|  |
| --- |
| 15-110 Recitation 7 |

# Recap

* Dictionaries
* Trees
* Sorting

# Reminders for Students

* Check 4 due Monday 10/17 @ Noon EDT
* Regrades for Check 3 and HW 3 due Tuesday 10/18 @ Noon EDT
* Exam 3 is Wednesday 10/19
	+ Small group review sessions
* Download the starter code for this recitation off the course website!

|  |
| --- |
| Problems |

# MERGESORT REVIEW

Trace merge sort on this list [7, 5, 2, 3, 9, 1]

What is the result of the first divide? Second? First merge?

Answers/Notes

Does the runtime improve if the list is already sorted in ascending order? Descending order?

**Thinking through runtime…**

How many steps per pass? Answer

How many passes? Answer

Total # steps? Answer

Total complexity? Answer

# DICTIONARY CODE WRITING: mostWins

We want to write a function for the following problem: **Given a list of wins by CMU, Pitt, OSU, PennState’s, and another unspecified number of football teams, return the team with the most wins. There will be no ties.**

mostWins(["OSU", "PennState", "PennState", "CMU", "OSU", "OSU", "Pitt"])

 -> "OSU"

mostWins(["PennState", "PennState", "MIT", "Stanford", "UF"])

 -> "PennState"

First, work through an O(n^2) solution in the starter file under **mostWinsSlow!** (Hint: use a list method to count the number of times a team appears in the list)

def mostWinsSlow(L):

While this is a valid solution, we can solve this problem even quicker. To do this, we use dictionaries!

Dictionary Review:

* Key-Value Pairs: Dictionaries store information in key-value pairs - you can access specific values in the dictionary by looking up the key! It’s similar to how you use an integer index to look for a specific value in a list, except dictionary keys can be more complex IMMUTABLE types like strings.

#create empty dictionary

d = dict() #or d={}

#add keys/values to dictionary

d["Jasmine"] = 32 # d -> {"Jasmine":32}

d["Darryl"] = 15 # d -> {"Jasmine":32, "Darryl":15}

#update or change values

d["Darryl"] = d["Darryl"] + 1 # d -> {"Jasmine":32, "Darryl":16}

d["Jasmine"] = 12 # d -> {"Jasmine":12, "Darryl":16}

* Why do we use dictionaries? We can search for a specific key in a dictionary in constant O(1) time (You’ll learn more about this on next Wednesday)! Once we know the key, we can get the corresponding value in constant time as well.

Now that you know more about dictionaries, work through an O(n) solution in the starter file under **mostWinsFast!** (Hint: store the counts of each team in a dictionary, and then look through all the teams (keys) in the dictionary to see which one has the greatest associated value)

def mostWinsFast(L):

What is the runtime of this function mostWinsFast?

# BINARY TREES CODE WRITING

In this class, the only trees you will be dealing with are binary trees:

* A binary tree is represented as a dictionary with 3 key-value pairs
	+ “value” - corresponds to the value of the current node
	+ “left” - the left subtree extending from the current node
	+ “right” - the right subtree extending from the current node
* If a given node has no left subtree, the “left” key will have a value of None - same goes for the right subtree of a given node

For the following two questions, you are given that the input tree will not be None.

**printLeaves** Write a function given a dictionary representation of a binary tree, that prints the leaves of the tree. Do not print the values of the non-leaf nodes of the tree.

def printLeaves(tree):

**countNodes** Write a function given a dictionary representation of a binary tree, that returns the number of nodes in the tree.

def countNodes(tree):