15-110 Exam 2 Review

Brought to you by the TAs! :)

Dictionaries

Dictionaries

- Dictionaries store data in **pairs** by mapping **keys** to **values**.
- We'll be able to access the value by looking up the key, like how we can access a list value using its index.
- Keys must be **immutable** (numbers, strings, booleans)
- Values can be any type of data
- Making empty dictionary: d = {} or d = dict()
- Looking through a dictionary

```
for <itemVariables> in <iterableValue>:
    <itemActionBody>
```

Basic Dictionary Implementation

d = { “apples” : 3, “pears” : 4 }

Getting values:

- d[“apples”] => give us the value pair 3
- len(d) => gives us length of dictionary
- d[“ice cream”] => key error because ice cream is not a key in d

Adding/Removing values:

- d[“bananas”] = 7
  => adds new key-value pair
- d[“apples”] = d[“apples”] + 1
  => updates key-value pair
- d.pop(“pears”) => destructively removes

Searching:

- “apples” in d => returns true
- “kiwis” in d => returns false

Trees

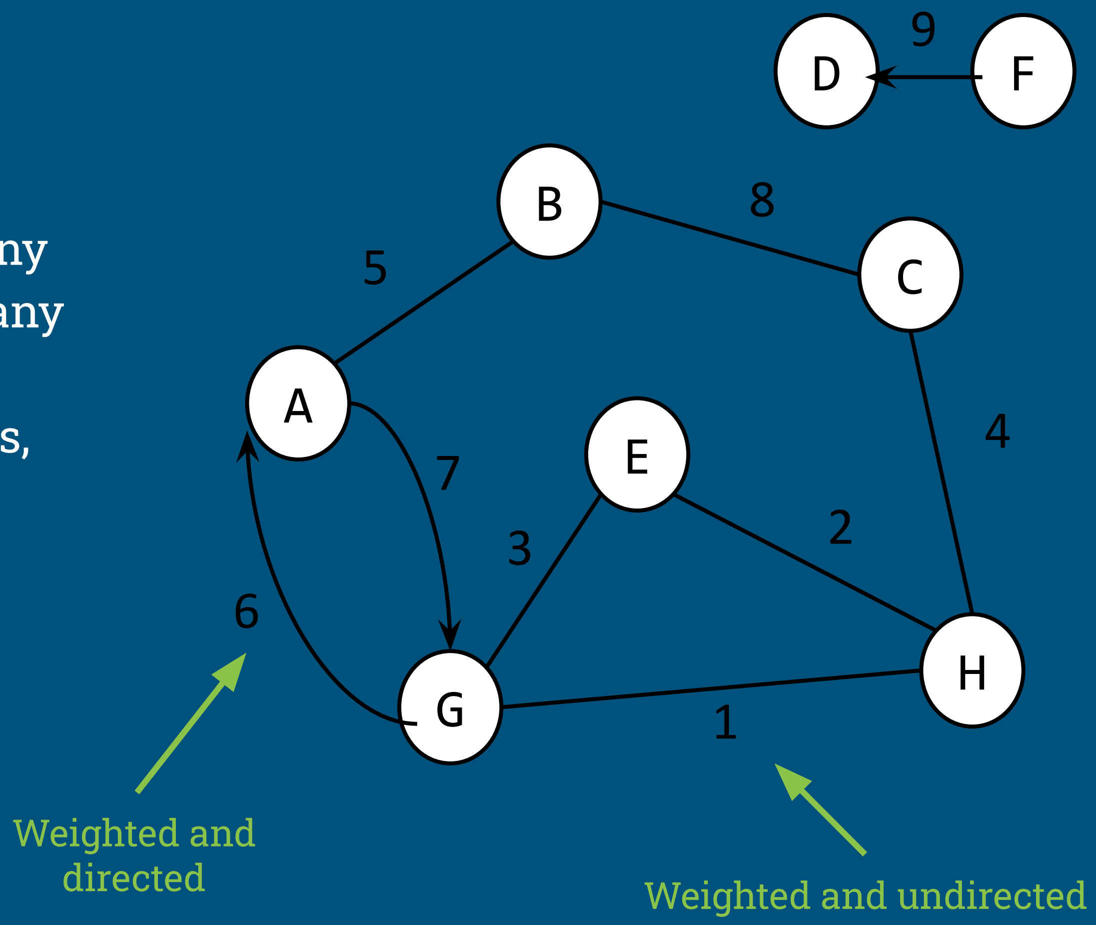
- Trees hold hierarchical data => data occurs at different levels and are connected
- Core parts of a tree include: nodes, children, the root, and leaves
- A node has exactly one parent, a parent can have any number of children
- Trees are **recursive**!
    - Each node's children are subtrees which are trees again
    - **Base case** - can be either a leaf or empty tree
    - **Recursive case** - makes problem smaller by repeating on the children
- Binary Trees: have at most 2 children per node

Coding with Trees

- Trees are implemented by recursively nested dictionaries
- Each **node** of the tree will be a dictionary that has three keys
    - First key is the string “contents” => value in the node
    - Second key is “left” => either maps to node if node has left child OR None if there is no left child
    - Third key is “right” => either maps to node if node has right child OR None if there is no right child
- Using recursion when coding with trees
    - **Base case:** when current node is a leaf and we need to do something its value
    - **Recursive case**: call function recursively on left child and then call again on right child, if they exist.

