Algorithms

WTP 2009
Computer Science
Day 05: Friday, July 2

What is an algorithm?

A finite set of precise instructions for accomplishing a certain task

- Important characteristics:
  - well-defined: each step is precise (no ambiguity)
  - correct: must produce the correct/desired output value(s)
  - finite: the algorithm runs in finite time
  - general: applicable to all problems of a similar form

What do we use algorithms for?

- Some example algorithms:
  - Searching
    - Finding if a certain name is in a list of names.
  - Sorting
    - Arranging a list words in alphabetical order
  - Cryptography
    - Sending data safely across the web
  - Data Compression
    - Fitting as many pictures as possible onto a memory card
  - Graph problems
    - Finding the quickest way from New York to Boston

Activity

Let's see who's the tallest in a group!

Algorithm descriptions

- English

  max(my_list): "scan each element of my_list from left to right, keeping track of the largest element you've seen so far and return this at the end"

- Python code

```
def max(my_list):
   max_elt = my_list[0]
   for x in my_list:
      if x > max_elt:
          max_elt = x
   return max_elt
```

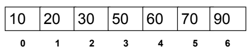

Search

- Problem: want to find a particular element in a sequence
- Simplest search algorithm: linear search
- Python code:

```
def linear_search(my_list, val)
  for e in my_list:
     if e == val:
        return list.index(e)
   return None
```

Binary Search

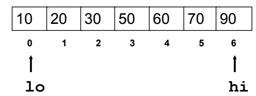
- What if your data is sorted? Can we do better than linear search?



- Let's look for the value 60

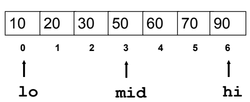
Binary Search

- Invariant. Algorithm maintains
  - **my_list[lo] ≤ val ≤ my_list[hi].**



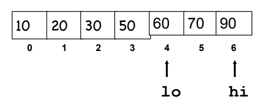
Binary Search

- Invariant. Algorithm maintains
  - **my_list[lo] ≤ val ≤ my_list[hi].**



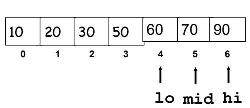
Binary Search

- Invariant. Algorithm maintains
  - **my_list[lo] ≤ val ≤ my_list[hi].**



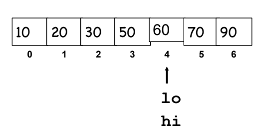
Binary Search

- Invariant. Algorithm maintains
  - **my_list[lo] ≤ val ≤ my_list[hi].**



Binary Search

- Invariant. Algorithm maintains
  - **my_list[lo] ≤ val ≤ my_list[hi].**



Binary Search

- Invariant. Algorithm maintains
  - **my_list[lo] ≤ val ≤ my_list[hi].**

10	20	30	50	60	70	90
0	1	2	3	4	5	6
				↑		
				lo		
				hi		
				mid		

Binary Search

- Invariant. Algorithm maintains
  - **my_list[lo] ≤ val ≤ my_list[hi].**

10	20	30	50	60	70	90
0	1	2	3	4	5	6
				↑		
				lo		
				hi		
				mid		

Binary Search

- Reduce search space by half each time by comparing to midpoint value
- Python code

```
def search(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:       # there is still a range to search
        mid = (low + high)/2 # position of the middle item
        item = nums[mid]
        if x == item:        # found it! return the index
            return mid
        elif x < item:       # x is in lower half of the range
            high = mid - 1
        else:                # x is in upper half of the range
            low = mid + 1
        return -1
```

Algorithm Analysis

- How can you determine if an algorithm is efficient or not?
  - Amount of time it takes to run?
  - The use of a shared resource?

The actual amount of time that an algorithm takes to run (measured in seconds, minutes, hours and millennia) depends on the details of the computing machine!

Algorithm Analysis

- We need a measure that is independent of the specific machine!

The measure we use in computer science is the number of basic operations required to complete the algorithm.

Algorithm Analysis

If we are using linear search to check whether the word “awesome” appears in a sentence that is eight words long then:

1. What is the best-case input?

2. What is the worst-case input?

Algorithm Analysis

- Big O notation
  - Describes how the time (or space) an algorithm takes up increases as the size of the input (n) grows.

Algorithm Analysis

- How many steps are necessary to find what we are looking for in the **linear search** algorithm?

```
def linear_search(my_list, val)
  for e in my_list:
     if e == val:
        return list.index(e)
   return None
```


Linear Search Performance

10	40	30	50

- How many comparisons?
  - If the size of the list is N:
    - At most N comparisons
  - Worst case running time O(N)

Binary Search Performance

- Each try, halve the search space
- How many times can we halve n numbers until there is only one remaining?

Binary Search Performance

10	20	30	50	60	70	90
0	1	2	3	4	5	6

List Size	Halvings
1	0
2	1
4	2
8	3
16	4
32	5
1,000,000	20

Exercise

- Sketch the graphs for f(n) = log(n) and g(n) = n for n, where $0 \le n \le 10$

Exercise

– Sketch the graphs for $g(n) = n$ and $h(n) = n^2$ for n, where $0 \le n \le 10$

Which search algorithm is better?

- Linear search requires **n** comparisons
- Binary Search requires **$\log_2(n)$** comparisons

- But binary search requires the list to be sorted ...

Sorting

- A sorting algorithm…
  - arranges the elements of an input list in a certain order
  - helps make other algorithms run more efficiently and data more readable for humans
  - outputs a permutation (reordering) of the input
  - no elements lost or added
- Sorting orders:
  - Numerical (1 < 2 < 3 ...)
  - Alphabetical (a < ab < b ...)

Insertion Sort

- Works like inserting a new card into a partially sorted hand by bubbling to the left into the sorted sublist
- Demo: http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/insert/insertionen.htm

Insertion Sort

- For each unsorted element of the list, walk backwards (forwards) through the sorted part until you find a smaller (larger) element; insert at this point
- Python code:

```
def insertion_sort(a):
    for i in range(1, len(a)-1):
            j = i
            t = a[i]
            j = i – 1
            while j > 0 and a[j-1] > t:
                a[j] = my_list[j-1]  # shift right
                j -= 1
               a[j] = t                  # insert
```


Analysis of sorting

- Insertion sort
  - For each element, up to n (technically, n-1) comparisons
  - Total of n elements
  - Worst-case run time: $O(n^2)$

Other sorting algorithms

- Selection sort: Scan the list multiple times and select the lowest element
- Bubble sort: continually swap values, so that lower elements "bubble" up the list
- Mergesort: divide and conquer!
- Quicksort: randomized version of merge sort

Sorting Demo

See:
http://www.sorting-algorithms.com/

Classes of Algorithms

- Linear Algorithms : O(n)
  - Linear in the size of the input
- Logarithmic Algorithms : O(log n)
  - Reduces the size of the input by a constant factor > 1
- Quadratic Algorithms : $O(n^2)$
  - Have nested loops

Sorting before searching?

- Can we sort in <u>sub-linear</u> time?
- Can we sort in <u>linear</u> time?
- Okay, how fast we can sort?

Total time for Linear Search and Binary Search

- Linear Search = O(n)
- Binary Search = O(n logn) + O(logn)

- Does that mean linear search is better than binary search?

  Depends what you are searching for

Total time for Linear Search and Binary Search

- Suppose you are looking for **k** elements
- Linear Search = O(k n)
- Binary Search = O(n logn) + O(k logn)

For very large k, we see that binary search performs better.

Today's exercises

- Identifying algorithms
- Practice analyzing algorithms
- Using sorting
- Readings for Monday: Chapter 8.6-10, 10.6-9, 11.1-4