EdX 6.00x Notes

Lecture 10:

- Algorithm and data structures
  - How do you find efficient algorithms?
    - Hard to invent new ones
    - Easier to reduce problems to known solutions
      - Understand inherent complexity of problem
      - Think about how to break problem into sub-problems
      - Relate sub-problems to other problems for which there already exist efficient algorithms
- Search algorithms
  - Search algorithm – method for finding an item or group of items which specific properties within a collection of items
  - Collection called the search space
  - Saw examples – finding square root as a search problem
    - Exhaustive enumeration
    - Bisection search
    - Newton-Raphson
- Linear search and indirection
  - Simple search method
  - Complexity?
    - If element not in list, O(len(L)) tests
    - So at best linear in lenth of L
    - Why "at best linear"
      - Assumes each test in loop can be done in constant time
      - But does Python retrieve the i^{th} element of a list in constant time? (Yes)
- Indirection
  - Simple case: list of ints
    - Each element is of the same size (e.g., four units of memory – or four eight bit bytes)
    - Then address in memory of i^{th} element is start + 4 * I where start is address of start of list
    - So can get to that point in memory in constant time
  - But what if list of objects of arbitrary size?
  - Use indirection.
  - Represent a list as a combination of a length (number of objects), and a sequence of fixed size pointers to objects (or memory addresses)

        - Each element of the list is not going to be the element itself but a pointer to an object
    - If length field is 4 units of memory, and each pointer occupies 4 units of memory
    - Then address of i^{th} element is stored at start + 4 + 4 * i
    - This address can be found in constant time, and value stored at address also found in constant time so search is linear.
    - **Indirection** – accessing something by first accessing something else that contains a reference to thing sought
- Binary Search
    - Can we do better than O(log(L)) for search?
    - If we know nothing about values of elements in list, then no.
    - Worst case, we would have to look at every element
- What if list is ordered?
    - Suppose elements are sorted in ascending order
    - Doing a greater than comparison improves average complexity, but worst case still need to look at every element.
- Use binary search
    - Pick an index, I, that divides list in half
    - Ask if L[i] == e
    - If not, ask if L[i] larger or smaller than e
    - Depending on answer, search left or right half of L for e
    - A new version of divide-and-conquer algorithm
        - Break into smaller version of problem (smaller list), plus some simple operations.
- Analyzing binary search
    - Does recursion halt?
        - Decrementing function
            - Maps values to which formal parameters are bound to non-negative integer
            - When value <=, recursion terminates
            - For each recursive call, value of function is strictly less than value on entry to instance of function
        - Here function high – low
            - At least 0 first time called (1)
            - When exactly 0, no recursive call, returns (2)
            - Otherwise, halt or recursively call with value halved (3)
        - So terminates
    - What is the complexity?
        - How many recursive calls? (Work within each call is constant)
        - How many times can we divide high – low in half before reaches 0?
        - Log_2(high – low)

        - Thus search complexity is O(log(len(L)))
- Sorting algorithms
    - So what about cost of sorting?
    - Assume complexity of sorting a list is O(sort(L))
    - Then if we sort and search we want to know if sort(L) + log(len(L)) < len(L)
        - i.e. should we sort and search using binary, or just use linear search
- Amortizing costs
    - "Amortize" – spread out a big cost over a period of time
    - Considers entire sequence of operations
    - But suppose we want to search a list k times?
    - Then is sort(L) + k*log(len(L)) < k*len(L)
        - Depends on k, but one expects that if sort can be done efficiently, then it is better to sort first
        - Amortizing cost of sorting over multiple searches to make this worthwhile
        - How efficiently can we sort?
- Selection sort
    - Given a list, we're going to find the smallest element in the list and swap it with the first element.
    - Then take the remainder of the list, find the smallest element of that, and swap it with the second element.
    - Keep doing that until we've done the overall search.
- Analyzing selection sort
    - Loop invariant
        - Given prefix of list L[0:i] and suffix L[i+1:len(L)-1], then prefix is sorted and no element in prefix is larger than smallest element in suffix
            - Base case: prefix empty, suffix whole list – invariant true
            - Induction step: move minimum element from suffix to end of prefix. Since invariant true before move, prefix sorted after append
    - When exit, prefix is entire list, suffix empty, so sorted
    - Complexity of inner loop is O(len(L))
    - Complexity of outer loop is also O(len(L))
    - So overall complexity is $O(len(L)^2)$ or quadratic
    - Expensive!
- Merge Sort
    - Uses a divide and conquer approach:
        - If list of length 0 or 1, already sorted
        - If list has more than one element, split into two lists, and sort each
        - Merge results
            - To merge, just look at first element of each, move smaller to end of the result
            - When one list empty, just copy rest of other list

- Complexity of Merge Sort
  - Comparison and copying are constant
  - Number of comparisons – O(len(L))
  - Number of copyings – O(len(L1) + len(L2))
  - So merging is linear in length of the lists
  - Merge is O(len(L))
  - Mergesort is O(len(L)) * number of calls to merge
    - O(len(L)) * number of calls to merge sort
    - O(len(L)*log(len(L)))
  - Log linear – O(n log n), where n is len(L)
  - Does come with cost in space, as makes new copy of list
- Improving efficiency
  - Combining binary search with merge sort very efficient
    - If we search list k times, then efficiency is n*log(n) + k*log(n)
  - Can we do better?
  - Dictionaries use concept of hashing
    - Lookup can be done in almost independent of size of dictionary
- Hashing
  - Convert key to an int
  - Use int to index into a list(constant time)
  - Conversion done using a hash function
    - Map large space of inputs to smaller space of outputs
    - Thus a many-to-one mapping
    - When two inputs go to same output – a collision
  - Increasing size of hash table reduces collisions, however the tradeoff is it takes more space
  - A good hash function has a uniform distribution – minimizes probability of a collision
- Complexity
  - If no collisions, then O(1)
  - If everything is hashed to the same bucket, then O(n)
  - In general, can trade off space to make hash table large, and with good function get close to uniform distribution, and reduce complexity to close to O(1)
- Note:
  - An example of a good hash function is one that relies on relies on modular arithmetic, which many real-life hash functions actually do use