Section 5

Thursday, April 25, 2019 4:26 PM

Section 05: Midterm Review

1. Asymptotic Analysis

Anything Difficult

Given a list of integers `lst`, return a set's which that `f(x)==f(y)`. Explain why your runtime of and `lst` is `O(n)`.

```
def find_pairs(lst):
    seen = set()
    for x in lst:
        if f(x) in seen:
            return True
        seen.add(f(x))
    return False
```

Runtime Analysis

- Iterate through the list, adding each element to a set if it hasn't been seen before.
  - If we ever get to a element we've seen before, return True.
  - Otherwise, if we get through the whole list, return False.
- Runtime Analysis
  - Linear Scaling.

`O(n)` Not `O(n^2)` because implementing a set/dictionary implemented as an array.

- Finding the minimum value in a BST: `O(h)`
- Finding the minimum value in a DLL: `O(n)`
- Finding the minimum value in a sorted array: `O(1)`
- Iterating through and printing every element in an array/ list using a for loop and the `get()` method: `O(n)` Due to a hash containing a dictionary implemented as a single linked list.
- Finding the minimum value in a set of size `n`, where the value was just removing a smaller than any other value currently in the set: `O(1)`

2. Eyeballing Big-O bounds `n = len(lst)`

For each of the following code blocks, what is the tightest bound? Write a big-O bound. You do not need to justify your bound.

```
def foo(lst):
    for i in range(len(lst)):
        for j in range(len(lst)):
            print(lst[i] + lst[j])
```

```
def bar(n):
    if n == 0:
        return 0
    else:
        return 1 + bar(n-1)
```

```
def baz(n):
    if n == 0:
        return 0
    else:
        for i in range(n):
            print("hello")
        return baz(n-1) + baz(n-1)
```

```
def qux(n):
    if n == 0:
        return 0
    else:
        for i in range(n):
            print("hello")
        return qux(n//2) + qux(n//2)
```

```
def quux(lst):
    for i in range(len(lst)):
        if lst[i] == 42:
            return True
    return False
```

```
def quuz(lst):
    for i in range(len(lst)):
        if lst[i] == 42:
            return True
    for i in range(len(lst)):
        print("hello")
    return False
```

```
def corge(n):
    if n == 0:
        return 0
    else:
        for i in range(n):
            for j in range(n):
                print("hello")
        return corge(n-1)
```

```
def grault(n):
    if n == 0:
        return 0
    else:
        for i in range(n):
            print("hello")
        return grault(n//2) + grault(n//2) + grault(n//2) + grault(n//2)
```

```
def garply(n):
    if n == 0:
        return 0
    else:
        for i in range(n):
            print("hello")
        return garply(n-1) + garply(n-1) + garply(n-1) + garply(n-1)
```

3. Recurrences and Summations

Find an expression for the total work of the following expressions using the Case Method. Over simplify that sum and express the total work in terms of the input variable `n`. Show your work.

Note: Not all of examples. Just use the ones you should go through the Case Method Problems.

1. Draw the recurrence tree.
2. Write a summation for the amount of work on each level. Write the amount of work done by each node in the tree multiplied by the number of nodes that exist at level `i`. This means that `i = 0` should be a summation for the work done by the root node.
3. Simplify the summation.
4. Where do the total number of nodes at level `i` in the tree, write out the sum from `i = 0` to `i = n`. Simplify the summation. Express the total work in terms of `n`.
5. What is the total work done across the all recursive work?
6. What value of `i` does the last level of the tree arise at?
7. What is the total work done across the all recursive work in the last `n` levels, the last level?
8. Combine your answers from previous parts to get an expression for the total work.

(a) $T(n) = \begin{cases} 2T(\frac{n}{2}) + n & n > 0 \\ 1 & \text{otherwise} \end{cases}$

(b) $T(n) = \begin{cases} T(n-1) + n^2 & n > 0 \\ 1 & \text{otherwise} \end{cases}$

(c) $T(n) = \begin{cases} 2T(n-1) + n & n > 0 \\ 1 & \text{otherwise} \end{cases}$

i) $n \cdot 2^i$

ii) $\sum_{i=0}^{n} n \cdot 2^i$

iii) $n \cdot \sum_{i=0}^{n} 2^i$

iv) $\frac{1}{2-1} \times (2^{i+1} - 1)$

v) $(n+1)(2^{i+1} - 1)$

vi) $\frac{1}{2-1} \times \frac{(2^{n+1} - 1)}{\text{work per node}}$

vii) $\sum_{i=0}^{(n+1)-1} \frac{(2^{i+1} - 1)}{\text{Base Case}} + \frac{57}{\text{Base Case work}}$

viii) $\sum_{i=0}^{(n+1)-1} n^2 - 2n + 2$

$\sum_{i=0}^{(n+1)-1} n^2 - (n+1)n$

$\sum_{i=0}^{(n+1)-1} 2n$

$\sum_{i=0}^{(n+1)-1} 2$

$-2n(n+1)(n+1-1)$

$2n$

$+ \frac{(n+1)^3 (n+1)(n+1-1) - (2(n+1)-1)}{6} + 57$

4. Modeling

Consider the following method. `lst` is a list integer value of the `n` parameter, and `lst` is the length of `lst`. `n == len(lst)`. This may assume that `n`.

```
def mystery(lst, n):
    output = [[0, 0, 0, 
```