Computer Science 331
Algorithms for Searching

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #21

Outline

The "Searching" Problem

Precondition 1:

a) A is an array with length $A.length = n \geq 1$ storing values of some type T
b) *key* is a value of type T that is stored in A

Postcondition 1:

a) The value returned is an integer i such that $A[i] = key$
b) A and *key* are not changed

The "Searching" Problem, continued

Precondition 2:

a) A is an array with length $A.length = n \geq 1$ storing values of some type T
b) *key* is a value of type T that is *not* stored in A

Postcondition 2:

a) A `notFoundException` is thrown
b) A and *key* are not changed

Linear Search

Idea: Compare $A[0], A[1], A[2], \ldots$ to *key* until either

- *key* is found, or
- we run out of entries to check

```
int LinearSearch(T key)
  i = 0
  while (i < n) and (A[i] ≠ key) do
    i = i + 1
  end while
  if i < n then
    return i
  else
    throw KeyNotFoundException
  end if
```


Example

	0	1	2	3	4	5	6	7	8	9	10
A:	43	30	6	18	-3	49	2	21	29	35	23

Search for 18 in the array A :

-
-
-
-

Partial Correctness

Loop Invariant: The following properties are satisfied at the beginning of each execution of the loop body:

- i is an integer such that $0 \leq i < n$
- $A[h] \neq key$ for $0 \leq h \leq i$
- A and *key* have not been changed

Proving the Loop Invariant: use induction on number of executions of the loop body (i)

Base Case:

-
-
-

Partial Correctness (inductive step)

Inductive hypothesis: assume that the loop body is executed at least $i \geq 0$ times and that the loop invariant is satisfied at the beginning of the ith execution.

By inspecting the code, we see that at the *end* of the ith execution:

-
-
-

If there is a $i+1$st execution of the loop body, then the loop test must pass after the end of the ith execution, implying:

-
-
-

Partial Correctness (applying the loop invariant)

At the *end* of the loop (loop condition fails), the following properties are satisfied:

- i is an integer such that $0 \leq i \leq n$
- $A[h] \neq key$ for $0 \leq h < i$
- A and key have not been changed
- Either $i = n$ or ($i < n$ and $A[i] = key$)

Conclusion: algorithm postconditions are satisfied because

-
-

Termination and Efficiency

Loop Variant: $f(n,i) = n - i$

Proving the Loop Variant:

- $f(n,i)$ is a decreasing integer function because integer i increases by one after each loop body execution
- $f(n,i) = 0$ when $i = n$, loop terminates (worst case) when $i \geq n$

Application of Loop Variant:

-
-
-

The "Searching" Problem in a Sorted Array

Precondition 1:

a) A is an array with length $A.length = n \geq 1$ storing values of some *ordered* type T
b) $A[i] < A[i+1]$ for every integer i such that $0 \leq i < n-1$
c) key is a value of type T that is stored in A

Postcondition 1:

a) The value returned is an integer i such that $A[i] = key$
b) A and key are not changed

The "Searching" Problem in a Sorted Array

Precondition 2:

a) A is an array with length $A.length = n \geq 1$ storing values of some *ordered* type T
b) $A[i] < A[i+1]$ for every integer i such that $0 \leq i < n-1$
c) key is a value of type T that is *not* stored in A

Postcondition 2:

a) A `notFoundException` is thrown
b) A and key are not changed

Linear Search

Idea: compare $A[0], A[1], A[2], \ldots$ to k until either k is found or

- we see a value larger than k — all future values will be larger than k as well! — or
- we run out of entries to check

```
int LinearSearch(T key)
  i = 0
  while (i < n) and  do
    i = i + 1
  end while
  if (i < n) and (A[i] = k) then
    return  i
  else
    throw KeyNotFoundException
  end if
```


Example

	0	1	2	3	4	5	6	7	8	9	10
A:	-3	2	6	18	21	23	29	30	35	43	49

Search for 17 in the array A :

-
-
-
-

Partial Correctness

Loop Invariant: The following properties are satisfied at the beginning of each execution of the loop body:

- i is an integer such that $0 \leq i < n$
- $A[h] < key$ for $0 \leq h \leq i$
- A and key have not been changed

Proving the Loop Invariant: use induction on number of executions of the loop body (i)

Base Case:

-
-
-

Partial Correctness (inductive step)

Inductive hypothesis: assume that the loop body is executed at least $i \geq 0$ times and that the loop invariant is satisfied at the beginning of the ith execution.

By inspecting the code, we see that at the *end* of the ith execution:

-
-
-

If there is a $i + 1$st execution of the loop body, then the loop test must pass after the end of the ith execution, implying:

-
-
-

Partial Correctness (applying the loop invariant)

At the *end* of the loop (loop condition fails), the following properties are satisfied:

- i is an integer such that $0 \leq i \leq n$
- $A[h] < key$ for $0 \leq h < i$
- A and key have not been changed
- Either $i = n$ or $i < n$ and $A[i] \geq key$

Conclusion: algorithm postconditions are satisfied because

- Case 1 ($i = n$):
- Case 2 ($i < n$ and $A[i] = key$):
- Case 3 ($i < n$ and $A[i] > key$):

Termination and Efficiency

Loop Variant: $f(n, i) = n - i$

Proving the Loop Variant:

- same as before

Application of Loop Variant:

- same as before (worst-case runtime is also $\Theta(n)$)

Note: although the worst-case involves examining all elements of the array, fewer will be examined on average

- improves on unsorted case (all array elements *must* be examined to determine that k is not in the array)

Binary Search

Idea: suppose we compare key to $A[i]$

- if $key > A[i]$ then $key > A[h]$ for all $h \leq i$.
- if $key < A[i]$ then $key < A[h]$ for all $h \geq i$.

Thus, comparing key to the *middle* of the array tells us a lot:

- can eliminate half of the array after the comparison

`int` **binarySearch**(`T` *key*)
  return bsearch(0, $n-1$, *key*)

Specification of Requirements for Subroutine

Calling Sequence: `int` bsearch(`int` *low*, `int` *high*, `int` *key*)

Preconditions 1 and 2: add the following to the corresponding precondition in the "Searching in a Sorted Array" problem:

d) *low* and *high* are integers such that
  - $0 \leq low \leq n$
  - $-1 \leq high \leq n - 1$
  - $low \leq high + 1$
  - $A[h] < key$ for $0 \leq h < low$
  - $A[h] > key$ for $high < h \leq n - 1$

The corresponding postcondition can be used without change.

Pseudocode: The Binary Search Subroutine

```
int bsearch(int low, int high, T ket)
  if low > high then

  else
    mid = ⌊(low + high)/2⌋
    if (A[mid] > key) then
      return
    else if (A[mid] < key) then
      return
    else
      return
    end if
  end if
```


Example

	0	1	2	3	4	5	6	7	8	9	10
A:	-3	2	6	18	21	23	29	30	35	43	49

Search for 18 in the array A :

-
-
-

Partial Correctness

Assumptions

- bsearch is called with the precondition satisfied
- Calls to bsearch *within the code* behave as expected

Case: $low > high$

-

Case: $low = high$

-
-

Case: $low < high$

-
-

Efficiency

Case: $low \geq high$

-

Case: $low < high$: Consider $i = \lceil \log_2(high - low + 1) \rceil$

- *Result of Function Call:*
  -
- *What Happens if* $i = 0$:
  -
- *Initial Value:*
  -
- *Conclusion:*
  -

References

`Java.utils.Arrays` package contains several implementations of binary search

- arrays with `Object` or generic entries, or entries of any basic type
- slightly different pre and postconditions than presented here

Textbook: Section 9.3.1