Binary Search

MARCH 9TH, 2015

The Search Problem

- One of the most common computational problems (along with *sorting*) is *searching*.

- In its simplest form, the input to the search problem is a list `L` and an item `k` and we are asked if `k` belongs to `L`. (The `in` operator in Python.)

- In a common variant, we might be asked for the index of `k` in `L`, if `k` does belong to `L`. (The `L.index()` method in Python.)

Searching lists

- Python provides several built-in operations for searching lists:
  - `elem in L`: evaluates to True if `elem` is in list `L`
  - `L.index(elem)`: returns the index of the first occurrence of `elem` in `L`; is an error if `elem` is not in `L`.
  - `L.count(elem)`: returns the number of occurrences of `elem` in `L`.

- Other related operations:
  - `min(L)`, `max(L)`: these return the minimum element and maximum element respectively of L.

Linear Search

- If we don't know anything about L, then the only way to solve the problem is by scanning the list L completely in some systematic manner.
- This takes time proportional to the size of the list, in the *worst case*.
- And for this reason, this is called *linear search*.
- Linear search can be quite inefficient for many applications because search is such a common operation in programs.
- The Python search operations mentioned in the previous slide all perform linear search because they are expected to work on any list.

Binary Search

- If the list L is known to be *sorted* (in ascending or descending order), then we can use a much more efficient algorithm called *binary search*.
- Binary search is so much more efficient than linear search that it provides a significant incentive to keep lists sorted.
- More on the efficiency of binary search later.

Binary Search Algorithm

- Suppose that L is sorted in ascending order.
- Compare k with the middle element of L.
  - If k == L[middle], we are done
  - If k < L[middle], we need to search the first half of L
  - If k > L[middle], we need to search the second half of L
- Notice that after one comparison, the size of the problem shrinks to 1/2 of what it was earlier.
- (Compare this with linear search where after one comparison, the problem size reduced by just 1 element.)

Binary Search Alorithm (more details)

- Explicitly maintain two indices `left` and `right`.
- The sublist `L[left..right]` (inclusive) is what still remains to be searched.
- Initially, `left` is 0 and `right` is `len(L)-1`.
- Since we are interested in comparing `k` with the "middle" element, we maintain a third index called `mid` (set to `(left + right)/2`).
- After one comparison, either we find `k` or we look for it in the left half (`right = mid -1`) or in the right half (`left = mid + 1`).

The function binarySearch

```
def binarySearch(L, k):
    left = 0
    right = len(L)-1

    # iterate while there is a sublist that needs to be searched
    while left <= right:
        mid = (left + right)/2 # index of the middle element

        # Comparisons and then adjusting the boundaries of
        # the sublist, if necessary
        if L[mid] == k:
            return mid # element is found at mid, so return this index
        elif L[mid] < k: # look for element in right half
            left = mid + 1
        elif L[mid] > k: # look for element in the left half
            right = mid -1

    return -1 # element is not found in the list
```

Execution Examples

binarySearch([1, 4, 11, 24, 24, 56, 60, 70], 65)

Slices searched:

0 7
4 7
6 7
7 7
Not found

binarySearch([1, 4, 11, 24, 24, 56, 60, 70], 4)

Slices searched:

0 7
0 2
Found

Worst Case Running Time

- Assume the worst case, i.e., we don't find k.
- After each comparison of k with L[mid] the problem size shrinks to ½ of what it was before the current iteration.

Problem Size	Number Iterations Completed
N	0
N/2	1
$N/2^2$	2
$N/2^3$	3

Worst Case Running Time (contd.)

- Thus after t iterations have been completed, the problem size has shrunk to $N/2^t$.
- Therefore, for the problem size to shrink to 1, we need

$$N = 2^t$$

$$\Rightarrow t = \log_2 N$$

- Thus the worst case running time of binary search is logarithmic in the size of the list.

Example that shows the speed of Binary Search

- **Problem:** If we sample N times uniformly at random from the integers {1, 2, 3,…, N}, how many distinct elements will we get?
- Statisticians are interested in these kinds of questions.
- It is easy to write a simple Python program to get a sense of this.

Code using slow search

```
import random

L = []
for i in range(50000):
    L.append(random.randint(1,50000))

count = 0
for e in range(1, 50001):
    if e in L:
        count = count + 1


print count
```

Output

Time to build list is 0.129420042038

31733

Time to count distinct elements is 45.7874200344

Faster Code using Binary Search

```
import random
from binarySearch import *

L = []
for i in range(50000):
    L.append(random.randint(1,50000))

L.sort()

count = 0
for e in range(1, 50001):
    if binarySearch(L, e) >= 0:
        count = count + 1
```

Output

Time to build list is 0.125706195831

Time to sort list is 0.0273258686066

31717

Time to count distinct elements is 0.3523209095