15-110 Exam2 Notes Sheet

Lists and Methods

List (`list`): ordered collection of data values (`[1, 'a']`)
2D List: a list containing other lists

List operations: `+`, `*`, `[]`, `[:]`, `in`

List functions:
`len(L)` - # items in `L`
`min(L)`/`max(L)` - min/max item in `L`
`sum(L)` - sum of items in `L`
`random.choice(L)` - random item in `L`

Method: a function called directly on a data value

```
value.method(args)
```

Methods:
`s.isdigit()`/`s.islower()`/`s.isupper()` - checks that property of `s`
`L.count(item)` - # times `item` appears
`L.index(x)` - index of `x`, error if missing
`s.lower()`/`s.upper()` - makes new version of `s` that is lowercase/uppercase
`s.replace(a, b)` - new version of `s` with `a` replaced by `b`
`s.strip()` - new version of `s` with extra whitespace removed
`s.split(delim)` - makes a list of parts of `s` separated by `delim`
`delim.join(L)` - makes a string of parts of `L` joined by `delim`

References and Memory

Reference: an address in memory. Connects a variable to its value.
Memory: sequences of bytes where data values are stored.
Aliased: two variables that share the same reference.

Mutable: a data type that can have data values modified directly in memory, often via methods.
Immutable: a data type that cannot be modified directly in memory; all changes must be made by changing the variable reference.

Mutating: a type of action that updates a data structure by modifying values in memory.
Non-mutating: a type of action that updates a data structure by moving the reference to a new memory location with new data values.

Mutating list approaches:
`L.append(val)` - adds `val` to end
`L.insert(pos, val)` - adds `val` into index `pos`
`L.extend(L2)` - adds elements from `L2` to end of `L`
`L.remove(val)` - removes `val` from `L`
`L.pop(pos)` - removes item at index `pos` from `L`
`L.sort()` - sorts `L`
`random.shuffle(L)` - shuffles `L`
`L[index] = value` - index assignment

15-110 Exam2 Notes Sheet

Recursion

Recursion: an algorithmic technique where you solve a problem through use of delegation instead of iteration.
Base case: a problem state that is so simple, it can be solved immediately.
Recursive case: a problem state that can be solved by recursively solving a smaller version of the problem, then combining that solution with the leftover part.

```
# Simple Recursion Template
def recursiveFun(problem):
  if ____: # base case
    return ____
  else: # recursive case
    smaller = ____
    result = recursiveFun(smaller)
    return ____
```

RecursionError: an error that occurs when a recursive function never reaches the base case.

Multiple recursive calls: a technique where you call the function multiple times in the recursive case instead of just once. Fibonacci and Towers of Hanoi use multiple recursive calls to simplify problem solving.

Search Algorithms I

Linear Search: a search algorithm where you search a list for an item by checking each item sequentially from left to right.

Binary Search: a search algorithm where you search a sorted list for an item by checking in the middle, then eliminating half the list based on comparison to the target. Repeat until target is found or there is nothing left to search.

Dictionaries

Dictionary (`dict`): collection of key-value pairs of items (`{ "a" : 1, "b" : 2 }`). Dictionaries index on key, not position.

Dictionary Operations:
`d[key]` - evaluates to value paired w/ `key`
`d[key] = value` - add/update pair w/ `key`
`key in d` - check if pair w/ `key` is in `d`

Dictionary Functions/Methods
`len(d)` - # pairs in `d`
`d.pop(key)` - remove pair w/ `key` from `d`

For-iterable loop: a for loop over an iterable value instead of a range.
Iterable: a type of value that can be looped over directly. Often composed of individual parts. Examples: strings, lists, dictionaries

```
for item in iterableValue:
    forBody
```

Runtime and Big-O Notation

Best case: an input that leads to an algorithm taking the least steps possible
Worst case: an input that leads to an algorithm taking the most steps possible

15-110 Exam2 Notes Sheet

Function family: a set of functions that all grow at a similar rate (eg, linear functions) expressed in a simplified format.
Common function families: constant, logarithmic, linear, quadratic, exponential

Big-O: a representation of the function family of the worst-case scenario for a specific algorithm. Represented as O(runtime).
Common Big-O runtimes: O(1), O(log n), O(n), $O(n^2)$, $O(2^n)$

Linear Search: O(n)
Binary Search: O(log n)

Calculating Big-O runtime: sequential and conditional statements are added together. Loops multiply number of iterations * work done by the body.

