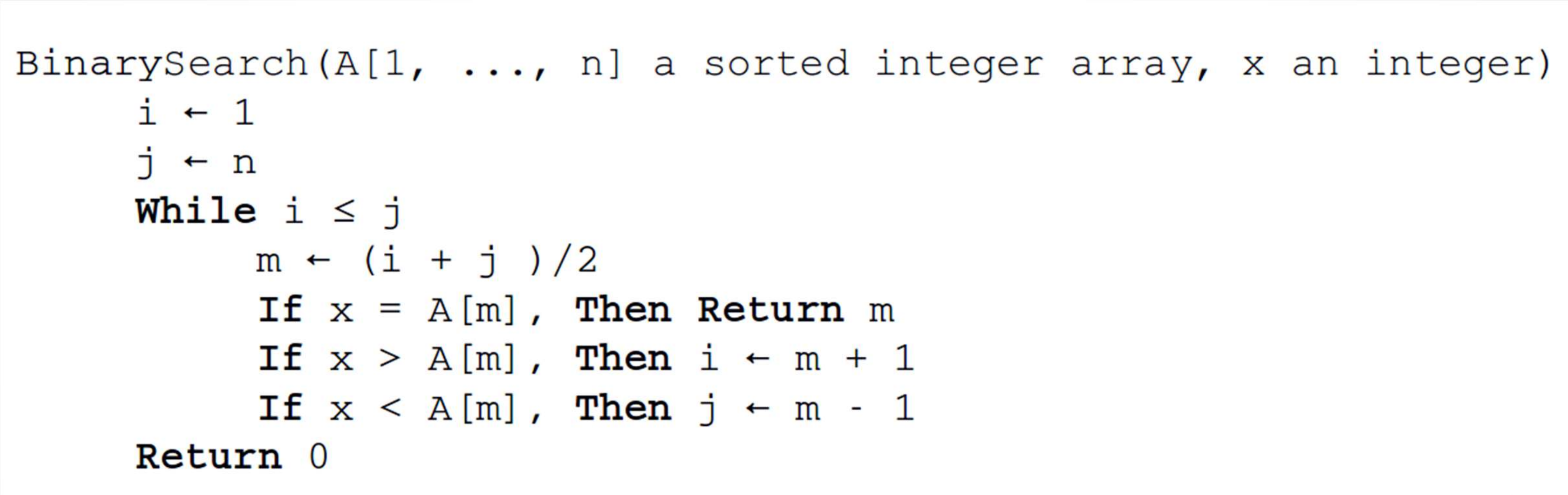
- We must show two things:

1) If binary search returns some nonzero position `p`, then `A[p]= x`.

2) If binary search returns 0, then there is no position `p` for which `A[p]=`

x

1) If it returns nonzero `p`, then `A[p]=x`.



```
BinarySearch(A[1, ..., n] a sorted integer array, x an integer)
    i ← 1
    j ← n
    While i ≤ j
        m ← (i + j )/2
        If x = A[m], Then Return m
        If x > A[m], Then i ← m + 1
        If x < A[m], Then j ← m - 1
    Return 0
```

What is the contrapositive of (2)?

$p \to q$

contrapositive

$\neg q \to \neg p.$

2) If binary search returns 0, then there is no position `p` for which `A[p]=x`.

if there is a position p for which $A[p] = x$. then

Binary search does not return 0.

What is the contrapositive of (2)?

2) If binary search returns 0, then there is no position `p` for which `A[p]=x`.

Contrapositive: If there is a position `p` for which `A[p]=x`, then binary search doesn't return 0.

