# Practical Questions

CSCA48 – Week 9

Here are some questions on heaps similar to the ones in class. Make sure you are confident with them. Letters are ordered alphabetically. That is, A < B < C <… < Y < Z

a) Insert the letters U,N,I,V,E,R,S,I,T,Y (in order listed) into a heap.
Show both the tree and the list after each step.

b) Extract each letter from your heap in part a).
Show both the tree and the list after each step.

c) Is it possible to insert the letters in {U,N,I,V,E,R,S,I,T,Y} (any order can be considered) into a heap so that all the consonants (non-vowels) are in the right sub-tree of the root?
If so, explain how, If not, explain why not

d) Is it possible to insert the letters in {U,N,I,V,E,R,S,I,T,Y} (any order can be considered) into a heap so that all the consonants (non-vowels) are in the left sub-tree of the root?
If so, explain how, If not, explain why not

e) Is it possible to insert the letters in {U,N,I,V,E,R,S,I,T,Y} (any order can be considered) into a heap so that, within the list, all the consonants are next to each other (i.e., there are no vowels between any pair of consonants)?
If so, explain how, If not, explain why not

f) Is it possible to insert the letters in {U,N,I,V,E,R,S,I,T,Y} (any order can be considered) into a heap so that, within the list, all the vowels are next to each other (i.e., there are no consonants between any pair of vowels)?
If so, explain how, If not, explain why not

g) Is it possible to insert the letters in {U,N,I,V,E,R,S,I,T,Y} (any order can be considered) into a heap so that, within the list, no vowel is next to another vowel and no consonant is next to another consonant?
If so, explain how, If not, explain why not

h) How many ways can the letters in {U,N,I,V,E,R,S,I,T,Y} (any order can be considered) so that no bubbling occurs? Explain your answers.

Consider the following algorithms

i) Inserting a node to the front of a singly linked list. Using Big-Oh notation, what is the Worst Case? How about the Best Case?

j) Deleting a node from a singly linked list. Using Big-Oh notation, what is the Worst Case? How about the Best Case?

k) Determine whether integer i is greater than all integer in a list of integers. Using Big-Oh notation, what is the Worst Case? How about the Best Case?

l) Determine whether there are duplicate (repeated) in a list of integers. Using Big-Oh notation, what is the Worst Case? How about the Best Case?

m) Comparing each integer to every integer in a list of integers. Using Big-Oh notation, what is the Worst Case? How about the Best Case?

Trace the follow functions carefully. Make sure you understand how they work!

n)

```
def find(L, item):
    for i in range(len(L)):
        if L[i] == item:
            return i
```

Why is the Best Case O(1)? And Why is the Worst Case O(n)? explain your reason
What about the Average Case?

o)

```
def bubble_sort(L):
    swapped = False
    for i in range(len(L)-1):
        for j in range(len(L)-1):
            if L[j] > L[j+1]:
                temp = L[j]
                L[j] = L[j+1]
                L[j+1] = temp
                swapped = True
        if not swapped:
            return None
```

Why is the Best Case O(n)? Why is the Worst Case O(n^2)? Explain your reason
What about the Average Case?

p)
```
def selection_sort(L):
    for i in range(len(L)-1):
        for j in range(i+1, len(L)):
            if L[i] > L[j]:
                temp = L[i]
                L[i] = L[j]
                L[j] = temp
```

Why is the Best Case O(n^2)? Why is the Worst Case O(n^2)? Explain your reason
Comparing to the Bubble Sort from part o), Why is the Best Case different?
Can we terminate the loop by checking if swaps have been made just like part o)? Why or Why not.

In Python, slicing or cloning a list actually walks through every element once. This takes an O(n) time to make a copy of a list or a sub list. O(n) seems to be fast but in a recursive function, Umm…. not that good!

q) Complete rsum(L) from ex5 without any list slicing.

r) Complete rmax(L) from ex5 without any list slicing.

s) Complete second_smallest(L) from ex5 without any list slicing.

t) Complete sum_min_max(L) from ex5 without any list slicing.

**u) <u>(Challenge)</u>**
Complete the above functions to accept nested lists. That is, complete ex6 again without any list slicing.

More recursion practices!

v) (**Challenge)** Complete the following function **recursively.**

```
def pack (L, n):
    '''Return the subset of L with the largest sum up to n
    >>> L = [4 , 1 , 3 , 5]
    >>> pack(s, 7)
    {3 , 4}
    >>> pack(s, 6)
    {1 , 5}
    >>> pack(s, 11)
    {1 , 4 , 5}
    '''
```

**w) (Challenge)** Suppose you have a string made up only the letters ‘a’ and ‘b’. Write a recursive function that checks if the string was generated using the following rules:

- The string begins with an ‘a’
- Each ‘a’ is followed by nothing or an ‘a’ or “bb”
- Each “bb” is followed by nothing or an ‘a’

**Discussion Question (Challenge)**

1. Why do we want to analyze the speed/efficiency of an algorithm?
    a. What do we mean when we say "algorithm A is faster than algorithm B"?
    b. What sort of speed differences do we care about?
        - How much do we care if algorithm B takes a constant amount of time more than algorithm A?
        - How much do we care if algorithm B takes a constant factor longer than algorithm A?
    c. What sort of analysis would we find useful? What's practical?
2. How do we measure the speed/efficiency of an algorithm?
    a. Should we code it in python and use a stopwatch? Why or why not?
3. What do we mean by running time of an algorithm?
    a. Running times of most algorithms depend on the size of its input.
    E.g., the time to sum a linked list of numbers depends on the length of the list -- a longer list would take longer to sum.
    How do we capture this notion in our definition of running time?
    b. Running times of many algorithms also depend on the specific input.
    E.g., the time to search for an item in a linked list depends on where in the list that item is found -- it would take longer to find if the item is nearer to the end of the list (or it's not even there).
    So what should we be measuring in such cases?
        - Best case?
        - Worst case?
        - Average case? What do we mean by *average*?

**Logic Question (Challenge)**

We're going to play a game. Sitting at a 1m x 1m square table, we will take turns putting coins on the table. All coins are circular with 1cm radius. Coins must be placed on the table such that at least 50% of the coin is on the table or else it will fall off. Whoever puts the last coin on the table such that their opponent cannot play, will win the game and all the coins.

If you go first, what is your strategy?

If you go second, what's your strategy?