Trees

Tree: a data structure composed of nodes holding values that are connected hierarchically in a recursive manner.
Binary Tree: a tree where each node has at most two children, called left and right

Parent: the node connected directly above the current node
Children: the nodes connected directly below the current node
Root: the topmost node of a tree (with no parent)
Leaf: a node with no children

Our tree format is a recursively nested dictionary:

```
{ "contents" : nodeValue,
  "left"     : leftChildSubtree,
  "right"    : rightChildSubtree }
```

If there is no left/right child, the key maps to `None` instead.

The common algorithm structure for trees is recursive:
Base case: when the tree is a leaf, or an empty tree
Recursive Case: recursively call the function on the left child and right child (if they exist) and combine the results with the current node

Graphs

Graph: a data structure composed of nodes holding values, connected by edges (sometimes with values)

Neighbors: a pair of nodes connected by an edge.
Directed: a graph where edges can go from one node to another and not vice versa. Opposite is undirected.
Weighted: a graph where edges have values (called weights). Opposite is unweighted.

Our graph format is a dictionary mapping nodes to lists of neighbors:

```
{ nodeValue : [ neighborValue ],
  ... }
```

If the graph is weighted, neighbors are represented as value-weight pairs:

```
{ node : [ [neighborValue, weight] ],
  ... }
```

15-110 Exam2 Notes Sheet

Search Algorithms II

Linear search: an algorithm that can be performed on a tree by searching the left child, searching the right child, and combining the results. Base cases are root = target and empty tree.
Binary search tree (BST): a tree for which the left child of every node (and all its children, etc) are strictly less than the node, and the right child of every node (and its children, etc) are strictly greater than the node.
Binary search: an algorithm that can be performed on a BST by making just one recursive call - to the left if the target is smaller than the root, to the right if larger.
Binary search runtime: binary search on a BST runs in O(log n) if the tree is balanced (all left and right subtree pairs are ~ the same size), O(n) if unbalanced.

Hashed Search: a search algorithm where you search a hashtable for an item by running a hash function on the item, going to the appropriate bucket, and searching that bucket for the item. O(1) when used with a good hash function and a large-enough hashtable.
Hash function: a function that maps an immutable value to an integer. Must be consistent, and must generally map different values to different results.

Tractability

Brute force approach: an algorithmic strategy - solve a problem by generating all possible solutions and checking them.

Travelling Salesperson: a problem where you find the shortest route across all nodes in a graph. Runs in O(n!).
Puzzle Solving: a problem where you solve a jigsaw puzzle by finding an arrangement of pieces that fits all constraints. Runs in O(n!).
Subset Sum: a problem where you find a subset of numbers in a list that sums to a target number. Runs in $O(2^n)$.
Boolean Satisfiability: a problem where you find a combination of inputs that makes a circuit output 1. Runs in $O(2^n)$.
Exam Scheduling: a problem where you find an arrangement of exams across *k* timeslots such that no student has a conflict. Runs in $O(k^n)$.

Tractable: a problem is tractable if its worst-case runtime can be represented as a polynomial equation. Opposite is intractable.

Complexity class: a collection of function families that have similar efficiency for certain tasks and are bounded by (no worse than) a certain runtime.

P: a complexity class of problems that are tractable to *solve*
NP: a complexity class of problems that are tractable to *verify*
P vs NP: a big unsolved problem in CS. Are the complexity classes P and NP the same? We don't know!

Heuristic: a search technique used to find good-enough solutions to problems. Generates scores to choose next steps instead of using brute force.