Graphs

- Graphs are like trees, but any node can be connected to any other node
- Core parts of a graph: nodes, edges, neighbors
- Edges can be weighted or unweighted
- Edges can be directed or undirected



Coding with Graphs

- The **keys** of the dictionary will be the **values** of the nodes. Each node maps to a **list of its adjacent nodes (neighbors),** the nodes it has a direct connection with.
- Weighted graphs have values associated with the edges. We need to store these values in the dictionary also
  - We'll do this by changing the list of adjacent nodes to be a **2D list.** Each of the inner lists represents a node/edge pair, so it has two values – the adjacent node's value and the weight of the edge.

Big O

Best Case & Worst Case

- Best case:
  - an input of size n that results in the algorithm taking the least steps possible.
- Worst case:
  - an input of size n that results in the algorithm taking the most steps possible.

Consider a function that takes in a list of strings as an input and uses linear search to return the second occurence of the string, "a" in the list.

What's the best case?

What's the worst case?

Big O

- When determining which Big O represents the actions taken by an algorithm, we say that **n is the size of the input**
  - For a list, that's the number of elements
  - For a string, that's the number of characters
- To determine an algorithm's Big O, you **ignore constant factors and smaller terms**
  - 3n + 8 is just O(n)
  - 4n^2 is just O(n^2)
- Big O is generally the **worst case** runtime of an algorithm

How to Calculate Big O

- Calculate the Big-O of each line/action of a function
  - Add sequential and conditional statements
  - Multiply the actions within a loop by the number of iterations performed
  - Get rid of constants and smaller terms!
- Watch out for built-in functions!
  - L.count(elem) # O(n)
  - L.remove(elem) # O(n)
  - L.pop(0) # worst case O(n)

Practice:

```
def f(L):
  result = []
  for i in range(len(L)):
    for j in range(5):
      if L[i] * j == 20:
        x = L.pop(0)
        result.append(x)
  return result
```

Linear Search & Binary Search

```python
def linSearch(lst, target):
  if len(lst) == 0:
    return False
  elif lst[0] == target:
    return True
  else:
     return linSearch(lst[1:], target)
```

O(n)

```python
def binSearch(lst, target):
  if lst == [ ]:
    return False
  else:
    mid = len(lst) // 2
    if lst[mid] == target:
       return True
    elif target < lst[mid]:
       return binSearch(lst[:mid], target)
    else: # lst[mid] < target
       return binSearch(lst[mid+1:], target)
```

O(logn) comparisons, what about runtime?

Example Problem

Come up with runtime of each line and provide overall runtime

```
def f(s): #N == len(s)
    count=0 #O()
    for c in 'aeiou': #O()
        if s.find(c)==True: #O()
            count+=1 #O()
    if count>3: #O()
        for i in range(count**2): #O()
            print(i) #O()
    return count #O()
```

Example Problem

Come up with runtime of each line and provide overall runtime

```
def f(s): #N == len(s)
    count=0 #O(1)
    for c in 'aeiou': #O()
        if s.find(c)==True: #O()
            count+=1 #O()
    if count>3: #O()
        for i in range(count**2): #O()
            print(i) #O()
    return count #O()
```

Example Problem

Come up with runtime of each line and provide overall runtime

```
def f(s): #N == len(s)
    count=0 #O(1)
    for c in ‘aeiou’: #O(1)
        if s.find(c)==True: #O()
            count+=1 #O()
    if count>3: #O()
        for i in range(count**2): #O()
            print(i) #O()
    return count #O()
```

Example Problem

Come up with runtime of each line and provide overall runtime

```
def f(s): #N == len(s)
    count=0 #O(1)
    for c in 'aeiou': #O(1)
        if s.find(c)==True: #O(N)
            count+=1 #O()
    if count>3: #O()
        for i in range(count**2): #O()
            print(i) #O()
    return count #O()
```

Example Problem

Come up with runtime of each line and provide overall runtime

```
def f(s): #N == len(s)
    count=0 #O(1)
    for c in 'aeiou': #O(1)
        if s.find(c)==True: #O(N)
            count+=1 #O(1)
    if count>3: #O()
        for i in range(count**2): #O()
            print(i) #O()
    return count #O()
```

Example Problem

Come up with runtime of each line and provide overall runtime

```
def f(s): #N == len(s)
    count=0 #O(1)
    for c in 'aeiou': #O(1)
        if s.find(c)==True: #O(N)
            count+=1 #O(1)
    if count>3: #O(1)
        for i in range(count**2): #O()
            print(i) #O()
    return count #O()
```

