Introduction to Programming

Spring 2022

Algorithm Design and Recursion

- **Searching**
- Recursive Problem Solving
- Sorting Algorithms
- Hard Problems

Searching

- Searching is the process of looking for a particular value in a collection.
- For example, a program that maintains a membership list for a club might need to look up information for a particular member – this involves some sort of search process.

A Simple Searching Problem

- Here is the specification of a simple searching function:

```
def search(x, nums):
    # nums is a list of numbers and x is a number
    # Returns the position in the list where x occurs
    # or -1 if x is not in the list.
```

- Here are some sample interactions:

```
>>> search(4, [3, 1, 4, 2, 5])
2
>>> search(7, [3, 1, 4, 2, 5])
-1
```

A Simple Searching Problem

- In the first example, the function returns the index where 4 appears in the list.
- In the second example, the return value -1 indicates that 7 is not in the list.
- Python includes a number of built-in search-related methods!

A Simple Searching Problem

- We can test to see if a value appears in a sequence using in.

```
if x in nums:
    # do something
```

- If we want to know the position of x in a list, the index method can be used.

```
>>> nums = [3, 1, 4, 2, 5]
>>> nums.index(4)
2
```

A Simple Searching Problem

- The only difference between our search function and index is that index raises an exception if the target value does not appear in the list.
- We could implement search using index by simply catching the exception and returning -1 for that case.

A Simple Searching Problem

```
def search(x, nums):
    try:
        return nums.index(x)
    except:
        return -1
```

- Sure, this will work, but we are really interested in the algorithm Python uses to search the list!

Strategy 1: Linear Search

- Pretend you're the computer, and you were given a page full of randomly ordered numbers and were asked whether 13 was in the list.
- How would you do it?
- Would you start at the top of the list, scanning downward, comparing each number to 13? If you saw it, you could tell me it was in the list. If you had scanned the whole list and not seen it, you could tell me it wasn't there.

Strategy 1: Linear Search

- This strategy is called a linear search, where you search through the list of items one by one until the target value is found.

```
def search(x, nums):
    for i in range(len(nums)):
        if nums[i] == x:
            return i

    return -1
```

- This algorithm wasn't hard to develop, and works well for modestly-sized lists.

Strategy 1: Linear Search

- The Python `in` and `index` operations both implement linear searching algorithms.
- I have to look at each data value till:
  - I find what I am looking for or
  - I reach end of the list
- If the collection of data is very large, it makes sense to organize the data somehow so that each data value doesn't need to be examined.

Strategy 1: Linear Search

- If the data is sorted in ascending order (lowest to highest), we can skip checking some of the data.
- As soon as a value is encountered that is greater than the target value, the linear search can be stopped without looking at the rest of the data.
- On average, this will save us about half the work.

Strategy 2: Binary Search

- If the data is sorted, there is an even better searching strategy
  - one you probably already know!
- Have you ever played the number guessing game?
  - where I pick a number between 1 and 100 and you try to guess it?
  - Each time you guess, I'll tell you whether your guess is correct, too high, or too low.
  - What strategy do you use?

Strategy 2: Binary Search

- Young children might simply guess numbers at random.
- Older children may be more systematic, using a linear search of 1, 2, 3, 4, … until the value is found.
- Most adults will first guess 50. If told the value is higher, it is in the range 51-100. The next logical guess is 75.

Strategy 2: Binary Search

- Each time we guess the middle of the remaining numbers to try to narrow down the range.
- This strategy is called binary search.
- Binary means two, and at each step we are dividing the remaining group of numbers into two parts.

Strategy 2: Binary Search

- We can use the same approach in our binary search algorithm! We can use two variables to keep track of the endpoints of the range in the sorted list where the number could be.
- Since the target could be anywhere in the list, initially low is set to the first location in the list, and high is set to the last.

Strategy 2: Binary Search

- The heart of the algorithm is a loop that looks at the middle element of the range, comparing it to the value x.
- If x is smaller than the middle item, high is moved so that the search is confined to the lower half.
- If x is larger than the middle item, low is moved to narrow the search to the upper half.

Strategy 2: Binary Search

- The loop terminates when either
  - x is found
  - There are no more places to look (low > high)

Strategy 2: Binary Search

```
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 middle item
        item = nums[mid]
        if x == item:          # Found it! Return the index
            return mid
        elif x < item:         # x is in lower half of range
            high = mid - 1     #  move top marker down
        else:                  # x is in upper half of range
            low = mid + 1      #  move bottom marker up
    return -1                  # No range left to search,
                               # x is not there
```

Comparing Algorithms

- Which search algorithm is better, linear or binary?
  - The linear search is easier to understand and implement
  - The binary search is more efficient since it doesn't need to look at each element in the list
- Intuitively, we might expect the linear search to work better for small lists, and binary search for longer lists. But how can we be sure?

Comparing Algorithms

- One way to conduct the test would be to code up the algorithms and try them on varying sized lists, noting the runtime.
  - Linear search is generally faster for lists of length 10 or less
  - There was little difference for lists of 10-1000
  - Binary search is best for 1000+ (for one million list elements, binary search averaged .0003 seconds while linear search averaged 2.5 seconds)

Comparing Algorithms

- While interesting, can we guarantee that these empirical results are not dependent on the type of computer they were conducted on, the amount of memory in the computer, the speed of the computer, etc.?
- We could abstractly reason about the algorithms to determine how efficient they are. We can assume that the algorithm with the fewest number of "steps" is more efficient.

Comparing Algorithms

- How do we count the number of “steps”?
- Computer scientists attack these problems by analyzing the number of steps that an algorithm will take relative to the size or difficulty of the specific problem instance being solved.

Comparing Algorithms

- For searching, the difficulty is determined by the size of the collection – it takes more steps to find a number in a collection of a million numbers than it does in a collection of 10 numbers.
- How many steps are needed to find a value in a list of size n?
- In particular, what happens as n gets very large?

Comparing Algorithms

- Let's consider linear search.
  - For a list of 10 items, the most work we might have to do is to look at each item in turn – looping at most 10 times.
  - For a list twice as large, we would loop at most 20 times.
  - For a list three times as large, we would loop at most 30 times!
- The amount of time required is linearly related to the size of the list, n. This is what computer scientists call a linear time algorithm.

Comparing Algorithms

- Now, let's consider binary search.
  - Suppose the list has 16 items. Each time through the loop, half the items are removed. After one loop, 8 items remain.
  - After two loops, 4 items remain.
  - After three loops, 2 items remain
  - After four loops, 1 item remains.
  - Needs at most 4 loops for 16 items
- If a binary search loops i times, it can find a single value in a list of size 2^i.

Comparing Algorithms

- To determine how many items are examined in a list of size n, we need to solve $n = 2^i$ for i

$$i = \log_2 n$$

- Binary search is an example of a log time algorithm – the amount of time it takes to solve one of these problems grows as the log of the problem size.
- If n = 12 million (12,000,000)
  - Linear search will take 12 million guesses
  - Binary search will take 24 guesses (only works if data is sorted)