Computer Science 331

Algorithms for Searching

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #21

Outline

The "Searching" Problem

Precondition:

A: Array of length n, for some integer $n \geq 1$, storing objects of some type

k: An object of the type that might be found in A

Postcondition:

i: An integer such that $0 \leq i < n$ and $A[i] = k$
(The array A and key k were not changed.)

Exceptions:

`KeyNotFoundException`: Thrown if k is not found in A

Linear Search

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

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

LinearSearch(k)
  $i = 0$
  while $(i < n)$ **and** $(A[i] \neq k)$ **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 :

- $i = 0 : A[0] = 43 \neq 18$
- $i = 1 : A[1] = 30 \neq 18$
- $i = 2 : A[2] = 6 \neq 18$
- $i = 3 : A[3] = 18$

Return 3

Partial Correctness

Loop Invariant: If the loop body is exectuted j or more times, then after j executions of the loop body

- $i = j$
- $0 \leq i \leq n$
- $A[h] \neq k$ for $0 \leq h < i$

Proving the Loop Invariant: use induction on j

Base Case ($j = 0$):

- before first execution of loop body we have $i = 0$
- loop invariant holds (conditions on i, no values of h such that $0 \leq h < 0$)

Partial Correctness (cont.)

Inductive hypothesis: assume that, if the loop iterates j times, then the loop invariant holds for $i_{old} = j$

Need to show that if the loop iterates a $j + 1$st time, then the loop invariant holds for $i_{new} = j + 1$:

- if true for iteration $i_{old} = j$, then $A[h] \neq k$ for $0 \leq h < i_{old}$
  - if loop iterates, then $A[i_{old}] \neq k$ and $i_{new} = i_{old} + 1$
  - thus $A[h] \neq k$ for $0 \leq h < i_{new}$
  - because the loop iterated for $i_{old} = j$, we have $i_{old} < n$ and $i_{new} \leq n$.
- Thus, the loop invariant holds for $j + 1$.

Partial Correctness (applying the loop invariant)

When the loop test fails, the loop invariant holds **and** either $i \geq n$ or $A[i] = k$

- Case 1 ($i \geq n$): loop invariant implies that $A[h] \neq k$ for $0 \leq h < n$, so k is not in A and `KeyNotFoundException` is thrown
- Case 2 ($i < n$): loop invariant implies that $A[i] = k$ and i is returned

Conclusion:

- postcondition is satisfied in either case, so **linearSearch** is paritally correct

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 after each loop body execution
- $f(n,i) = 0$ when $i = n$, loop terminates (worst case) when $i \geq n$

Application of Loop Variant:

- existence demonstrates termination
- worst-case number of iterations is $f(n,0) = n$
- loop body runs in constant time, so worst-case runtime of **LinearSearch** is $\Theta(n)$

New Problem: Searching in a Sorted Array

Precondition:

A: Array of length n, for some integer $n \geq 1$, storing objects of some ordered type
New Requirement: $A[i] \leq A[i+1]$ for $0 \leq i < n-1$

k: An object of the type that might be found in A

Postcondition:

i: An integer such that $0 \leq i < n$ and $A[i] = k$
(The array A and key k were not changed.)

Exceptions:

`KeyNotFoundException`: Thrown if k is not found in A

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

LinearSearch(k)
$i = 0$
while $(i < n)$ **and** $(A[i] < k)$ **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 :

- $i = 0 : A[0] = -3 < 17$
- $i = 1 : A[1] = 2 < 17$
- $i = 2 : A[2] = 6 < 17$
- $i = 3 : A[3] = 18 \geq 17$

Throw `KeyNotFoundException`

Partial Correctness

Loop Invariant: If the loop body is exectuted j or more times, then after j executions of the loop body

- $i = j$
- $0 \leq i \leq n$
- $A[h] < k$ for $0 \leq h < i$

Proving the Loop Invariant: use induction on j

Base Case ($j = 0$):

- before first execution of loop body we have $i = 0$
- loop invariant holds (conditions on i, no values of h such that $0 \leq h < 0$)

Partial Correctness (cont.)

