Analysis of Algorithms

Announcements

- Lab 7 due Saturday at midnight
  - Run update21, use quotes.txt for testing
- Quiz 4 is on Friday
  - Review at ninja session tonight

Today's plan

- Go over quiz 4 topics
- Review linear search and binary search
- Algorithm analysis

Quiz 4

- Be able to understand, use, implement functions that:
    - Are called for their side effects (printing, getting input, mutating a list, etc.)
    - Are called for their return value
    - Are called because they have a useful side effect and a useful return value
- Be able to draw stack diagrams for programs that call such functions

Quiz 4

- Ways to mutate a list:
  - Index assignment
  - Methods like .append() that mutate
  - Existing functions like shuffle() that mutate
  - New, user-defined functions that do mutation using one of the above

Quiz 4

- Don’t focus on top-down design, except that you should be able to understand programs with multiple functions
- Don’t focus on file i/o

```python
def mystery(L):
  for i in range(len(L)):
    if L[i] % 2 == 0:
      L[i] = L[i]**2
  # DRAW STACK HERE

def main():
  myList = range(1,6)
  print(myList)
  mystery(myList)
  print(myList)

main()
```

Linear search

- Search task: determine if a value, ***`x`***, appears in a list, ***`L`***
- Algorithm: go through the items in ***`L`*** one at a time. If you find ***`x`***, return ***`True`***. If you get through all the items without finding ***`x`***, return ***`False`***.
- Worst-case run time proportional to length of list
- It's what the ***`in`*** operator does

Linear search

```
def linearSearch(x, L):
  """
  Purpose: determine if x appears in the list L
  Parameters: x - value we're searching for
              L - list that might contain x
  Returns: True if x is in L, False otherwise
  """
  for item in L:
    if x == item:
      return True
  return False
```

Linear search variation

```
def linearSearchIndex(x, L):
  """
  Purpose: determine the index at which x appears in L
  Parameters: x - value we're searching for
              L - list that might contain x
  Returns: index at which x appears in L or None if x
           does not appear in L
  """
  for i in range(len(L)):
    if L[i] == x:
      return i
  return None
```

Binary search

- Search task: determine if a value, ***`x`***, appears in a **sorted** list, ***`L`***

- Algorithm: keep track of the lowest and highest indices where ***`x`*** might appear (***`lo`*** and ***`hi`***). Repeatedly examine the value at the midpoint between ***`lo`*** and ***`hi`*** (***`mid`***), returning ***`True`*** if this value is equal to ***`x`*** or updating the range of possible indices if it is not. If this range of indices ever becomes empty, return ***`False`***.

- Worst-case run time proportional to logarithm of length of list

```python
def binarySearch(x, L):
  lo = 0
  hi = len(L)-1

  while lo <= hi:
    mid = (lo + hi)/2
    if L[mid] == x:
      return True
    elif L[mid] < x:
      lo = mid + 1
    else:
      hi = mid - 1

  return False
```

Trace binary search

- Show chart with values for ***`lo`***, ***`mid`***, and ***`hi`*** as they update in binary search algorithm

```
L = [ 'a', 'b', 'd', 'f', 'h', 'i', 'k', 'n', 'o', 'q', 'w', 'x', 'z']
#      0    1    2    3    4    5    6    7    8    9    10   11   12
binarySearch('f', L)
binarySearch('t', L)
```

Analysis of algorithms

- We have multiple algorithms for accomplishing the same task—how do we choose which one to use?
- Speed or **run time** is a big consideration; there are other considerations, like memory usage, simplicity, and generality.

Run time analysis

- Just use Python's built-in timer?
  - `linearSearch(5, [1, 3, 5])`: 5 microseconds
  - `binarySearch(5, range(1000000))`: 15 microseconds
- Let's make it a fair comparison:
  - `linearSearch(5, range(1000000))`: 8 microseconds
  - `binarySearch(5, range(1000000))`: 15 microseconds

Run-time analysis

- Ok, but let's consider the worst case:
  - `linearSearch(10000001, range(1000000))`: 40000 microseconds (or 40 **milli**seconds)
  - `binarySearch(10000001, range(1000000))`: 14 microseconds
- Ok, but computers vary in terms of speed. And other programs running at the same time will have an effect. And the speed of computers increases over time.

Run-time analysis

- Timing can be useful, but it's not the best way to compare algorithms. Instead we look for a mathematical function that equals the number of steps an algorithm takes in terms of the size of the input to the algorithm, ***n***. We typically start by considering the worst case.
  - linear: ***2*n + 1***, binary: ***4*log n + 3***
- Ok, but now it depends on the size of n. We typically look at what these functions do as n goes to infinity (like a limit from math) and take only the fastest-growing term, ignoring constant factors
  - linear: *O(n)*, binary: *O(log n)*

Final analysis

- Binary search is faster, but only works for sorted lists.
- Linear search is easier to implement and thus less likely to contain a bug. It works for any list. For small lists, the difference in run time isn’t noticeable.

Good luck studying!