Example Problem

Come up with runtime of each line and provide overall runtime

```
def f(s): #N == len(s)
    count=0 #O(1)
    for c in ‘aeiou’: #O(1)
        if s.find(c)==True: #O(N)
            count+=1 #O(1)
    if count>3: #O(1)
        for i in range(count**2): #O(1)
            print(i) #O()
    return count #O()
```

Example Problem

Come up with runtime of each line and provide overall runtime

```
def f(s): #N == len(s)
    count=0 #O(1)
    for c in 'aeiou': #O(1)
        if s.find(c)==True: #O(N)
            count+=1 #O(1)
    if count>3: #O(1)
        for i in range(count**2): #O(1)
            print(i) #O(1)
    return count #O()
```

Example Problem

Come up with runtime of each line and provide overall runtime

```
def f(s): #N == len(s)
    count=0 #O(1)
    for c in 'aeiou': #O(1)
        if s.find(c)==True: #O(N)
            count+=1 #O(1)
    if count>3: #O(1)
        for i in range(count**2): #O(1)
            print(i) #O(1)
    return count #O(1)
```

Example Problem

Come up with runtime of each line and provide overall runtime

OVERALL RUNTIME: O(N)

```
def f(s): #N == len(s)
    count=0 #O(1)
    for c in 'aeiou': #O(1)
        if s.find(c)==True: #O(N)
            count+=1 #O(1)
    if count>3: #O(1)
        for i in range(count**2): #O(1)
            print(i) #O(1)
    return count #O(1)
```

Tractability

Is the problem tractable?

- A problem is tractable if it has a **reasonably efficient runtime**
- "Reasonably efficient" means the runtime can expressed as polynomial equation
  - Tractable: O(1), O(logn), O(n), O(n^2), O(n^k)
  - Intractable: O(2^n), O(k^n), O(n!)

Why does it matter?

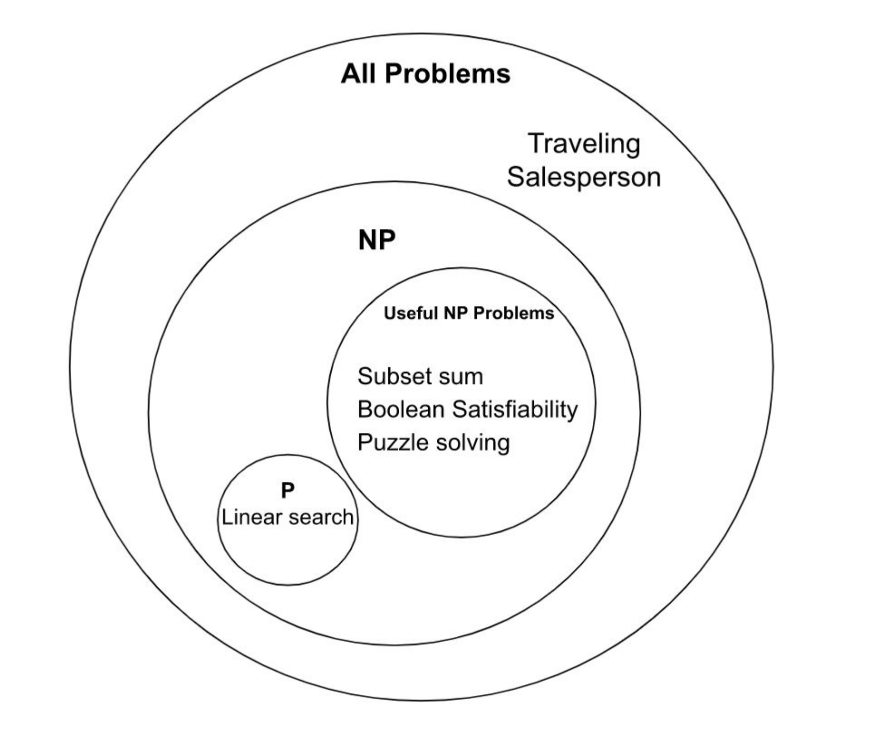
For some problems, we have to use a **brute force approach** (generating every possible solution and checking each of the generated solutions to see if any of them work for the problem's constraints)

If the **size of an input is extremely large**, using an algorithm a runtime that is not in polynomial time can take far **too long**.

Complexity Classes

- P is the set of problems that can be
    - Solved in polynomial time (tractable)
    - Checked in polynomial time (tractable)
- NP is the set of problems that can be
    - Checked in polynomial time (tractable)

Why does it matter? If P = NP, we could solve a lot of difficult problems.



Heuristics

- A **heuristic** is a technique used to find a solution that is "good enough"
- Typically used for NP problems where finding the solution is intractable
- A heuristic can rank potential next steps to help with each decision

Example:

Consider a weighted graph with nodes consisting of CMU building names and edges having weights representing the distances between the buildings.

Problem: Find the best possible path from Gates to Hall of Arts.

Heuristic: Always trying to take the shortest path first (edge with the lowest weight)