Inductive hypothesis: assume that, if the loop iterates j times, then the loop invariant holds for $i_{old} = j$

Need to show that if the loop iterates a $j + 1$st time, then the loop invariant holds for $i_{new} = j + 1$:

- if true for iteration $i_{old} = j$, then $A[h] < k$ for $0 \leq h < i_{old}$
  - if loop iterates without terminating, $A[i_{old}] < k$ and $i_{new} = i_{old} + 1$
  - thus $A[h] < k$ for $0 \leq h < i_{new}$
  - because the loop iterated for $i_{old} = j$, we have $i_{old} < n$ and $i_{new} \leq n$.
- Thus, the loop invariant holds for $j + 1$.

Partial Correctness (applying the loop invariant)

When the loop test fails, the loop invariant holds **and** either $i \geq n$ or $A[i] \geq k$

- Case 1 ($i \geq n$): loop invariant implies that $A[h] < k$ for $0 \leq h < n$, so k is not in A and `KeyNotFoundException` is thrown
- Case 2 ($i < n$ and $A[i] = k$): key is found and i is returned
- Case 3 ($i < n$ and $A[i] > k$): loop invariant implies that $A[i] < k$ for $0 \leq h < i$, so k is not in A and `KeyNotFoundException` is thrown

Conclusion:

- postcondition is satisfied in all cases, so **linearSearch** is paritally correct

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 k to $A[i]$

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

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

- can eliminate half of the array after the comparison

binarySearch(k)
  return bsearch($0, n-1, k$)

Specification of Requirements for Subroutine

Calling Sequence: bsearch(low, $high$, k)

Precondition:

A, k: Same for for "Searching in a Sorted Array"

low, $high$: Integers such that

- $0 \leq low \leq n$ and $-1 \leq high \leq n-1$
- $low \leq high + 1$
- $A[h] < k$ for $0 \leq h < low$
- $A[h] > k$ for $high < h \leq n-1$

Postcondition and Exceptions:

Same as for "Searching in a Sorted Array"

Pseudocode: The Binary Search Subroutine

bsearch(low, $high$, k)
  if $low > high$ **then**
    Throw `KeyNotFoundException`
  else
    $mid = \lfloor (low + high)/2 \rfloor$
    if $(A[mid] > k)$ **then**
      return bsearch(low, $mid - 1$, k)
    else if $(A[mid] < k)$ **then**
      return bsearch($mid + 1$, $high$, k)
    else
      return mid
    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 :

- **bsearch**(0,10,18): $mid = (0+10)/2 = 5$, $A[5] = 23 > 18$
- **bsearch**(0,4,18): $mid = (0+4)/2 = 2$, $A[2] = 6 < 18$
- **bsearch**(3,4,18): $mid = (3+4)/2 = 3$, $A[3] = 18$

Return 3

Partial Correctness

Assumptions

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

Case: $low > high$

- base case (no elements) — throw `KeyNotFoundException` (correct)

Case: $low = high$

- return $mid (= low = high)$ if $A[mid] = k$ (correct)
- otherwise recursive call with $low > high$ (correct)

Case: $low < high$

- return mid if $A[mid] = k$ (correct)
- recursive call (correct by assumption)

Efficiency

Case: $low \geq high$

- $\Theta(1)$ steps

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

- *Result of Function Call:*
  - i decreases by 1 (because new $high - low + 1$ is less than half the old value)
- *What Happens if* $i = 0$:
  - $high = low$ (algorithm terminates after at most one more iteration)
- *Initial Value:*
  - $i = \log_2 n$, because $low = 0$ and $high = n - 1$ initially
- *Conclusion:*
  - worst-case run time is $\Theta(\log_2 n)$ (constant number of steps per iteration)

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 7.1: design of recursive algorithms
- Section 7.3: discussion of recursive array search algorithms (linear and binary), Java implementation