TUTORIAL 8

O(\_\_)

EFFICIENCY
REMINDERS

• No Assignment due next week!
  – Assignment 7 due Wednesday, July 17th, at 10:00 AM
RUNTIME REVIEW

- Look at the “worst case” scenario (i.e. longest runtime)
- Assume function works (i.e. will not return an error when you run it)
- Based on the assumptions learned in class (and in the modules)
RUNTIME REVIEW

• **O(1)** – Constant
  – does not depend on the size of the input
  – For numbers:
    • Numeric operations: +, *, /, -, %, //
    • max, min
  – For list L:
    • L[0], len(L)…
    • L.append(4)…

• **O(n)** – Linear
  – depends on the size of the input
  – For list L(assume the length of L is n):
    • L[1:], max(L), L + L, sum(L), L.remove(0)…
    • list(map(lambda x: x+1, L))
RUNTIME REVIEW

• \(O(n^2)\) – Quadratic
  – time proportional to square of size of the input
  – Be careful of abstract functions:
    • \(\text{list(map(lambda k: list(range(k)), list(range(n))))}\)

• \(O(2^n)\) – Exponential
  – As size of input increases by 1, the run time doubles
  – example: Module 5, Slide 15: \(\text{fib}\)
RECURRENCE RELATIONS

• \( T(n) = O(1) + T(n-1) \rightarrow O(n) \)
• \( T(n) = O(n) + T(n-1) \rightarrow O(n^2) \)
• \( T(n) = O(1) + T(n/2) \rightarrow O(\log n) \)
• \( T(n) = O(n) + 2T(n/2) \rightarrow O(n \log n) \)
  - \( T(n) = O(n) + 2T(n/2) \rightarrow O(n \log n) \)
• \( T(n) = O(1) + T(n-1) + T(n-2) \rightarrow O(2^n) \)
  - \( T(n) = O(1) + 2T(n-1) \rightarrow O(2^n) \)
  - \( T(n) = O(n) + T(n-1) + T(n-2) \rightarrow O(2^n) \)
  - \( T(n) = O(n) + 2T(n-1) \rightarrow O(2^n) \)
USEFUL SUMMATIONS

- $\sum_{i=1}^{n} 1 = O(n)$
- $\sum_{i=1}^{n} i = O(n^2)$
- $\sum_{i=1}^{n} n = O(n^2)$
- $\sum_{i=1}^{n} \sum_{j=1}^{n} 1 = O(n^2)$
RUNTIME EXAMPLE 1

# Let n = len(L)
def fn(L):
    if L==[]:
        return 0
    else:
        return 1 + fn(L[1:])

Count steps for:
• Compare L with []
• Calculate L[1:]
• Call fn recursively on a list of length n-1
• Add 1 to the recursive call of fn

• T(n) = O(n) + T(n-1) => O(n^2)
# Let \( n = \text{len}(L) \)

def fn(L):
    ans = []
    for x in L:
        if x[0] == 'A':
            ans.append(x)
    return ans

Count steps for:

- Assign \([\])\ to \(\text{ans}\)
- Loop:
  - Number of Iterations
  - Asymptotic run time of the body of loop:
    - Check if \(x[0] == 'A'\)
    - \(\text{ans}.\text{append}(x)\)
- Return \(\text{ans}\)
- \(\sum_{i=1}^{n} 1 = O(n)\)
# Let n = len(L)

def fn(L):
    L1 = L[0::2]
    if L == []:
        return []
    else:
        return fn(L1)

Count steps for:

• L1 = L[0::2]
• Compare L with []
• Call fn recursively on a list of length n//2
• T(n) = O(n) + T(n/2) => O(n)
def fn(n):
    if n % 2 == 0:
        return "outcome1"
    elif n % 3 == 0:
        return "outcome2"
    elif n % 5 == 0:
        return "outcome3"
    else:
        return "outcome4"

Count steps for:

- Calculate n%2
- Compare it with 0
- Calculate n%3
- Compare it with 0
- Calculate n%5
- Compare it with 0
- Return the answer
- O(1)
QUESTION 2 - QUICKSORT

Consider a different way of sorting a list \( L \) of distinct integers:

- Let \( x \) be the first element of the list
- Let \( lst1 \) be all the elements in the list smaller than \( n \)
- Let \( lst2 \) be all the elements in the list larger than \( n \)
- Recursively quicksort \( lst1 \) and \( lst2 \)
- \( lst1 + [x] + lst2 \)

Write a function \texttt{quicksort} which consumes a list of distinct integers, \( lst \), and sorts it using the quicksort algorithm.
EXAMPLE

quicksort([2,3,1,4,0])

→ quicksort([1,0]) + [2] + 
   quicksort([3,4])

→ (quicksort([0]) + [1]) + [2] + 
   ([3] + quicksort([4]))

→ ([0] + [1]) + [2] + ([3] + [4])

→ [0, 1] + [2] + [3, 4]

→ [0, 1, 2, 3, 4]
RUNTIME OF QUICKSORT

• **Worst case runtime:**
  - $T(n) = O(n) + T(n-1) \Rightarrow O(n^2)$
  - The list is already sorted

• In practice, quicksort can avoid the worst case most of the time, and, on average, runs in $O(n \log n)$ time.