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
```


Correctness and Efficiency

Correctness: covered in Tutorial 2

Efficiency:

- worst-case number of iterations is n
- loop body runs in constant time
- so worst-case runtime of **LinearSearch** is in $\Theta(n)$

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** $(A[i] < k)$ **do**
    $i = i + 1$
  end while
  if $(i < n)$ **and** $(A[i] = k)$ **then**
    return i
  else
    throw `KeyNotFoundException`
  end if

Correctness and Efficiency

Correctness: similar to unsorted case. Loop Invariant:

- 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

Efficiency: also $\Theta(n)$ in the worst case

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 key)
  if low > high then
    throw KeyNotFoundException
  else
    mid = ⌊(low + high)/2⌋
    if (A[mid] > key) then
      return  bsearch(low, mid − 1, key)
    else if (A[mid] < key) then
      return  bsearch(mid + 1, high, key)
    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

Induction on the length $n = high - low + 1$ of the subarray $A[low], \ldots, A[high]$

Inductive Hypothesis: Calls to **bsearch** *within the code* (subarray length $< n$) behave as expected

Base Case: $low > high$ $(n = 0)$

- no elements — throw `KeyNotFoundException` (correct)

Inductive Step: $low \leq high$ $(n > 0)$

- return mid if $A[mid] = key$ (correct)
- recursive call (correct by assumption). Should verify that:
  - preconditions of **bsearch** are satisfied for the recursive call
  - size of subarray in recursive call is $< n$

Efficiency and Termination

To search in array of size n:

1. if n is odd: recursively search subarrays of size $\frac{n-1}{2}$
2. if n is even: recursively search subarrays of sizes $\frac{n}{2} - 1$ and $\frac{n}{2}$

Summary: largest subarray is of size $\lfloor \frac{n}{2} \rfloor$

Efficiency and Termination, Cont.

$T(n)$: number of steps to search in array of size n

$$T(n) \leq \begin{cases} c_1 & \text{if } n = 0 \\ c_2 + T(\lfloor \frac{n}{2} \rfloor) & \text{if } n \geq 1 \end{cases}$$

for some constants $c_2 > c_1 > 0$.

Expand the recurrence relation:

$$\begin{aligned} T(n) &\leq c_2 + (c_2 + T(\lfloor \frac{n}{2^2} \rfloor)) \\ &= 2c_2 + T(\lfloor \frac{n}{2^2} \rfloor) \\ &\leq \cdots \\ &\leq kc_2 + T(\lfloor \frac{n}{2^k} \rfloor) \end{aligned}$$

Efficiency and Termination, Cont.

$T(n)$: number of steps to search in array of size n

- Recursion until $\lfloor \frac{n}{2^k} \rfloor = 0 \implies k = \lfloor \log_2 n + 1 \rfloor$
- Therefore, $T(n) \leq c_2 \lfloor \log_2 n + 1 \rfloor + c_1$

Can be shown that $T(n) \geq c \log_2 n$

- searching for an element greater (smaller) than the largest (smallest) element in the array

Conclusion: $T(n) \in \Theta(\log_2 n)$

A Note on the Analysis

When analyzing algorithms, sometimes we encounter the operators $\lfloor\rfloor$ and $\lceil\rceil$

- In general, these operators do not change the asymptotic running time of algorithms
- We usually ignore them, e.g., as if n was a complete power of 2 (will be more formally justified in CPSC 413)

Binary Search Algorithm:

- $T(n) \leq kc_2 + T(\frac{n}{2^k})$
- Therefore, $k = \log_2 n + 1 \implies T(n) \leq c_2(\log_2 n + 1) + c_1$

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

Further Reading and Java Code:

Data Structures & Algorithms in Java, Chapter 6