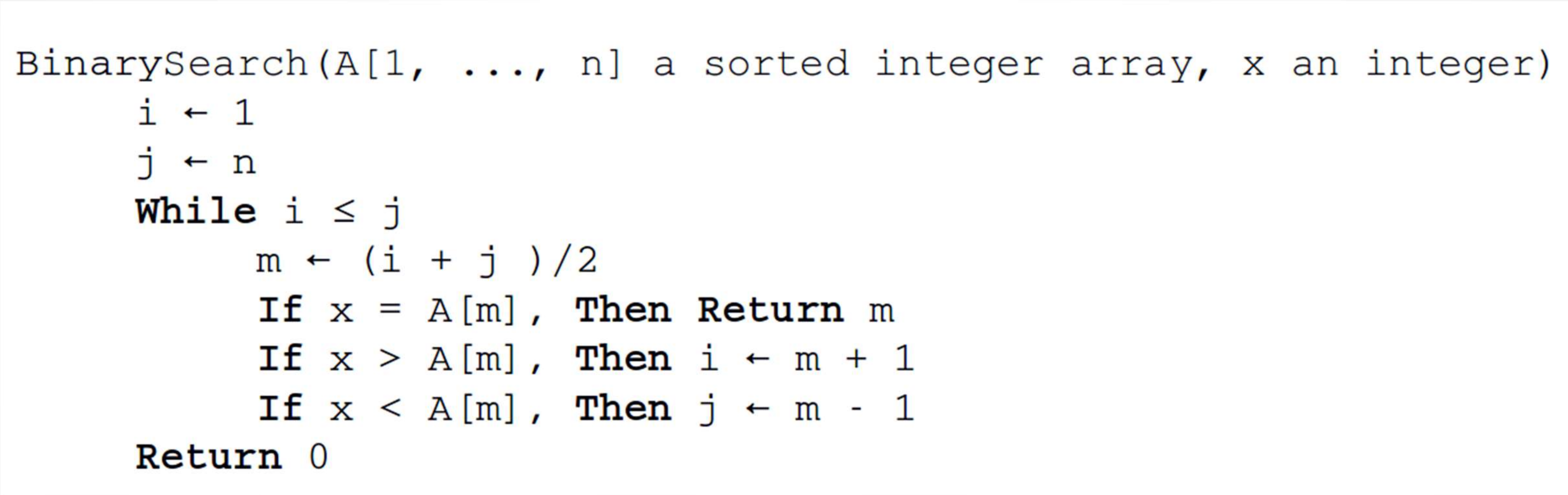
Loop Invariant

– **Want to show:** If there is a position `p` for which `A[p]=x`, then binary search doesn't return 0.

– **Loop invariant:** After the algorithm has set i and j, Suppose there is a position `p` for which `A[p]=x`. Then:

$$i \le p \le j$$

Loop Invariant $i \le p \le j$



```
BinarySearch(A[1, ..., n] a sorted integer array, x an integer)
    i ← 1
    j ← n
    While i ≤ j
        m ← (i + j )/2
        If x = A[m], Then Return m
        If x > A[m], Then i ← m + 1
        If x < A[m], Then j ← m - 1
    Return 0
```

Binary search terminates

- With each probe, you reduce the size of the subarray you are searching to at most half of its previous size.

Number of probes	Max size of subarray
1	n/2
2	n/4
3	n/8
4	n/16
…	…
w	?

Binary search terminates

- With each probe, you reduce the size of the subarray you are searching to at most half of its previous size.

Number of probes	Max size of subarray
1	n/2
2	n/4
3	n/8
4	n/16
…	…
w	$n/2^w$

Binary search terminates

- With each probe, you reduce the size of the subarray you are searching to at most half of its previous size.
- How many probes w do we need to make sure our search has terminated? That is, what value of w is sure to make the subarray size less than 1?

$$\frac{n}{2^w} < 1$$

$$n < 2^w$$

$$\log(n) < w$$

- The smallest integer w that is sure to be greater than $log(n)$ is

$$w = \lfloor \log(n) \rfloor + 1$$

Comparison with Linear Search

- Therefore, we have shown that it takes at most

$$w = \lfloor \log(n) \rfloor + 1$$

- probes for binary search to terminate. This is the worst case.

- By comparison, linear search takes as many as n probes.Even in the average case, linear search takes about $n/2$ probes.

n	n/2	log(n)
100	50	6.6
200	100	7.6
300	150	8.2

The cost of binary search

- While binary search is faster than linear search, it depends upon your array being sorted.

- What if you have an unsorted array you need to search? Should you sort it first so you can use the better search?

The cost of binary search

- There's a tradeoff between the cost of sorting an array and the benefit of being able to search faster.

- If you need to search the same array many times, it becomes more worthwhile to invest the initial time in sorting your array.