15-104 Introduction to Computing for Creative Practice

Fall 2020

19 Linear Search (and Algorithmic Analysis)

Instructor: Tom Cortina, tcortina@cs.cmu.edu, GHC 4117, 412-268-3514

1

indexOf

- Suppose you have an array of elements and you want to find the location (index) of some particular item?
- We can use the indexOf method on an array.
  `arr.indexOf(searchElement [, fromIndex] )`
  Search the array *arr* for *searchElement* starting at index *fromIndex* (or 0 if not specified). Returns the index of first exact match if found, or -1 otherwise.
- Example:

```
var cars = ['Pontiac', 'Olds', 'Cadillac', 'Buick', 'Chevrolet'];
print(cars.indexOf('Cadillac'));        prints 2
print(cars.indexOf('Ford'));            prints -1
print(cars.indexOf('Buick', 1));        prints 3
print(cars.indexOf('Pontiac', 3));      prints -1
```

2

Linear search

- How does `indexOf` work? It performs a linear search of the elements until a match is found or the end of the array is encountered, whichever comes first.
- How would we implement this ourselves?

```
function lin_search(arr, element, index) {
    if (index < 0 || index >= arr.length) return -1;    // Why?
    var i = index;
    while (i < arr.length) {
        if (arr[i] == element) return i;
        i++;
    }
    return -1;
}
```

3

Linear search (another way)

```
function lin_search(arr, element, index) {
    if (index < 0 || index >= arr.length) return -1;
    for (var i = index; i < arr.length; i++) {
        if (arr[i] == element) return i;
    }
    return -1;
}
```

Note that `i` is a local variable, declared for the `for` loop only.
This will be important in the next example.

4

Linear search (yet another way)

```
function lin_search(arr, element, index) {
    if (index < 0 || index >= arr.length) return -1;    // Why?
    var i = index;
    while (i < arr.length && arr[i] != element) {
        i++;
    }
    if (i < arr.length) {
        return i;
    }
    return -1;                // else is not needed, why?
}
```

If you write this as a `for` loop, you must declare `i` using `var` before the loop starts, otherwise `i` is a local variable that can only be used in the `for` loop.

5

Find the index of the maximum

Suppose you want to find the location of the maximum in a non-empty array of numbers.

```
function find_index_of_max(arr) {
    var max_index = 0;
    var i = 1;
    while (i < arr.length) {
        if (arr[i] > arr[max_index]) {
            max_index = i;
        }
        i++;
    }
    return max_index;
}
```

6

Trace

```
function find_index_of_max(arr) {
    var max_index = 0;
    var i = 1;
    while (i < arr.length) {
        if (arr[i] > arr[max_index]) {
            max_index = i;
        }
        i++;
    }
    return max_index;
}
arr =        [42, 17, 56, 35, 71, 80, 22, 50, 39, 68]
i                 1   2   3   4   5   6   7   8   9
max_index     0   0   2   2   4   5   5   5   5   5 ← answer
```

Why is `arr[i] > arr[i-1]` wrong?

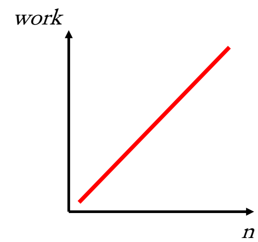
7

Algorithmic Analysis

- Suppose you performed linear search on an array of n elements.
- What is the worst case for this search?
- How many elements are examined in the worst case?
- If you double the number of elements in the original array, how elements would be examined in linear search on the array in the worst case?
- If you triple the number of elements in the original array, how elements would be examined in linear search on the array in the worst case?
- If you quadruple the number of elements in the original array, how elements would be examined in linear search on the array in the worst case?

8

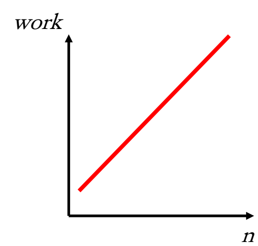
Measuring the work

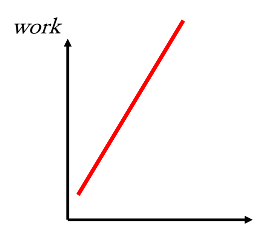


- We don't measure algorithmic complexity by measuring its running time since processors vary, compilers optimize code differently, etc.
- Instead, we look at how much "work" the algorithm does.
- For searching, the work is related the number of elements examined. (For sorting, the work is often the number of elements compared to each other.)
- If we plot the number of elements vs. the number of elements examined (work) for linear search, we see a straight line (linear).
- There is additional work in linear search (e.g. controlling the loop, executing the return, etc.), but as the number of elements grows, the dominant computation is the examination of elements.

9

Measuring the work

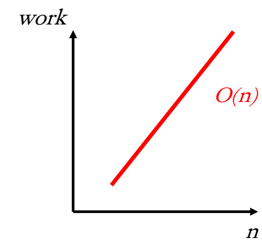




- What if we included not just examinations (comparisons) but also the process of returning the answer?
- In the worst case, linear search would take n+1 operations but if we plotted this as a function of n, it's still a straight line.
- What if we looked at each element twice (for some reason) and then returned the answer? It would take 2n+1 operations but this is still linear.
- What matters here is that the relationship between the number of elements and the amount of work is linear. Put another way: *The amount of work is linearly proportional to the number of elements.*

10

Big O



- Computer scientists analyze algorithms to determine how they will scale for large quantities of data.
- For linear search, no matter how we count the operations, the relationship is linear, a function of n^1.
- So we express this using **big O notation**, which indicates to what class of computations this algorithms belongs.
- We say linear search is **O(n)** in the worst case.
  - All algorithms in this class do an amount of work linearly proportional to the number of data values (n).
  - If an algorithm is O(n), then if we double the number of inputs/elements, then we can expect twice as much work, approximately.

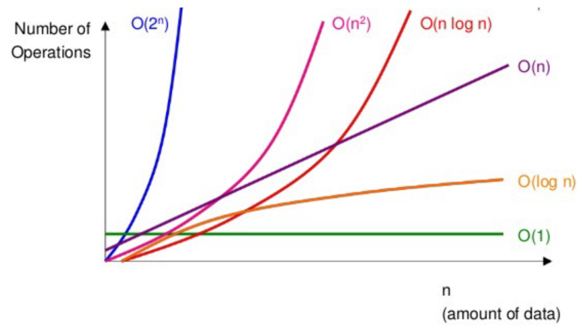
11

Other algorithms

• Linear search – average case	O(n)
• Linear search – best case	O(1)
• Binary search – worst case (array must be sorted)	O(log n)
• Sorting using "selection sort" – worst case	$O(n^2)$
• Sorting using "merge sort" – worse case	O(n log n)
• Computing the truth table for Boolean functions of n variables	$O(2^n)$
• Computing the cost of every route for a traveling salesperson who visits n cities.	O(n!)

12

Comparing Algorithms



When *n* is small, the algorithm you pick doesn't really matter. But when *n* is large, it matters!

13

Algorithmic Analysis

- Knowing how your algorithms will perform computationally is an important skill for anyone who will develop software.
- What if you had n = 1 million elements and each element required 1 microsecond to examine?
- Worst case (rough approximations since algorithms are expressed using big O):
  - Binary Search — 20 microseconds (but the array must be sorted!)
  - Linear Search — 1 second
  - Merge Sort — 20 seconds
  - Selection Sort — 11.5 days (it's actually less, but still days)
  - Traveling Salesperson — $3 * 10^{144}$ years, approximately (!?!?!)
- Be careful what you code… it might run a long, long time!!!

14