Computer Science 331
Algorithms for Searching

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #15

Outline

1. Searching in an Unsorted Array
   - The Searching Problem
   - Linear Search

2. Searching in a Sorted Array
   - The Searching Problem
   - Linear Search
   - Binary Search

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** $(A[i] \neq key)$ **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 iterations is n
loop body runs in constant time
So worst case runtime is Theta of 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** **do**
$\quad i = i + 1$
end while
if $(i < n)$ **and** $(A[i] = k)$ **then**
$\quad$ **return** i
else
$\quad$ throw KeyNotFoundException
end if

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 $0 \leq i < n$
- $A[j] < key$ for $0 \leq j \leq i$
- A and *key* have not been changed

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

Base Case:
- before first execution of loop body, we have i=0
- loop test passes, implying that A[0]<key
- A and key have not been changed

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:
- 0<=i<=n
- A[j] < key for 0 <= j < i
- A and key have not changed

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 (so $i < n$ and $A[i] < key$), implying that immediately before the $i + 1$st execution:
- 0<=i<n
- A[j]<key for 0 <=j<=i
- A and key have not changed

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 $0 \leq i \leq n$
- $A[j] < key$ for $0 \leq j < i$
- 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:

- $f(n, i)$ 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:

- Existence demonstrates termination
- Worst case number of iteration is f(n,0) = n
- Loop body runs in constant time, so worst case runtime of LinearSearch is Theta n

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

    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

*Note: Recursive Algorithms require induction

	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 :

- Midpoint is 5
- Midpoint is 2
- 18 reached

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:

*preconditions of bsearch are satisfied for 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

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:

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

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:

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

Data Structures: Abstraction and Design Using Java

- by Elliot B. Koffman and Paul A. T. Wolfgang
- Section 5.3